tractual obligations, accounting periods, customer service, and sales lead follow-up. Typical workflow goals are shorter time to market and quicker response times.

Are Multiple People Involved in the Business Process?

Workflow co-ordinates cross-functional, cross- departmental work activities and promotes accountability. It also enables dynamic redistribution and reprioritization of work.

Is There a Need for Work Scheduling?

Workflow management can be extended to automate work scheduling. A system may be able to do as good a job, or better, in scheduling a users work. This might be due to a very large amount of work to be assigned to a large pool, a complex method of assigning priorities, an extremely dynamic environment, or some other reason. Another advantage to work scheduling is that the system can initiate some needed activity automatically for the user in anticipation of the next task.

Do Integration Issues Exist?

It is important to determine how well the workflow system integrates with host-based hardware, system software, database management systems, and communication networks. Examples of items to consider include E-mail, database, GUI tool, PC applications, other office systems, and business applications.

How Scaleable is the Product?

Number of workers the product could reliably support in a production environment. Two major product factors characterize scalability: (1) Platform alternatives (hardware and operating system); and (2) Message-based architecture (relying on specific mail systems for much of the functionality) versus Database-based.

What is the Nature of the Workflow?

How an organization approaches the management of its workflow will determine which workflow management tools are appropriate to the organization. In general, there are three types of workflow, production, collaborative, and ad hoc. A production environment involves high transaction rates and thousands of documents in which the rules for a certain document can be defined for most of the time. Examples include accounts payable, insurance claims processing, and loan processing. A collaborative environment involves multiple departments viewing a single document with typically less number of documents than in the production environment. One example is a sales order. Ad hoc workflows arise from the specific temporary needs of a project team whose members become active and inactive depending on their function within the group.

What is the Relationship Between the Workflow and Imaging Components?

It may be important to determine whether or not the products work routing function is integrated and inseparable from document storage and retrieval functions.

What are the Necessary Functions and Features?

Issues to consider include the following: (1) samples and assists that are available to the developer; (2) existence of a scripting or programming language; (3) granularity of the security, or in other words, at what levels can security be added; (4) freedom of choosing productivity applications; (5) existence of aggregate functions which allow for analysis of the workflow efficiency; (6) existence/need for Business Processing Re-engineering tools.

How Stable is the Vendor?

One should consider the leadership and size characteristics of the products vendor compared to the workflow software marketplace. Another consideration is whether the vendor is a member of Workflow Management Coalition. This coalition is beginning to have a bigger impact on the direction of vendors workflow management products.

How Mature is the Product?

One should consider the age, release, and installed base of the product.

How Flexible is the Product?

A product should be able to support changing workflows at various levels of detail.

BUSINESS LOGIC **1022**, **1024**

The execution architecture services are all generalized services designed to support the applications Business Logic. How Business Logic is to be organized is not within the scope of the execution architecture and must be determined based upon the characteristics of the application system to be developed. This section is intended to serve as a reminder of the importance of consciously designing a structure for Business Logic which helps to isolate the impacts of change, and to point out that the underlying Netcentric architecture is particularly well suited for enabling the packaging of Business Logic as components.

Business Logic is the core of any application, providing the expression of business rules and procedures (e.g., the steps and rules that govern how a sales order is fulfilled). As such, the Business Logic includes the control structure that specifies the flow for processing business events and user requests. There are many ways in which to organize Business Logic, including: rules-based, object-oriented, components, structured programming, etc. however each of these techniques include, although perhaps not by name, the concepts of: Interface, Application Logic, and Data Abstraction. FIG. **33** depicts the various components of the Business Logic portion of the Netcentric Architecture Framework.

Interface Logic **(3302)**

Interface logic interprets and maps the actions of users into business logic processing activities. With the assistance of Presentation Services, Interface logic provides the linkage that allows users to control the flow of processing within the application.

Application Logic (b**2504**)

Application Logic is the expression of business rules and procedures (e.g., the steps and rules that govern how a sales order is fulfilled). As such, the Application Logic includes the control structure that specifies the flow for processing for business events and user requests. The isolation of control logic facilitates change and adaptability of the application to changing business processing flows.

Data Abstraction (b**2506**)

Information Access Services isolate the Business Logic from the technical specifics of how information is stored (e.g., location transparency, RDBMS syntax, etc.). Data Abstraction provides the application with a more logical view of information, further insulating the application from physical information storage considerations.

The developers of business logic should be shielded from the details and complexity of other architecture services (e.g., information services, component services), and other business logic for that matter.

It is important to decide whether the business logic will be separate from the presentation logic and the database access logic. Today separation of business logic into its own tier is often done using an application server. In this type of an environment, although some business rules such as field validation might still be tightly coupled with the presentation logic, the majority of business logic is separate, usually residing on the server. It is also important to decide whether the business logic should be packaged as components in order to maximize software re-use and to streamline software distribution.

Another factor to consider is how the business logic is distributed between the client and the server(s)—where the

SAP - Exhibit 1105
SAP v. Valtrus et al. - IPR2025-00420
Page 185 of 281

business logic is stored and where the business logic is located when the application is being executed. There are many ways to distribute business logic: (1) business logic can be stored on the server(s) and executed on the server(s); (2) business logic can be stored on the server(s) and executed on the client; (3) business logic can be stored and executed on the client; (4) some business logic can be stored and executed on the server(s) and some business logic can be stored and executed on the client; etc.

Having the business logic stored on the server enables developers to centrally maintain application code; thereby eliminating the need to distribute software to client machines when changes to the business logic occur. If all the business logic executes on the server, then the application on the client will make requests to the server whenever it needs to execute a business function. This could increase network traffic, which may degrade application performance. On the other hand, having the business logic execute on the client, may require longer load times when the application is initially launched. However, once the application is loaded, most processing is done on the client until synchronization with the server is needed. This type of an architecture might introduce complexities into the application that deal with the sharing of and reliance on central data across many users.

If the business logic is stored and executed on the client, software distribution options must be considered. Usually the most expensive option is to have a system administrator or the user physically install new applications and update existing applications on each client machine. Another option is to use a tool that performs automatic software distribution functions. However, this option usually requires the software distribution tool to be loaded first on each client machine. Another option is to package the application into ActiveX controls, utilizing the automatic install/update capabilities available with ActiveX controls—if the application is launched from a Web browser.

Currently, Internet applications house the majority of the business processing logic on the server, supporting the thin-client model. However, as technology evolves, this balance is beginning to shift, allowing business logic code bundled into components to be either downloaded at runtime or permanently stored on the client machine. Today, client side business logic is supported through the use of Java applets, JavaBeans, Plug-ins and JavaScript from Sun/Netscape and ActiveX controls and VBScript from Microsoft.

The developers of business logic should be shielded from the details and complexity of other architecture services (e.g., information services, component services), and other business logic for that matter.

It is important to decide whether the business logic will be separate from the presentation logic and the database access logic. Today separation of business logic into its own tier is often done using an application server. In this type of an environment, although some business rules such as field validation might still be tightly coupled with the presentation logic, the majority of business logic is separate, usually residing on the server. It is also important to decide whether the business logic should be packaged as components in order to maximize software re-use and to streamline software distribution.

Another factor to consider is how the business logic is distributed between the client and the server(s)—where the business logic is stored and where the business logic is located when the application is being executed. There are many ways to distribute business logic: (1) business logic can be stored on the server(s) and executed on the server(s); (2) business logic can be stored on the server(s) and executed on the client; (3) business logic can be stored and executed on the client; (4) some business logic can be stored and executed on the server(s) and some business logic can be stored and executed on the client; etc.

Having the business logic stored on the server enables developers to centrally maintain application code; thereby eliminating the need to distribute software to client machines when- changes to the business logic occur. If all the business logic executes on the server, then the application on the client will make requests to the server whenever it needs to execute a business function. This could increase network traffic, which may degrade application performance. On the other hand, having the business logic execute on the client, may require longer load times when the application is initially launched. However, once the application is loaded, most processing is done on the client until synchronization with the server is needed. This type of an architecture might introduce complexities into the application that deal with the sharing of and reliance on central data across many users.

If the business logic is stored and executed on the client, software distribution options must be considered. Usually the most expensive option is to have a system administrator or the user physically install new applications and update existing applications on each client machine. Another option is to use a tool that performs automatic software distribution functions. However, this option usually requires the software distribution tool to be loaded first on each client machine. Another option is to package the application into ActiveX controls, utilizing the automatic install/update capabilities available with ActiveX controls—if the application is launched from a Web browser.

Currently, Internet applications house the majority of the business processing logic on the server, supporting the thin-client model. However, as technology evolves, this balance is beginning to shift, allowing business logic code bundled into components to be either downloaded at runtime or permanently stored on the client machine. Today, client side business logic is supported through the use of Java applets, JavaBeans, Plug-ins and JavaScript from Sun/Netscape and ActiveX controls and VBScript from Microsoft.

PATTERNS

OVERVIEW OF PATTERNS

Introducing Patterns

The goal of patterns within the software community is to create a body of literature to help software developers resolve common difficult problems encountered throughout all of software engineering and development. Patterns help create a shared language for communicating insight and experience about these problems and their solutions. Formally codifying these solutions and their relationships lets us successfully capture the body of knowledge which comprises one's understanding of good architectures that meet the needs of their users. Forming a common pattern language for conveying the structures and mechanisms of architectures allows us to intelligibly reason about them. The primary focus is not so much on technology as it is on creating a culture to document and support sound engineering architecture and design.

What is a Pattern?

A pattern is a named nugget of insight that conveys the essence of a proven solution to a recurring problem within

a certain context amidst competing concerns. Patterns are a more formal way to document codified knowledge, or rules-of-thumb. Patterns represent the codified work and thinking of our object technology experts. While experts generally rely on mental recall or rules-of-thumb to apply informal patterns as opportunities are presented, the formalization of the patterns approach allows uniform documentation and transfer of expert knowledge.

Patterns are not unique to object technology or even software development, having been invented by Christopher Alexander, a building architect. However, they have not been applied to other information technology development techniques. Thus, they are an exclusive feature of object technology. Furthermore, patterns are becoming widely accepted by the worldwide object community as an important element in successfully rolling out the technology, and enabling the maturation of software development as an engineering process.

Patterns are usually concerned with some kind of architecture or organization of constituent parts to produce a greater whole. Richard Gabriel, author of Patterns of Software: Tales From the Software Community, provides a clear and concise definition of the term pattern:

> Each pattern is a three-part rule, which expresses a relation between a certain context, a certain system of forces which occurs repeatedly in that context, and a certain software configuration which allows these forces to resolve themselves.
>
> As an element in the world, each pattern is a relationship between a certain context, a certain system of forces which occurs repeatedly in that context, and a certain spatial configuration which allows these forces to resolve themselves.
>
> As an element of language, a pattern is an instruction, which shows how this spatial configuration can be used, over and over again, to resolve the given system of forces, wherever the context makes it relevant.
>
> The pattern is, in short, at the same time a thing, which happens in the world, and the rule which tells us how to create that thing, and when one must create it. It is both a process and a thing; both a description of a thing which is alive, and a description of the process which may generate that thing.

In Software Patterns; Jim Coplien writes, a good pattern may do the following:

> It solves a problem: Patterns capture solutions, not just abstract principles or strategies.
>
> It is a proven concept: Patterns capture solutions with a track record, not theories or speculation.
>
> The solution isn't obvious: Many problem-solving techniques (such as software design paradigms or methods) try to derive solutions from first principles. The best patterns generate a solution to a problem indirectly—a necessary approach for the most difficult problems of design.
>
> It describes a relationship: Patterns don't just describe modules, but describe deeper system structures and mechanisms.
>
> The pattern has a significant human component . . . All software serves human comfort or quality of life; the best patterns explicitly appeal to aesthetics and utility.

COMPONENT-BASED DEVELOPMENT

Introduction to Component Based Development

Component systems model—how the business works Component-orientation is a strategic technology that may significantly impact a user's practice and clients. Component technologies are a natural evolution from object-oriented systems providing a more mature way of packaging reusable software units. Object-oriented systems more closely support business integration framework for solution delivery by shifting design focus away from an underlying technology toward a company's business conduct and functional behaviors. Business entities are represented as objects, which package data and functional behavior. This is in distinct contrast to traditional development approaches that maintain a ubiquitous split between functional behaviors and data.

Object-orientation has accelerated into the take-up curve. All of the major commercial component models are object-oriented. In addition, all of the major vendors have adopted the "Unified Modeling Language" (UML) as a standard notation for describing object models. A tremendous reservoir of knowledge capital, practice aids and starter kits related to object and component technology can be found on the Knowledge Exchange.

More and more, users are asking for assistance to deploy Netcentric eCommerce applications based on components. These applications are frequently based on object-oriented languages like Java, Visual Basic and C++.

Objects are an easy metaphor to understand and manage. There are still substantial risks involved, particularly because component- and object-orientation has a pervasive impact on areas as broad as analysis and design, planning, and development tools.

Component-Based Overview

Component Technology Impacts Most Aspects of Development

Component and object technology impacts most aspects of software development and management. Component technology is a new technology and a driving influence in the evolution of object-oriented (OO) methodologies. The Management Considerations section of the Introduction to Component-Based Development uses the Business Integration (BI) Model to discuss the impact of OO, including:

> Strategy and planning with a long-term view towards building reusable, enterprise software assets.
>
> Technology and architecture approaches for building cohesive, loosely coupled systems that provide long-term flexibility.
>
> Processes that shift analysis/design techniques from functional, procedural decomposition to business process modeling. These techniques are then used to decompose the system into domain objects and processes.
>
> People and organization strategies that emphasize greater specialization of skills within structures that support inter-team collaboration.

Balancing Tradeoffs is key to Applying Components for Mission-critical Systems

Tradeoffs are an important theme. Experience with large, mission-critical systems has shown that the most complex issues require strategic tradeoffs between quality, cost, and time. These tradeoffs usually involve interdependent considerations between strategy, technology, process, and people. See FIG. **34** which illustrates a relationship between major themes. For example, how should an architecture be tailored to effectively support a specific methodology, for a given organization's skill set? Competing tensions also cloud decisions at a more detailed level. For example, how should an architecture be customized to better support performance, at the potential cost of increased coupling between components?

SAP - Exhibit 1105
SAP v. Valtrus et al. - IPR2025-00420
Page 187 of 281

Many of these considerations have been addressed over the last few years. Most published literature continues to focus on narrow technology issues, such as programming techniques or generic methodologies, such as analysis and design approaches or notation. Still, a growing number of publications and vendor strategies attack the enterprise needs within on-line netcentric execution models. Real-world, client solutions involve making pragmatic decisions, in which compromise occurs at the intersection of the four major OO themes. Experience with many component client projects in diverse industries uniquely positions a user to effectively address these complexities.

Management Considerations Overview

The Management Considerations section discusses the key benefits, risks, and issues introduced by a component engagement. Key topics include:

- Managing risk in balancing tradeoffs between strategy, people, process, and technology
- Considering issues related to configuration management, testing, and performance of object systems
- Addressing the component development learning curve
- Differences between development architecture considerations leveraging the advantages of a component industry.

The Management Considerations section also address issues not unique to Component technology, including:

- Estimating, planning, and managing iteration
- Organizing and managing to achieve reuse of both architecture and business logic

Netcentric Patterns Overview

Netcentric Patterns Focus on Application Frameworks

Netcentric Patterns focus on how to design and leverage application frameworks, which are pieces of reusable application architecture that provide a highly configurable, flexible and maintainable system. They are aligned with SAF and/or DAF service layers. Alignment with SAF and/or DAF makes the patterns easier to grasp the context for which they are solving problems.

There was no mandate to express implementation within any given particular OO language. Java and Visual Basic have increased in popularity over the last few years and C++ continues to be a solid foundation on which to build many types applications. In addition, some implementations chose the design syntax of UML. One should see the value of the pattern regardless of the implementation personality. Nowhere has this been more strongly demonstrated than in the Eagle Starter Kits. Here, the Eagle Architecture Specification has been documented in patterns and implemented in Visual Basic, Java, C++ and a host of execution environments within these language offerings. The power is in the reusable design patterns.

For a high-level description of the context for the patterns within a service layer of SAF and/or DAF, click the title of the section. Please refer to the SAF and/or DAF for more detailed descriptions of the service layers. From the Frameworks Main Page, under Framework Extensions, the "Component Technology Extension" describes, in the context of the Netcentric Architecture framework, the additional, specialized, architecture services that are required when building a system using component technologies.

Approach

Over the past years, component-based development has become an important, but often-misunderstood concept in the IT world. Components in themselves don't guarantee successful business applications, but coupled with a proven methodology and continuous technological advancements, they make it possible to realize a number of important benefits such as flexibility, adaptability, maintainability, reusability, integration readiness, interoperability, and scalability.

Components have been around for a long time. The wheels on an ancient Roman chariot were certainly components. When the local chariot maker invented a new wheel (one that promised greater speeds and improved reliability on a wider variety of terrain), chariot owners would replace their worn-out, inefficient, and out-dated wheels with the new ones, but only if the new ones offered, at a minimum, the same function (i.e., rolling) through the same interface (i.e., the connection between the wheel and the chariot).

Today components are used to build everything from cars to computers. In electronics, for example, they have led to the proliferation of product features, disposability, miniaturization, product selection, price reduction, and standard interfaces—all good for the consumer. This example also draws attention to some of the challenges that accompany components: setting standards, determining the right components, the need to change standard interfaces based on new requirements, and the legal and commercial structure for selling components.

Throughout the industry the word "component" is used broadly and often loosely. Components come in a wide variety of shapes and sizes. For example: JavaBeans, ActiveX controls, and COM objects. And more generically: application, architecture, development, engineering, Web, server, and business components.

Many industry experts have attempted to define "component." Unfortunately, many of these definitions are too abstract, too academic, or too specialized to be useful. Yet below the surface of these definitions is some real business value for organizations.

Experience has shown that it's quite common for people to view components from different perspectives, as illustrated in FIG. **35**. Some of them-typically designers—take a logical perspective. They view components as a means for modeling real-world concepts in the business domain. These are Business Components. Others—typically developers—take a physical perspective. They view components as independent pieces of software, or application building blocks, that implement those real-world business concepts. These are Partitioned Business Components. Developers also emphasize that Partitioned Business Components can be built from other independent pieces of software that provide functionality that is generally useful across a wide range of applications. These are Engineering Components.

To use an analogy, the designer of a PC workstation would initially think in terms of logical components such as Disk Storage, Memory, Display, etc. These are analogous to Business Components. At some point in the design process, however, this thinking must become more precise. For example, Disk Storage might become a Hard Disk Drive and Disk Controller Card. These are analogous to Partitioned Business Components. And finally, the designer might use generic parts in the design of the Disk Controller Card, such as Memory Chips for cache, Bus Adapters, etc. These are analogous to Engineering Components.

Establishing one definition to satisfy all of these perspectives is certainly not required to be successful with components. What's more important is to recognize the different perspectives and to understand when it's appropriate to talk about a particular type of component. Hence, multiple definitions, one for each type of component:

Business Components represent real-world concepts in the business domain. They encapsulate everything about

those concepts including name, purpose, knowledge, behavior, and all other intelligence. Examples include: Customer, Product, Order, Inventory, Pricing, Credit Check, Billing, and Fraud Analysis. One might think of a

Business Component as a depiction or portrait of a particular business concept, and as a whole, the Business Component Model is a depiction or portrait of the entire business. It's also important to note that although this begins the process of defining the application architecture for a set of desired business capabilities, the applicability of the Business Component Model extends beyond application building.

Whereas Business Components model real-world concepts in the business domain, Partitioned Business Components implement those concepts in a particular environment. They are the physical building blocks used in the assembly of applications. As independent pieces of software, they encapsulate business data and operations, and they fulfill distinct business services through well-defined interfaces. Business Components are transformed into Partitioned Business Components based on the realities of the technical environment: distribution requirements, legacy integration, performance constraints, existing components, and more. For example, a project team might design an Order Business Component to represent customer demand for one or more products, but when it's time to implement this concept in a particular client/server environment, it may be necessary to partition the Order Business Component into the Order Entry component on the client and the Order Management component on the server. These are Partitioned Business Components.

Engineering Components are independent pieces of software that provide functionality that is generally useful across a range of applications. They come in all shapes and sizes, and they are typically packaged as black box capabilities with well-defined interfaces. They are the physical building blocks used in the assembly of Partitioned Business Components. Examples include: a workflow engine, a JavaBean that encapsulates a reusable concept like address or monetary unit, a complex widget that allows users to edit a list of order lines, a group of objects responsible for persistence, a JavaBean that sorts a collection of objects, and a simple list box coded as an ActiveX control.

Components are useful throughout the development process. As a design artifact, early in the process, Business Components provide an underlying logical framework for ensuring flexibility, adaptability, maintainability, and reusability. They serve to break down large, complex problems into smaller, coherent elements. They also model the business in terms of the real-world concepts that make up the domain (e.g., entities, business processes, roles, etc.). Thus they provide the application with conceptual integrity. That is, the logical Business Components serve as the direct link between the real-world business domain and the physical application. An important goal is to build an application that is closely aligned with the business domain. Later in the process, Partitioned Business Components and Engineering Components provide a means for implementing, packaging, and deploying the application. They also open the door to improved integration, interoperability, and scalability.

FIG. **36** shows a relationship between business components **3600** and partitioned business components **3602**. Business Components are an integral part of the previously discussed Framework Designs. Business Components represent real-world concepts in the business domain. They encapsulate everything about those concepts including name, purpose, knowledge, behavior, and all other intelligence.

In the Business Architecture stage **3604**, a project team begins to define the application architecture for an organization's business capabilities using Business Components. Business Components model real-world concepts in the business domain (e.g., customers, products, orders, inventory, pricing, credit check, billing, and fraud analysis). This is not the same as data modeling because Business Components encapsulate both information and behavior. At this point in the process, an inventory of Business Components is sufficient, along with a definition, list of entities, and list of responsibilities for each Business Component.

In Capability Analysis **3606** and the first part of Capability Release Design **3608**, the project team designs Business Components in more detail, making sure they satisfy the application requirements. The team builds upon its previous work by providing a formal definition for each Business Component, including the services being offered. Another name for these services is "Business Component Interfaces." The team also models the interactions between Business Components.

Throughout the remainder of Capability Release Design and into Capability Release Build and Test **3610**, Business Components are transformed into Partitioned Business Components based on the realities of the technical environment. These constraints include distribution requirements, legacy integration, performance constraints, existing components, and more. Furthermore, to ensure the conceptual integrity of the Business Component model, a given Partitioned Business Component should descend from one and only one Business Component. In other words, it should never break the encapsulation already defined at the Business Component level. Also at this time, the project team designs the internal workings of each Partitioned Business Component. This could mean the Engineering Components that make up the Partitioned Business Component, the "wrapper" for a legacy or packaged system, and other code.

In Capability Release Build and Test, Partitioned Business Components are built and tested. The build process varies depending upon the technology chosen to build the internal workings of each Partitioned Business Component. Among the many tests that are performed during this stage, the component, assembly, and performance tests are impacted the most by this style of development. A component test addresses a Partitioned Business Component as a single unit by testing its interfaces and its internal workings, while an assembly test addresses the interactions between Partitioned Business Components by testing broader scenarios. The performance test is impacted primarily by the techniques one would use to resolve the various performance issues. For example, it's common to run multiple copies of a Partitioned Business Component across multiple servers to handle a greater transaction volume.

In Deployment **3612**, the Partitioned Business Components are packaged and deployed as part of the application into the production environment. The application parameters and the manner in which the Partitioned Business Components are distributed are tweaked based on how well the application performs.

Well designed Business Components are anthropomorphic. That is, they take on characteristics and abilities as if they were alive. This means that Business Components should reflect directly the characteristics and abilities (i.e., the information and behavior) of the business concepts they represent. Therefore, only by examining the various types of business concepts will one discover an acceptable way to classify Business Components.

Business concepts come in a wide variety. For example, a product represents something of value that is up for sale,

SAP - Exhibit 1105
SAP v. Valtrus et al. - IPR2025-00420
Page 189 of 281

while a credit check represents the work that needs to be done to determine if a customer's credit is good. The former is centered around an entity—the product—while the latter is centered around a process—credit check.

This line of thinking leads to two types of Business Components: entity-centric and process-centric. Unfortunately, what commonly results from this paradigm is an argument over whether or not a particular Business Component is entity-centric or process-centric. In reality, Business Components are always a blend of both information and behavior, although one or the other tends to carry more influence. An appropriate mental model is a spectrum of Business Components.

Business Components on the entity-centric side of the spectrum tend to represent significant entities in the business domain. Not only do they encapsulate information, but also the behaviors and rules that are associated with those entities. Examples include: Customer, Product, Order, and Inventory. A Customer Business Component would encapsulate everything an organization needs to know about its customers, including customer information (e.g., name, address, and telephone number), how to add new customers, a customer's buying habits (although this might belong in a Customer Account component), and rules for determining if a customer is preferred.

Business Components on the process-centric side of the spectrum tend to represent significant business processes or some other kind of work that needs to be done. Not only do they encapsulate behaviors and rules, but also the information that is associated with those processes. Examples include: Pricing, Credit Check, Billing, and Fraud Analysis. A Pricing Business Component would encapsulate everything an organization needs to know about how to calculate the price of a product, including the product's base price (although this might belong in a Product component), discounts and rules for when they apply, and the calculation itself.

One might argue that the Pricing component is more entity-centric than process-centric. After all, it's centered around the concept of price, which is an entity. In reality, though, it depends on the business requirements, but again, whether or not a given Business Component is entity-centric or process-centric is not important yet. What is important is how well the Business Component represents its corresponding real-world business concept. The fact that most business concepts are a blend of information and behavior means that most Business Components should also be a blend of information and behavior. Otherwise applications would be much like they are today with a distinct separation of data and process.

Another way to think about the process-centric side of the spectrum is by asking, "What role performs the process?" For example, it's the picker-packer who picks inventory and packs it into a shipment. This might lead to the Picker-packer component. Another example is a Shopping Agent component that knows someone's buying preferences, shops for the best deals, and either reports back to the user or makes the purchase.

A pattern emerges when one examines the way these Business Components interact with each other. Process-centric Business Components are "in control," while entity-centric Business Components do what they're told. To be more explicit, a process-centric Business Component controls the flow of a business process by requesting services in a specific sequence according to specific business rules (i.e., conditional statements). The services being requested are generally offered by entity-centric Business Components, but not always. Sometimes process-centric Business Components trigger other process-centric Business Components.

FIG. **37** shows how a Billing Business Component **3700** may create an invoice. The control logic **3702** (i.e., the sequence of steps and business rules) associated with the billing process is encapsulated within the Billing component itself. The Billing component requests services from several entity-centric Business Components, but it also triggers Fraud Analysis **3704**, a process-centric Business Component, if a specific business rule is satisfied. Note also that "Step 6 " is performed within the Billing component itself. Perhaps this is where the invoice is created, reflecting the design team's decision to encapsulate the invoice within the Billing component. This is one valid approach. Another is to model a separate entity-centric Invoice component that encapsulates the concept of invoice. This would effectively decouple the invoice from the billing process which might be a good thing depending on the requirements.

It would be logical to conclude that the two types of Business Components translate to two types of Partitioned Business Components, but a small adjustment is required. Entity-centric Business Components translate directly to Business Entity Components, but a closer look at the ways in which a business process can be implemented in an application reveals two possibilities for process-centric Business Components. A business process can be: 1) automated, like a billing process, or 2) controlled by a user; like an order entry process. The former results in a Business Process Component, while the latter results in a User Interface Component.

FIG. **38** illustrates the relationship between the spectrum of Business Components **3800** and the types of Partitioned Business Components **3802**. Business Entity Components **3804** and Business Process Components **3806** are straightforward. The former is the physical implementation of an entity-centric Business Component (e.g., Customer), while the latter is the physical implementation of an automated process-centric Business Component (e.g., Billing). User Interface Components **3808**, on the other hand, require further explanation.

As mentioned above, a User Interface Component is the implementation of a business process that is user controlled, but more explicitly it is a set of functionally related windows that supports the process(es) performed by one type of user. Examples include: Customer Service Desktop, Shipping Desktop, and claim Desktop. These are not to be confused with low-level user interface controls (e.g., Active X controls), rather User Interface Components are usually built from low-level user interface controls. The reason for the dashed arrow in the diagram above is a subtle one. It points to the fact that earlier in the development process User Interface Components are generally not modeled as process-centric Business Components. Instead, they typically originate from the workflow, dialog flow, and/or user interface designs. See FIG. **39**, which illustrates the flow of workflow, dialog flow, and/or user interface designs **3902**, **3904**, **3906** to a User Interface Component **3908**. This makes complete sense given their direct tie to user controlled business processes.

FIG. **40** is a diagram of the Eagle Application Model which illustrates how the different types of Partitioned Business Components might interact with each other. Business Entity Components **4002** and Business Process Components **4004** typically reside on a server, while User Interface Components **4006** typically reside on a client.

FIG. **41** illustrates what makes up a Partitioned Business Component **4100**. As long as a component does what it's

SAP - Exhibit 1105
SAP v. Valtrus et al. - IPR2025-00420
Page 190 of 281

suppose to do, it doesn't matter what kind of code is used to build the component's internal workings. It could be anything from COBOL to Java. This is a key benefit of encapsulation. Classifying this code is a different matter. Some code **4102** is specific to the Partitioned Business Component. Other code is more widely reusable, both functionally and technically; this is where one finds Engineering Components **4104**. Another possibility is to "wrap" existing code **4106** from legacy and-packaged systems. Finally, it's important to note that patterns and frameworks are frequently used as starting points for designing and building this code.

Engineering Components are physical building blocks used in the assembly of Partitioned Business Components. They are independent pieces of software that provide functionality that is generally useful across a range of applications, and they are usually packaged as black box capabilities with well-defined interfaces. Engineering Components can be bought or built, and they come in a wide variety. Examples include: a workflow engine, a JavaBean that encapsulates a reusable concept like address or monetary value, a complex user interface control that allows users to edit a list of order lines, a group of objects responsible for persistence, a JavaBean that sorts a collection of objects, and a list box coded as an ActiveX control.

A pattern is "an idea that has been useful in one practical context and will probably be useful in others." Think of them as blueprints, or designs for proven solutions to known problems. Having found the right pattern for a given problem, a developer must then apply it. Examples of patterns include: an analysis pattern for hierarchical relationships between organizations and/or people, a design pattern for maintaining an audit trail, a design pattern for applying different levels of security to different user types, and a design pattern for composite relationships between objects.

A framework is a template for the implementation of a particular function (similar to a shell program). It usually embodies a known pattern (or group of patterns) in a specific technical environment. Frameworks are available from a number of third-party vendors, and they are also developed on projects. Developers are typically expected to customize and extend frameworks to meet their specific requirements, but this involves a tradeoff. Customizing and extending a framework may optimize its use, but the resulting framework tends to be less abstract, and therefore less reusable in other contexts. Examples of frameworks include: a framework for displaying an object and its properties in Smalltalk, a Java-specific framework for persisting data, and a messaging and publish/subscribe framework for DCOM.

FIG. **42** illustrates the role of patterns and frameworks. More specifically, it introduces the Eagle Architecture Specification **4200** and the Component Solutions Handbook **4202**, both of which are groups of patterns. Eagle also offers technology-specific starter kits **4204**, which include frameworks for various environments.

The pace of change in today's business world is increasing faster than ever before. Meanwhile, advances in information technology have enabled businesses to better understand their customers, provide greater value, and create new markets. However, as technology becomes more complex, applications have become more difficult and time-consuming to build and maintain. Looking forward, applications must be dramatically more responsive to change. They must be more:

	In theory . . .	In practice . . .
Flexible	Making it possible to quickly satisfy new business requirements by replacing or modifying certain components with minimal impact to others.	Making it possible to accommodate a new product line solely by updating the Product component.
Adaptable	Making it easy to deliver an application to a variety of user types through a variety of delivery channels with minimal impact to the core application.	Making it easy to provide in-home access to customer account information by developing only a new user interface while reusing existing components.
Maintainable	Making it easy to update an application by reducing the area of impact for most changes.	Making it easy to add a new customer attribute by isolating the change to one component-the Customer component.
Reusable	Making it possible to quickly assemble unique and dynamic solutions from existing components.	Making it possible to assemble an application at a fraction of the cost because eight of the twelve components that are needed already exist.
Integration Ready	Making it possible to reuse the functionality within existing systems by wrapping them as components within new applications.	Making it possible to absorb newly acquired divisions by "wrapping" their systems and "plugging" them into the enterprise infrastructure.
Interoperable	Making it possible to request services across platforms.	Making it possible to integrate two applications built on different platforms.
Scalable	Making is easy to distribute and reconfigure components to satisfy various transaction volumes.	Making it easy to accommodate the holiday crunch by running multiple copies of the Order component across multiple servers.

Components will help an IT organization achieve these quality attributes. Through encapsulation they make it possible to develop applications that are more responsive to change. One can make this claim with confidence because a component that is well encapsulated (i.e., an independent, black box component with predictable, well defined interfaces) can be used in any situation, as long as it's used for its intended purpose. It knows how to perform its services without regard to what's happening outside of its boundaries (e.g., the actions that precede or follow it).

Another key to embracing change is the predictability and conceptual integrity of the parts that make up an application. Fred Brooks, author of The Mythical Man-Month, writes, ". . . conceptual integrity is the most important consideration in system design." Therefore, components must be conceptually whole, and they must perform functions that are aligned with their purpose and within their sphere of knowledge. If they accurately reflect the real world, they are much easier to develop and maintain. If the real world changes, so must the corresponding component. Given a design with these characteristics, the opportunity for reuse is significantly enhanced, and the time it takes to upgrade the system is dramatically reduced. The Gartner Group agrees that component-based development will be a dominant method of application development in the years to come. They say that "by 2001, at least 60 percent of all new applications development will be based on assemblies of componentware, increasing speed to market and the ability to cope with change (0.7 probability)."

SAP - Exhibit 1105
SAP v. Valtrus et al. - IPR2025-00420
Page 191 of 281

Business Components and Partitioned Business Components represent a major improvement in design capability—some might argue the first major change in design thinking since structured design. There are several reasons for this breakthrough:

Business Components model entities and processes at the enterprise level, and they evolve into Partitioned Business Components that are integrated into applications that operate over a network. Consequently, they serve as an excellent first step in the development of scalable, distributed enterprise applications that map closely to the business enterprise itself (i.e., the way it operates and the information that defines it).

Business Components model the business, and thus they enable applications to more completely satisfy the business needs. They also provide a business-oriented view of the domain and consequently a good way to scope the solution space. This results in a good context for making process and application decisions. Finally, Business Components provide a common vocabulary for the project team. They educate the team in what's important to the business.

When modeled correctly, entity-centric Business Components represent the most stable elements of the, business, while process-centric Business Components represent the most volatile. Encapsulating and separating these elements contributes to the application's overall maintainability.

To manage the complexity of a large problem, it must be divided into smaller, coherent parts. Partitioned Business Components provide an excellent way to divide and conquer in a way that ties the application to the business domain. They provide the ability to "package software capabilities into more manageable (and useful) chunks." By contrast, traditional modules are too cumbersome to be reusable in multiple contexts. On the other end of the spectrum, objects are too small to effectively divide and conquer; there are simply too many of them.

Partitioned Business Components provide a greater emphasis on application layering—a well known, but often neglected concept in application development.

Partitioned Business Components are application building blocks. As an application modeling tool, they depict how various elements of an application fit together. As an application building tool, they provide a means for systems delivery.

Proven processes, patterns, and frameworks offer a higher level of reuse. This is one of the key advantages because it means greater agility. These mechanisms make it possible for hundreds of developers to do things consistently and to benefit from previously captured, reusable knowledge capital.

Business Components model the business. It sounds straightforward, but even with experience it's a challenge to identify the right components and to design them for flexibility and reuse. Flexibility and reuse are certainly more achievable with Business Components, but they are not inherent to Business Components. To accomplish these goals, as the previous examples suggest, one must understand what's happening within the enterprise and across the industry. One must work with business experts who understand the factors that will influence the current and future evolution of the business domain. This will improve one's ability to anticipate the range of possible change (i.e., to anticipate the future). The Business Component Model will be more flexible and reusable if it is challenged by scenarios that are likely to take place in the future.

Reuse becomes a reality more quickly if one plans for it. And it endures if one manages it over time. However, both of these things are difficult to do, especially for large projects and large enterprises. First of all, it's easy for communication across one or more projects to break down. It's also common for individual projects to pay more attention to their requirements and deadlines than to project-wide or enterprise-wide reuse. After all, their most important objective is to deliver value to their customers. Reuse must be engrained into the culture. This could mean teams responsible for project-wide and enterprise-wide reuse, but no matter how it's done, reuse must be one of the most important technology objectives.

Too much focus on low-level (i.e., code) reuse can be a trap. To draw an analogy, take a look at the recent history of the auto industry. Some auto makers were focused on inter-changeable parts and low-level standardization. For example, they decided to use the same body style for all of their cars. Unfortunately, when the industry began to move away from the boxy body style, they were not well prepared, nor were they agile enough to react in a timely fashion. They had invested too much in low-level standardization. Conversely, other auto makers were focused on quality processes and frameworks (i.e., high-level reuse). As a result, they were able to respond more quickly to the changing requirements. Engagement experience has shown that the same thing can happen with components and objects (e.g., too much emphasis on low-level inheritance). That's why it's important to focus appropriately on the high-level reuse enabled by processes, patterns, and frameworks.

Although Business Components and Partitioned Business Components represent a significant breakthrough in design capability, the architectural frameworks to support this breakthrough are still maturing. Standards come to mind first: Will it be COM, JavaBeans, or CORBA? It's still not clear. Likewise with languages: Will it be Visual Basic, Java? Tools and repositories offer another challenge. Clear winners have yet to emerge, and newcomers are constantly popping up with promising products. Finally, the legal and commercial market for buying and selling components is not mature. The market for high-level common business objects is just emerging, while the market for low-level components is still chaotic.

One of the most important challenges is teaching a new application development style. Although components and objects have been around for a while, they are new to most people. Furthermore, component-based development requires a change in the way one thinks about designing and building applications. Engagement experience has shown that it takes a couple of months to feel comfortable with this paradigm—and longer for those pursuing deeper technical skills. But this challenge is certainly not impossible to overcome. A combination of training and mentoring has proven to be the best way to teach these concepts, and the more rigorous approach that results from this education is well worth the journey.

The following tips and techniques provide an introduction to some of the issues surrounding the design of Business Components.

What is the Right Number of Business Components? How big Should They be?

The granularity of Business Components is a frequent topic of discussion. A fairly common misconception is that Business Components are the same as applications, but in fact, applications are assembled from Business Components (or Partitioned Business Components to be more accurate). A typical application might have ten to twenty Business Components. On the other end of the spectrum, Business Components are larger than business objects. In fact, some people refer to Business Components as large-grained business objects.

So What is the Right Size for a Business Component?

Business Components should encapsulate concepts that are significant to the business domain. Of course, this is subjective, and it certainly varies by business domain. In fact, business domain experts, with help from component modelers, are in the best position to make this judgment.

Bigger Business Components hide more complexity, which in general is a good thing. However, too much complexity in a component can lead to many of the problems that preceded component-based development. For example, embedding too much policy information can lead to a Business Component that is more difficult to maintain and customize. Another advantage is the fact that the coupling between bigger components tends to be weaker. On the other hand, bigger components are generally less cohesive and consequently less flexible. For example, assume that the concepts of warehouse and inventory have been combined into one Business Component. This could be problematic if a future application needs warehouse information, but not inventory information.

Smaller Business Component tends to be more flexible. It's also easier to reuse them in future applications. Unfortunately, smaller components typically result in a higher degree of coupling. One will find significantly more interactions between smaller components. This could also lead to performance problems. If two or three small components send each other a lot of messages, it might make sense to combine them into one. Smaller components may also be more difficult to manage, simply because more of them exist.

It's important to strike a balance, and keep in mind that the ideal size depends on the domain. If there's a question in one's mind, it makes sense to lean toward smaller components. It's easier to combine them than to break them up.

What's the Best way to Identify Business Components?

During the Business Architecture stage, the project team defines its business capabilities. At this point in the process, one can begin to search the business domain for Business Components. Then again later, during Capability Release Design, when the project team documents scenarios and workflows, one can perform a second iteration through the identification process.

The following steps describe one technique for identifying Business Components. FIG. **43** illustrates this Business Component Identifying Methodology **4300** including both Planning and Delivering stages **4302**, **4304**:

1. Start with entity-centric Business Components. For example, the customer is a significant entity in most business domains, therefore a Customer component may be included. A Customer Business Component would encapsulate everything an organization needs to know about its customers, including customer information (e.g., name, address, and telephone number), how to add new customers, a customer's buying habits (although this might belong in a Customer Account component), and rules for determining if a customer is preferred. Entities themselves can be physical or conceptual. For example, customers and products are physical—you can touch them. Orders, on the other hand, are conceptual. An order represents a specific customer's demand for a product. You cannot touch that demand.
2. Look for process-centric Business Components next. Generally speaking, a process-centric Business Component controls the flow of a business process. For example, in the utility industry, a Billing component would process customer, product, pricing, and usage information into a bill. Sometimes one will find an entity associated with the process—in this case, a bill or invoice—but another option is to model this entity as a separate, entity-centric Business Component, thus decoupling it from the process.

What's the Best way to Identify the Responsibilities of a Business Component?

Review the business capabilities, business processes, business practices, scenarios, workflows, and other requirements. Look for behaviors that will be supported by the application. In other words, what are the business functions that will be performed by the system? Assign them as responsibilities to the most appropriate component. If components were people and computers didn't exist, one might ask, "Who is responsible for this task?" In fact, sometimes it's helpful to assign component owners who speak up when they encounter a responsibility that should belong to their components—"Hey, I should be responsible for that!"

This section addresses several frequently asked questions that more broadly apply to the physical implementation of component- and object-based solutions. The answers are intended to increase the awareness of the reader. Most of them only scratch the surface of issues that are somewhat controversial within the component and object community.

What is the Role of Components in net-centric Computing?

Physical components play a critical role in net-centric computing because they can be distributed, as encapsulated units of executable software, throughout a heterogeneous environment such as the Internet. They have the ability to make the Web more than a toy for retrieving and downloading information. Robert Orfali, Dan Harkey, and Jeri Edwards, well-known experts in the field of component- and object-based development, wrote the following about distributed objects (same as "distributed components" for the purpose of this discussion):

The next-generation Web—in its Internet, intranet, and extranet incarnations—must be able to deal with the complex requirements of multi-step business-to-business and consumer-to-business transactions. To do this, the Web must evolve into a full-blown client/server medium that can run your line-of-business applications (i.e., a delivery vehicle for business transaction processing) . . . To move to the next step, the Web needs distributed objects.

What's the Difference Between Components and Objects?

From a logical perspective, components and objects are the same. They both model concepts from a particular domain, and they both encapsulate information and behavior. On this level, good component models and good object models share the same characteristics: high cohesion, low coupling, reusability, well defined services, and more. One might argue that granularity is a key difference. After all, for an object-oriented design, components are made up of objects. This may be true, but in reality both of them come in all sizes, thus making this difference rather insignificant.

From a physical perspective, components and objects are similar, but different. The key difference relates to the different ways in which they are implemented. As long as a component's interfaces comply with an accepted standard like COM, JavaBeans, or CORBA, its internal workings can be implemented using any technology (e.g., Java, Visual Basic, Smalltalk, C, or even COBOL). The internal workings of an object, on the other hand, can only be implemented using object technology. For the same reason (i.e., standard interfaces), it is possible to request a component's

services from any platform. That's not true of objects, unless they are wrapped with interfaces that comply with the accepted standards, which would make them distributed objects (i.e., components) instead.

Robert Orfali, Dan Harkey, and Jeri Edwards also wrote the book *The Essential Distributed Objects Survival Guide* (1996). Chapter 2, "From Distributed Objects to Smart Component," is an excellent source of information about objects, components, and the differences between them. They say the following about physical components:

A component is an object that's not bound to a particular program, computer language, or implementation . . . They are the optimal building blocks for creating the next generation of distributed systems . . . Components are standalone objects that can plug-and-play across networks, applications, languages, tools, and operating systems. Distributed objects are, by definition, components . . . Unlike traditional objects, components can interoperate across languages, tools, operating systems, and networks. But components are also object-like in the sense that they support encapsulation, inheritance, and polymorphism.

What is a Component Model?

This is a common point of confusion. From a logical perspective, the term "component model" is frequently used to refer to a Business Component Model in the same way that "object model" is used to refer to a business object model.

From a physical perspective, a component model (or a component object model) defines a set of conventions that provides a standard way to develop and use physical components, including how to define properties, events, behaviors, etc. It also includes the standard structure of a component's interfaces, the mechanism by which a component interacts with other components, patterns for asking a component about its features, a means for browsing active components, and more. Some of the existing component models are COM, JavaBeans, and CORBA.

Example: A Grocery Store

A grocery store chain is creating an enterprise-wide Business Component model. Currently the individual stores do not record specific customer information.

Consequently, a model based on today's requirements would not retain customer information.

However, they are looking into preferred customer cards. Furthermore, while analyzing the industry, the project team reads about a competitor with a pharmacy and video rental service. In both cases, customer information becomes critical. So the project team creates scenarios describing how they would use customer information to support these requirements. They create one Business Component Model that supports both today's and tomorrow's view of the customer.

In the near future, when the chain adopts preferred customer cards, and in the more distant future, if they decide to add a pharmacy or video rental service, the Business Component design for their current application will provide a solid foundation for the future requirement of tracking customer information. If they weren't using Business Components, they would not have a model that maps to their business domain, and introducing new requirements would require more abrupt changes.

Example: Inventory Management

A telecommunications company in the paging business sells and leases pagers and services. One part of the company is installing an inventory management system for tracking pagers, while another part of the company is trying to determine how to track the frequencies that are owned and leased by the company. What does this company mean by inventory? Does it simply mean knowing what items are in a warehouse?

When the company thinks abstractly about the concept of inventory, they discover that it's all about managing anything of value. When they look at what they have in inventory, they discover that it is countable, reservable, and has a cost associated with it. Inventory doe's not require specific knowledge of the use of an item in inventory; that knowledge can be put into another component, such as Item. If inventory does not need to know the specifics about its use, then it could apply its ability to count, reserve, and value anything it is associated with. Inventory could be used to manage a variety of things: conference rooms, fixed assets, work in process, finished goods, and leased frequencies.

So one can start out building an inventory management application and then build the ready-to-reuse Inventory component which, without modification, can support many other uses. In this way one can unload the concept of inventory so that it can be reused outside the context it was initially planned for.

This section highlights key messages for project management. The Management Lessons discuss these points further. Manage Expectations-component Technology is not a Silver Bullet

Components promise to enhance the ability to quickly build robust systems through the use of reusable pre-built software components. Properly leveraged, components can provide the foundation upon which one meet and exceed the demands of a global marketplace which increasingly uses technology as a primary competitive advantage. Like object technology before, components are often portrayed as the magic silver bullet to slay the ills of software technology.

Yet, the silver bullet mentality inevitably leads to unreasonable expectations. Intense media attention fuels these expectations. For example, components are often compared to Lego blocks that are simply plugged together to form complex systems. Experience has shown, however, that component technology is not that simple and that payoffs are primarily in the long term. There are several factors impede short-term payoffs.

Most important, demand exceeds supply for professionals with component and object-oriented skills. Thus, many initial projects incur start-up costs related to recruiting, training, and learning curve. Furthermore, after receiving investment in training, individuals find themselves in demand, becoming higher risk to leave the organization.

Another unreasonable expectation is the belief that components may provide immediate software reuse. Experience has shown that reuse is not automatically attained; it is necessary to establish a disciplined approach to reuse and create a development culture that embraces reuse.

A client's view of component technology may vary depending on their previous experiences. Client's with no component or object experiences may have the most unrealistic expectations for what the technology can delivery. In contrast, clients that have attempted object-oriented applications and failed may understand that components are not the "silver bullet" that many have promised. In fact, these clients may require additional evidence of the viability of a component approach. For these clients, a component approach can be very appealing since a component-based architecture can combine both traditional and object technologies. And lastly, there is the third category of clients that have achieved some measure of success with object technology and view component technology as the natural evolution towards the goals that are only partially delivered by object technology alone.

Component-based Development's Focus on the Long-term is Usually a Good Tradeoff

Component-based development is also inherently biased towards the long-term. For example, the development process strives for a higher degree of quality and reuse, incorporating iteration between design and code to support refinement. Striving for this higher design quality may almost always, by definition, cost more up front. Despite these initial costs, component-based development's focus on the long-term makes economic sense. Experience has shown that 60–80% of development costs are in maintenance.

Recruit a Project Champion or Sponsor With a Long-term Focus

To ensure that short-term concerns do not outweigh the potential benefits, project management should maintain a realistic view of the benefits and risks of components. Thus, recruiting a project champion or sponsor with a balanced, long-term view is a key to success.

Business Benefits Must Support Adoption of Component Technology

Establish Clear Goals for a Component-based Project

Component technologists sometimes promote component development for its own sake, without regard: for the business benefits. However, rarely may management justify something they do not understand. Component technology introduces a daunting array of new terminology. Furthermore, if a pilot component project is launched with unclear goals or mission, the significant short-term costs and challenges may inevitably undermine the commitment to components.

Thus, component technology must be justified in business rather than technology terms. In many cases, a traditional client/server solution can deliver the benefits. This proves especially true for short-lived, simple, or moderately complex applications. On the other hand, component technology may benefit applications with characteristics such as:

a long maintenance life

complex processing or significant asynchronous logic

complex data relationships

very dynamic business requirements

multiple access channels

legacy evolution or replacement

functionality common across multiple applications

Firm Clients Have Achieved Business Benefits

The number of engagements that have employed component and object technologies has continued to grow over the last few years. These engagements have shown that object and component-based approaches can lead to significant business benefits.

Reduces Maintenance Costs

Properly designed component-based systems should reduce maintenance costs. Encapsulating implementation details and data make a system more resilient to changes in the business or underlying technology. Furthermore, design decisions must rigorously consider what is likely to change. Susceptible points should be hidden behind an abstract, public interface that decouples their potential changes from impacting other components.

Component Reuse Reduces Development Time

Components are more easily reused because they provide well-defined interfaces and can often be used through visual development tools. This make it more straightforward to develop components for one project and share them across other projects. Furthermore, components can be designed so that their properties can be tailored to meet varying requirements. Once a reusable base of components has been established, the development time for subsequent projects can be reduced.

In one utility company they saw significant gains in the reuse of components across initiatives. Rather than copying and tailoring source code for new initiatives they were able to assemble applications from already created components.

Another engagement estimated that new system development was reduced 25% once the first application was released and a core set of components was established. Even though the engagement ultimately realized the benefits of reuse, the client still had the expectation that reusable components would save time and money for the first project. To manage this expectation, the project team needed to re-emphasize that component-based development requires an initial investment.

Leverage Existing Technology Investments

Many clients have existing technology assets that would require significant investments to replace. Components can enable these legacy systems to be wrapped with component interfaces so that new applications can easily interact with them. Later, these legacy applications could be replaced without changes to the new applications.

Shields complexity and Supports Re-engineered Processes

Objects Raise the Level of Abstraction in the Software Solution

Object development enables closer integration between developing applications and reengineering business processes. The first object-oriented language, Simula, was invented to enable simulation. It and other object development environments provide capabilities that raise the level of abstraction of the software. That is, object-oriented languages and design techniques enable writing software in terms closer to the real-world business rather than the computer.

Enables Improved Usability

Object-oriented technology can support improved usability in two ways. First, objects messaging each other lends itself to simplified programming of advanced, direct manipulation or multi-media interfaces. Second, an object metaphor for designing the user interface may be a more desirable interaction style for some types of users such as knowledge workers needing flexible navigation.

Reduces System Test Complexity and Cost

In a few different instances, the object-oriented development approach has significantly reduced system test complexity. In all these cases the projects fell behind schedule due to learning curve, the complexity of custom architecture development, and increased effort for component and assembly testing. However, once core, reusable objects in the domain model and application framework stabilized, system testing the functionality and performance was much easier. For example, since less code and data knowledge was replicated throughout the system, global changes could often be made by making a change in one place.

Component Technology may Help Improve Communications With Users

The close tie that component and object modeling enables between the software solution and business process may help software analysts and users or business analysts to better understand each other, reducing errors in communications. This represents a significant opportunity, because misunderstanding user requirements has been proven to be the most costly type of mistake in systems development. A component model further improves the understanding of the software design by providing a larger-grained model that is easier to digest.

Lastly, communication with users is often improved by using scenarios which convey requirements through familiar business situations.

Multiple Access Channels

Component architectures are inherently service-oriented. Components provide their services through interfaces which consist of operations. Because components are independent pieces of software they can be reused by any number of applications. Thus, component-based architectures are well suited to environments that need to provide multiple application "personalities" or access channels. New personalities can be provided by creating a new user interface layer that reuses the existing business components.

Managing Risk is Key

Component technology is still high risk, because it may often:

have a pervasive impact on the overall development approach

require immature technology or tools

implicitly involve complex functional requirements

Component-based Development is not Only new Technology; it is a new Approach to Software Engineering

Component-based development should not be understood as just a technology decision; rather, it is a new approach for software engineering. Thus, it affects almost all aspects of development including methodology, tools, organization, and architecture approaches. This broad impact creates multiple learning curves, complicating the migration of an organization. Finding available skills is also difficult, because demand currently outweighs supply.

Component-based systems may also require immature technology or tools. Many of the core development tools such as the programming language and environments for C++, Visual Basic, Java and Smalltalk are actually very robust. However, some of the ancillary tools such as the CASE tools and web development tools or technology architecture components such as messaging middleware may not be as mature. Thus, the team may face a choice of managing some risk exposure with a tool or library that simplifies development, or avoiding this tool risk but facing a more complex development challenge.

Another, more subtle source of risk is the inherent functional complexity of applications often chosen for component-based projects. Component technology's technical characteristics enable dynamic, functionally complex systems. For example, business reengineering can capitalize on the inherent flexibility of component-based systems. However, reengineering creates more dynamic functional requirements, thereby increasing risk. Not to mention that business reengineering is itself a risky venture.

Thus, proactive risk management is an essential practice in development. Traditional risk management techniques apply to component-based projects. For example, a "top ten" risk list can help focus management attention. This risk focus must then influence the development tasks carried out by the team early in the project to ensure risks are addressed in a timely fashion.

Architecture is Essential to Delivering the Benefits

Component Technology Enables Application Frameworks

Component-based systems extend the notion of architecture beyond that in a traditional system. Much of the power of component-based systems is the ability to leverage application frameworks. Frameworks are somewhat analogous to program shells found in a traditional environment such as the INSTALL1 online system with components like MES and CCP. However, this is only an approximate analogy. An application framework goes beyond traditional application architectures to provide a greater degree of default behavior and flow of control in a skeleton of the application.

For example, traditional program shells rely heavily on cut-and-paste techniques to achieve reuse. This places a heavier burden on the developer and exposes the structure of the application. With an application framework, object-oriented capabilities minimize or eliminate the need for cut-and-paste reuse. A well-designed framework reduces the burden on application developers by providing an architecture environment that effectively says, "Don't call us, we'll call you."

There are many frameworks within the Java programming environment. For example, Java Security, a very important topic in new netcentric architectures, provides a Java Security Framework. This is a plug and play framework that allows developers the option of plugging in a security provider of their choice (DES, RSA, etc) or developing a custom security solution that can be called by security clients. To create a new security provider, the developer must only implement the required interfaces for the framework and provide a well-known name. Once these requirements are met, the component can be plugged into the framework.

Component-based Systems are Distinguished by a Business Component Model

The Presence of a Reusable Business Component Model is a key Ccharacteristic

A component-based software architecture may have a domain component model shared by the application processes. The component model contains the core business components that represent the business directly in software. These components perform behaviors upon request by windows, reports, or batch process control objects.

The presence of a component model distinguishes component-based systems from procedural, client/server systems. In a procedural approach, there is no shared business component model. This typically requires, for example, programs to pass data to each other in a context record. Thus, any changes to the data may affect many programs. The extent of business logic reuse is also usually less with the procedural approach.

The presence of a business component model also distinguishes a component-based is architecture from that produced by componentware tools. Specifically, many traditional and even component-based tools provide data-aware controls that tie the user interface directly, to the database. This is indeed a powerful technique to rapidly build simpler, less strategic applications. However, it suffers from a lack of smaller-grained business reuse and increased coupling between presentation and data. This may increase maintenance costs and miss opportunities to flexibly model complex business processes, as can be done with a component model. On the other hand, producing a reusable component model requires a higher level of abstraction and is therefore a more difficult approach.

Component Systems are Based on Standards

Component-based systems are also usually distinguished by their use of one or more of the leading component standards, i.e. CORBA, DCOM, or JavaBeans. These standards define the mechanisms that business components may use to communicate with each other. However, a system does not necessarily have to use one of these technologies to be considered component-based. The most important criteria is that the application is made up of reusable, service-oriented building blocks that encapsulate their functionality.

Component-based Systems can Incorporate a Variety of Technologies

Clients can Select the Most Appropriate mix of Technologies

Just as none of a user's client experience with objects has involved migration to a completely pure object solution,

SAP - Exhibit 1105
SAP v. Valtrus et al. - IPR2025-00420
Page 196 of 281

components may involve a variety of technologies. This is even more true for component-based systems since they provide the ability to integrate different technologies through well-defined interfaces. The ease of integration is very appealing to clients since it allows them to maintain their existing technology investments, leverage their existing skills, and select a mix of technologies that best fit their tolerance for risk.

More Diverse Skills may be Required

Because components can be implemented in a variety of programming languages on a number of platforms, it is often necessary to have competencies in a number of technologies. For example, one client used Visual Basic, Smalltalk, C++, and COBOL for different layers of the system. The increasing number of technology combinations also increases the complexity associated with development activities such as testing, debugging, and configuration management.

Component can Wrap Procedural Applications

Wrapping is a technique to integrate traditional system components. It applies to both the application and system levels. For example, a component can provide a public interface, encapsulating a legacy application.

Wrapping can be effectively applied to integrate a legacy billing system with a large, object-oriented customer care system.

At the architecture level, wrappers often provide database interface objects to shield the application from the database vendor.

Architecture Development Must Start Early

A Tension Exists Between Scenarios and Frameworks

As with client/server, architecture work must start early. As noted above, this is particularly challenging because of the level of application reuse in a well-designed application framework and domain component model. Because of this reuse, the framework must be heavily driven by application requirements, or scenarios. Yet, the architecture team must stay one step ahead of application development teams to ensure that the architecture and component model are ready in time to be reused. Thus, a difficult tension exists between scenarios and frameworks.

The tension between scenarios and frameworks can be simplified to the extent that third-party or standard architectures such as Eagle can be leveraged. In any case, the following guidelines should be considered, particularly for custom architectures:

- The architecture should be defined and prototyped, if necessary, early in the preliminary design
- The architecture should be complete-at the very least, the development architecture and overall framework, prior to developers actually coding; the design must be in place earlier when functional developers start detailed design; private architecture aspects may be deferred
- Time must be planned for architecture support based upon unforeseen scenarios, performance tuning, documentation and developer mentoring
- Developing a custom application framework should be estimated as a set of tasks in addition to much of the traditional technology architecture development

New Roles and Organization Strategies Must be Introduced

Component Projects Require Modeling Skills

Most traditional engagements divide roles into two basic categories, functional and technical, or architecture. Component-based development introduces a third dimension by requiring an extensive modeling role. Early experience has shown that the capability to draw abstractions in modeling a business problem or application framework is a unique skill set distinct from purely technical or functional skills.

Managing the Domain Component Model Requires new Organization Approaches

In addition, the extensive reuse of a core business component model requires an organization structure that manages it as a shared resource. This creates a tension between the needs to support consistent reuse of core components, and the desire to solve a business problem front-to-back. Experience has shown this often requires some form of matrix organization, combining vertical-based leadership along the lines of business functions, and horizontal-based leadership along the lines of architecture layers.

Leveraging Expert Mentors and Time are key to Scaling the Learning Curve

The Learning Curve is Greater, Because it has Multiple Dimensions

Component-based development involves a longer learning curve than comparable software technologies, because it has multiple dimensions. Component technology skills cover a wide range of competencies—from modeling and design skills to detailed programming syntax. Yet, a user may have good success with people scaling the learning curve in a reasonable amount of time.

Programmers can expect to perform simple tasks in 2–4 weeks when an architecture is in place. More complete implementation skills may require 8–24 weeks. Design skills also typically require the same amount of learning curve, 2–4 weeks for simple tasks and 8–24 weeks or slightly more for complex design problems. Usually programming should precede design experience, if possible.

Thus, leveraging experienced component and object technology skills is key to success. Even a few skilled component developers can provide significant leverage to mentor and support an inexperienced development team. Experience has shown that at least 20% of the development team should have component technology or process skills at the outset. This represents a minimal level for large engagement teams with projects of one year or more duration. Smaller teams or shorter duration projects may typically require more. It is also extremely important to have a significant percentage of the team with client/server skills, to reduce additional learning curves such as GUI design or client/server architecture development.

Estimating and Planning Present new Management Challenges

Projects Should Allow Time for Start-up Costs and Contingencies

There is still not enough experience with component technology to support rigorous, detailed metrics. One reasonable checkpoint for estimating an initial project is to use traditional techniques, and then add time to adjust for contingency and start-up costs such as training, learning curve, and architecture development. Early client engagements have demonstrated that an initial project may almost always be more expensive due to these start-up costs.

Yet, care should be exercised in applying traditional estimating metrics. For example, traditional metrics often use number of days per window or report. Component-based development can result in significantly different window counts for similar functionality.

In addition, the fixed versus variable nature of costs should be considered. Start-up costs are often not simply a variable percentage of the project size, because roughly the same architecture components may be required independent of size. Thus, anecdotal evidence suggests that the start-up costs usually have a greater effect on a small project.

Development Requires a mix of Waterfall and Iteration

Systems development traditionally relies on a waterfall model. This approach manages development in sequential

SAP - Exhibit 1105
SAP v. Valtrus et al. - IPR2025-00420
Page 197 of 281

phases of activity such as analysis, design, code, and test. The waterfall provides control and discipline to development, particularly critical for large, mission-critical efforts.

On the other hand, iteration enables proving out design assumptions in code early in the process, and testing the validity of code before proceeding on a wide scale. The information and learning gained from iteration are especially important for component-based development, because it is so new. As component-based architecture and methodologies mature, the need to iterate may be reduced

Significant Planning and Status Monitoring is Necessary to Manage Iteration

However, managing iteration on a large scale is difficult. The team can easily slip into hacking, in which design is simply skipped before coding. Or, a team may use iteration as an excuse to not exercise due diligence in completing tasks. Thus, a merging of waterfall and iterative principles is beneficial. Yet, striking acompromise between waterfall and iteration is not easy. Thus, significant effort must be invested for detailed workplanning and status monitoring.

Incremental Development may Help Manage Scope and Risk

Incremental Development Partitions the System Roll-out Into Releases

Perhaps the most effective way to mitigate the risks of a large project is to simply avoid being large. Incremental development addresses risk by reducing the necessary team size and scope. "Incremental" and "iterative" development are often used interchangeably, but they are different approaches.

Incremental development partitions the system roll-out into successive releases. For example, the initial release of a customer system might comprise order processing, followed by a subsequent release for billing, and a third release for collections processing. Thus, incremental development adds new functionality, while iterative development continuously refines existing functionality.

Incremental development avoids the complexity of a big bang integration. Furthermore, although an incremental approach delivers less in each successive release, it can deliver higher priority portions of the system much earlier than a traditional approach, thereby recognizing business benefits in a shorter time frame.

Despite these benefits, incremental development is not a panacea. Many times a big bang conversion has proven necessary, if the cost and risks of having parallel systems and bridges, performing conversion, and rolling out training are high. These costs must balance those introduced by the delayed delivery of business benefits and the risks implied by increasing scope and team size. The urgency of the business and the desire to manage development size may sometimes favor an incremental approach.

Commercially Available Methodologies Have a Narrow Focus

Most component-based methodologies focus primarily on analysis and design techniques. For example, less guidance is available for configuration management or testing. Yet, both of these aspects are more complex with component-based development, because of the greater level of granularity of the software decomposition. Because the methodologies are generic, they also typically do not address detailed architecture or design steps.

Configuration Management and Testing are More Complex

As noted above, the increased granularity of a component-based system and the variety of technologies associated with it complicate testing and configuration management. A component-based system may often have more than ten times as many components as a traditional system. While component-based systems are more granular than purely object-oriented systems, configuration management is not necessary less complex. While the use of components allows objects to be packaged into more comprehensible interfaces, it also increases the number of elements that need to be managed. Typically, the following entities may be tracked:

Methods

Classes

Packages (which are often aligned with components)

Components

Configurations

Applications

Configuration management requires a comprehensive approach of tools, procedures, and organization approaches. Multiple levels of component ownership must be defined. The higher level of reuse requires frequent roll-outs of updated component versions. This also typically requires the workplan and other status monitoring techniques to track dependencies between components at a much lower level of detail.

In addition, completing a set of processing requires many software components working together. Thus, testing involves integrating many more components. The complexity is magnified, because the integration work often cuts across different developers. The testing strategy must generally include more testing phases, each specifying a lower level of detail. Furthermore, automated regression testing has proven essential to address the complexity of integration.

Address Performance Risks Early, but Defer Application Tuning

Timing when to address performance has subtle complexities for a component-based system. Certainly, component-based development involves new technologies that introduce performance risks. Prototyping architecture components should be initiated early to adequately address the performance risks.

On the other hand, excessive application tuning should not be done to the exclusion of following good design principles, especially if the components are built using object technology. Experience has shown that dramatic performance improvements can be made late in object-oriented development projects. Furthermore, following good design principles actually better enables these tuning capabilities.

However, if more traditional approaches are used to implement the components, then it may be more appropriate to tune performance throughout the development life cycle.

Third-Party Components Have Increasing Importance

Third party components can play an important role in software development. Today's development tools make it easy to incorporate off-the-shelf components and customize them to a project's specific requirements. Thus far, off-the-shelf components have primarily consisted of user interface or architecture components. One project bought third party components for the user interface, device drivers, bar-coding, and database drivers. This project found that it saved a significant amount of time, especially in areas that required specialized programming skills. Unlike architecture components, it is not likely that third-party business components may be available any time soon.

Staffing, Training and Skills Development

This chapter discusses management issues related to staffing, training, and skills development.

Component-based Systems Rrequire a mix of Technical Skills

Object Skills are Common, but not Required

Components and objects are frequently considered to be equivalent technologies; however, they are not one in the same. While object-oriented systems may be developed using object-oriented analysis, design, and programming, a component-based system can be developed using a wide variety of languages, including procedural ones. As a result, the required depth of skills for a component-based project may depend on the blend of technologies used. For example, one project may require skills in COBOL, C++, and Smalltalk, while another may use Visual Basic exclusively. Because many projects are building components with objects, deep object-oriented skills may continue to be an essential ingredient in the success of a project.

Competencies in Multiple Technologies may be Required

Since component technologies make it possible to integrate different platforms, languages, and other technologies, it is often necessary to develop a broad portfolio of skills on a project. It is important to develop an early understanding of the different skills required and how they can be developed and leveraged across a project.

Leveraging Experienced Component Practitioners is Key

Leveraging experienced component technology skills is key to success. Even a few skilled component developers can provide significant leverage to mentor and support an inexperienced development team.

At least 20% of the implementation team should have component skills

Small teams or short projects likely require more

Experience has shown that at least 20% of the development team should have object/component technology or process skills at the outset. These represent minimal levels for large engagement teams with projects of one year or more duration. Smaller teams or shorter duration engagements need a higher ratio of experienced component developers. Furthermore, custom building the architecture from scratch may generally demand even more and deeper skills, unless the team has exceptionally talented individuals, extensive client/server experience, and ample time to scale the learning curve.

It is important to note that component technology skills cover a wide range of competencies—from modeling and design skills to detailed programming syntax. Rarely may one individual have the necessary expertise in all these areas. Thus, experience has shown that it is necessary to find individuals that specialize in one of these areas to leverage across a large team. The key is obtaining the right balance of technology and methodology skills.

One engagement Used a 1:1:1 Rule to Leverage Expertise

One large engagement found the most effective leveraging ratio was 1:1:1, comprising an experienced object specialist, an experienced programmer without object skills, and an inexperienced person. Note that this ⅓ ratio rule only applied to the team doing implementation. Thus, even though the total team size was about 200, only 40–50 were doing hands-on implementation, implying the need for about 13–17 skilled people.

Another engagement found the best mix to be one experienced developer to every four or five new developers. This project had a well-defined architecture and used Visual Basic to develop components. The relatively short learning curve of Visual Basic allowed this project to further leverage its experienced developers.

Exercise Caution When Contracting External Component Specialists

In some cases, independent contractors have proven an effective solution for filling gaps with specific niche skills. Experience has shown, however, these people may not be business-oriented, adapt well to the structure of a large engagement, nor have experience with mission-critical development.

Another problem has been having to fight object religion wars.

Managers Must Adopt New Techniques, Yet Not Forget Fundamentals

It's often said that, a good manager can manage anything. Many management skills such as planning, monitoring status, working with end-customer expectations, and managing risk certainly apply to any domain. These blocking-and-tackling aspects of management must not be forgotten on a component-based development project. Managers may, at times, be intimidated by component experts, and ignore the basics of project management.

Managing Iteration is Difficult, but Possible

In particular, object industry and academic gurus frequently suggest that object development and iteration simply cannot be managed. Their recommended approach is usually some form of time-boxing the development, simply declaring victory whenever time is up. However, this represents a very unappealing approach to promising delivery of business benefits to clients. Fortunately, experience has shown that this does not have to be the case. Managing iteration, while certainly more difficult, is possible.

However, software development managers must recognize that component technology has a pervasive impact on many aspects of the development process including estimating, planning, methodology, and technology architecture. For example, iteration impacts many of the standard rules-of-thumb for work completion. And the extensive reuse of a common business component model requires more sophisticated organization strategies.

Managers Must Invest Time in Training

Thus, successful managers must be willing to invest the time to learn new terminology and techniques to adapt to these changes. Traits common to those who have successfully scaled the component management learning curve include:

Experience with client/server development and a technical orientation

Willingness and flexibility to learn new terminology, tools, and techniques

Strong communication and people skills.

Sound understanding of the system's development lifecycle and the risks at the various stages

Architecture Roles Require Diverse Skills

Complicating the search for architecture skills is the need to find developers who also possess the necessary communications and teamwork skills. The architecture team must be capable of both delivering an application framework, and giving people appropriate mentoring and support. Many technology architects are simply not well equipped to handle the tutoring, coaching, and communications demands inherent in component-based development.

Avoid starting inexperienced people in architecture roles. There are simply too many skills to learn. Architects need to have a deep knowledge of design patterns, programming languages, technical infrastructure, and methodologies. It is better to start new developers in application development roles where they may have the opportunity to view the architecture as a consumer. This perspective may make them more effective in future architecture roles.

SAP - Exhibit 1105
SAP v. Valtrus et al. - IPR2025-00420
Page 199 of 281

While the dual role of building and supporting an architecture exists in a traditional client/server system, it may be more pronounced with component technology. Component-based systems require a higher degree of coordination by the framework developers partly because more application developers may be inexperienced with the environment. However, even an experienced team requires extensive coordination, because a greater level of consistency is required.

Developing with component technology demands more consistency, because an application framework and business or domain component model provide more reuse. In particular, much of the business logic may be shared by a common domain component model, viewed by many windows. To strive for this greater level of reuse across many business functions requires coordination among many developers. The risk is that the components may not fit together.

This type of development approach requires a strong architecture vision that is clearly communicated and supported through training, mentoring, and documentation. If a strong vision does not exist, then the components may inevitably not fit together into a cohesive, integrated architecture. In addition, this strong vision must include an understanding of the business objectives and functions of the system to be effective.

Strong architecture direction must also be accompanied by a positive "bedside manner". Application window developers may often perceive a framework somewhat restrictive of their creativity, too limiting, or burdensome, particularly when bugs hold up their delivery. It's important for the frameworks developers to be service-oriented; and, to realize that developing a reusable component is hard work and requires iteration.

Do Not Organize All the Component Skills on the Architecture Team

Because of the significant technical challenges often faced, a team may be tempted to staff all the experienced component developers on an architecture frameworks team. This strategy makes some sense. However, it should not be followed to the exclusion of leveraging the application or component modeling development team. Developing the functional business logic requires component development and methodology skills, as well.

Staff an Engagement Team With a Mix of Backgrounds

Staffing an engagement with deep technical skills is clearly a challenge. However, the engagement team should not overlook the importance of functional skills. Experience has shown that technical backgrounds may sometimes be over-emphasized to the detriment of functional expertise.

It is important to remember that many roles on the team are more demanding functionally than technically. Interviewing users, analyzing business processes, and designing the user interface all do not require extensive technical training. Moreover, not adequately understanding and analyzing the functional requirements are the most expensive mistakes. Research has shown that 70–80% of a system's mistakes result from misunderstood requirements.

Component Technology Involves Multiple Learning Curves

A component approach affects almost all aspects of the development lifecycle. For this reason the component learning curve cannot be equated with a programming learning curve such as 'C'. There are multiple, distinct learning curves that affect individuals at many different levels in the organization:

- Component and object-oriented concepts and terminology
- Object analysis and design
- Programming language
- Programming environment and other development tools (e.g., browsers, debuggers, user interface tools)
- New architectures—such as how to use the project-specific application framework
- Management—such as estimating and planning for work, and managing iteration and prototyping

Educating management about the multiple learning curves helps manage expectations. It's also important to avoid equating experience with pure elapsed time. For example, a person may be in the implementation phase doing things unrelated to building their component skills such as creating test conditions.

Component Skills may Take Longer to Transition to the Client

As a result of the many learning curves, it can take longer to successfully transition skills to the client. It is essential to have client participation in all areas of the project to ensure the transfer of skills. One of the most effective approaches is to have client personnel pair up with more experienced developers. Of course, this may be more expensive and may required buy-in from management.

The Rate at Which Individuals Scale the Learning Curve Varies Widely

Experience has shown that individuals scale the learning curve at very different rates. A user may have good success with individuals becoming productive in a reasonable amount of time. In some cases, people have learned extremely fast; on the other hand, a few have had considerable difficulty.

A useful model of the expected learning curve is outlined by Goldberg & Rubin [3]. These results are based on their extensive experience training personnel, primarily in the Smalltalk environment. Three primary levels of proficiency include:

- Basic—capable of doing basic assignments with adequate supervision, usually attained after formal training and some experience with simple assignments
- Functional—capable of doing most assignments with a predictable level of productivity and minimal supervision
- Advanced—an expert resource capable of solving very difficult or unusual problems

They distinguish the, learning curve in four different skill areas as shown below, measured in months:

Category	Basic	Func	Adv
Analysis and Design	4 wks	6–8 mos.	18–24 mos
Implementation	3–4 wks	5–6 mos	18–24 mos
Frameworks Design	16 wks	12–24 mos	24–48 mos
Management	3–4 wks	12–18 mos	24–36 mos

The above results are reasonably consistent with a user's experience on client engagements. Some experience suggests that most firm personnel, on average, reach proficiency levels slightly faster than the above figures. However, a user may experience a much larger deviation, both positive and negative, than that reported above.

For example, some talented individuals reached a functionally competent level in implementation skills in as little as 8 or 10 weeks, less than half that suggested above. On the other hand, about 10–15% of individuals did not ever reach this level of expertise in a reasonable amount of time.

SAP - Exhibit 1105
SAP v. Valtrus et al. - IPR2025-00420
Page 200 of 281

Early Experience has Identified key Predictors of Success

As noted above, a user may experience a reasonable degree of success in training personnel on engagements. Unfortunately, some clients have not been as successful.

Key predictors of success can be drawn from this experience and others. It is important to recognize that the list below is drawn from a very small experience base. As one's experience grows, the list of traits may be refined with-hopefully-more objective measurability. This may be key to helping both a user and clients to be more successful with components.

Ability to Deal With Change

Component-based development requires a high degree of change. Firm personnel deal with change their entire career. Often, client personnel may not be as adaptive. They may have worked with the same structured methodology and COBOL for 5 or 10 years. To change their entire process can be a big culture shift. Individuals must have the right attitude and interpersonal flexibility to change. This factor may help explain why less experienced people have often scaled the learning curve faster than more seasoned developers.

Yet, the simple fact that someone has deep COBOL experience does not mean that they may fail. There have been several examples of people on engagements who successfully made the transition from COBOL to Smalltalk, including architecture roles. However, all of these individuals were highly motivated with an open mind to change.

On the other hand, migrating to C++ may be a considerable challenge for people who do not have experience with a pointer-based language. That is, C++ projects should favor staffing people who have minimally programmed in languages such as C or assembly language.

Quick Study

Component technology involves multiple learning curves-people may need to learn fast. They must be motivated self-starters, capable of learning quickly on their own, and willing to read and perform supplemental tasks to improve their competencies.

Communications Skills

Component-based projects are very social endeavors. Because any given business function requires several collaborating components, developers also have to collaborate with one another. To ensure that components integrate smoothly, and to achieve the desired reuse, a high degree of communications and teamwork is necessary. This is significantly different than many traditional systems where a system is decomposed into larger, monolithic modules. These modules are typically developed front-to-back by each developer in relative isolation.

Creativity—experience With Custom Systems Development

A component-based development project requires creativity. The overall atmosphere is usually very challenging with fewer, concrete rules. The answer to many analysis and design decisions is, "it depends". Similarly, the development environments encourage exploration and browsing.

Work Ethic

Individuals must be motivated to undertake personal training. There often is not enough time to support all the training needs during normal work hours for the system to meet a reasonable schedule. Thus, at times, individuals must pursue personal study and experimentation after hours. This type of commitment requires enthusiastic, hard-working individuals.

Initial Training Requires Hands-on Case Studies to be Effective

Initial training requires significant upfront investment. Project Eagle achieved very good results with their multi-week Eagle University. Unfortunately, this represents a larger amount of upfront time than many engagements can realistically support. In addition, timing may be difficult, because often project team members may roll on the project at different times.

Thus, many engagements may need a more flexible model with training time staggered in smaller chunks. For example, the training may be accomplished through some combination of formal classroom training done in waves, self-study, case study experience with mentoring, reading, and on-the-job training. The key point, however, is that a significant commitment to training must be made-whether done upfront or spread throughout the project.

There are several other lessons learned that can be drawn from the Eagle experience. Perhaps most important, training should be based on case studies. It should involve a significant degree of learning-by-doing including both design and coding exercises. Examples can be taken from the actual application to be built, thus reducing the perception of pure training investment. However, care must be taken to ensure that day-to-day project demands do not detract from the training. For example:

- Simple examples from well-known domains (e.g., checkbook application) ensure that the application requirements do not bog down the learning process.
- People may need to be taken away from the project site, or firewalls created, to enable a total immersion environment.
- Individuals should work in teams to simulate the collaboration necessary on an engagement.
- If real portions of the application are used, the team should manage expectations so as not to confuse training goals with producing deliverables.
- Reuse should be taught and encouraged through exercises that force the developer to browse.

On-going Support is Necessary for Developers to Scale the Learning Curve

On-going support is necessary to help developers continue building skills. On-going training is important because the entire development lifecycle is affected, to some degree, by the shift to components. An individual's first few assignments should be carefully planned to enable growing skills, and to identify people who demonstrate aptitude. Time must also be allowed for scaling the productivity learning curve, after initial skills are developed. This generally requires a fair degree of commitment from experienced frameworks developers to provide mentoring.

A Formal Certification Process Supports On-going Skills Development

Since component technology can result in many new skills and competencies, an ongoing, comprehensive skills assessment and certification process has proven beneficial. A certification process defines areas of competence and then critically evaluates individuals capability and progression. This can extend across design and coding skills to include familiarity with portions of the architecture. Peoples' skills can be assessed in compulsory design and code reviews. In effect, this becomes a component-specific skills evaluation.

A skills certification process helped to:

- More rigorously identify and describe competencies of what is really desired in terms of skills and competence; and, what habits should be discouraged and flagged as performance problems.
- Track peoples' growth-it encourages improvement by challenging people.
- Provide a more effective way to assign appropriate roles to people and offer up the opportunity for people to

grow into a more challenging role as quickly as they are adequately prepared.

Support more effective communications of what resources had which skills (e.g., through a wallchart)

Summary

Component-based development requires more time to scale the learning curve, because it has multiple dimensions. Component technology skills cover a wide-range of competencies including analysis, design, programming, and management. Thus, leveraging expert mentors and skills, investing in adequate training, and ensuring continued support are all key to success.

Team Organizations and Roles

This chapter discusses the team organization and roles involved with component-based development.

Manage the Team Size With Care

Team size should be managed carefully. Component-based development involves difficult coordination overhead. This stems from the higher degree of reuse and greater modularity of the system. A greater number of common components are reused across business functions. In addition, components are smaller than traditional modules. Thus, more work from multiple people must integrate smoothly. This complicates increasing the team size.

If a project slips off- schedule, caution should be exercised in adding people. Brook's fundamental law states:

Adding More People to a Late Project Makes it Later.

It is easy to underestimate the impact more people have on coordination and communications. Start-up costs can also be significant. New developers may have a learning curve. Even experienced developers must learn project-specific aspects such as the framework, business requirements, and team structure. These initial costs not only impact a new team member's productivity, they also reduce experts' availability for mentoring others.

Manage Expectations Regarding Developer Productivity

Industry Gurus Have Created Unrealistic Expectations for the Required Team Size

The need to manage team size must not create unrealistic expectations for developer productivity. High expectations have been fueled by many object industry experts who recommend a dramatically smaller team. Many have suggested that as little as 80–90% fewer people can accomplish an equivalent amount of work as a traditional development team.

However, experience does not support these exaggerated claims. Initial engagements have incurred considerable start-up costs such as training, architecture development, and building reusable components.

Some compelling evidence suggests object/component technology can improve productivity enough to reduce team size later in the software development lifecycle or for subsequent projects. Brooklyn Union Gas cut their maintenance staff in half and still accomplished as much or more work. Other firm experience has shown object technology reduced system test effort, enabling a smaller team. Large engagements have also realized benefits of reuse, significantly reducing development time for windows later in development. However, none of these experiences reported an order of magnitude reduction in team size.

Use Components As Work Packages

Components can Define Work Packages

Perhaps the most effective way to mitigate the risks of a large project is to simply avoid being large. Partitioning a project into smaller sub-systems is one way to reduce size. Component-based development is particularly well-suited to partitioning the development effort because the constituent components can map directly to team responsibilities. This simplifies division of responsibility and roles, because software and team organizations can mirror each other.

For example, FIG. **44** shows a high level picture of application component interaction for an Order Entry system. The boxes represent the application components of an application being developed. Orders are fulfilled by interaction with the Product, Customer, and Warehouse Application Components. These software application components can then serve to define the structure of teams and their collaborations with each other.

Keep in mind, however, the benefits of this partitioning approach may be influenced by the degree with which these components interact. Thus, determining the appropriate granularity of the components is a key, strategic design decision.

Greater Specialization of Roles is Necessary

Two recent engagements involved very large teams, in one case peaking at over 200 people working with object-oriented technology. In both cases, the engagement teams leveraged expertise in a manner somewhat similar to a traditional engagement. There were, however, important differences in scaling object-oriented development to such a large size.

One important distinction is the categories of expertise to be leveraged. For a traditional engagement, most developers tend to be divided in two basic categories—functional or technical. These two dimensions represent the primary types of leveraged expertise. That is, guidance is provided by functional and technical experts.

Component Development Requires Functional, Technical, and Modeling Competencies

A component-based project adds a third dimension—modeling. The skill set to model and represent behaviors and relationships in components and objects is a distinct, complimentary skill set to functional and technical skills. Thus, most projects find that they need a third type of expert—e.g., a component/object modeling architect(s), to provide direction.

Four primary online development roles may be defined:

window team members developed the window-specific functionality. Their role was biased towards consuming rather than providing common object behaviors, although there was some degree of the latter.

object model team members developed complex behaviors in the common object model; they also performed quality and consistency reviews for object model behaviors implemented by window developers.

frameworks team members developed the overall architecture mechanisms, providing structure and default behavior for the entire application.

server team members developed common data access and service routines on the server.

Architecture roles must be defined to support this greater degree of specialization. One engagement used the following partitioning strategy:

Functional architect-responsible for resolving decisions for what the system should do. This person is ideally a user with a solid understanding of systems, or a systems person with a good understanding of, and relationship with, the users.

Technology architect-responsible for delivering the platform, systems software, and middleware infrastructure to support execution, development, and operations architectures.

User interface architect-responsible for setting direction of the user interface metaphor, layout standards, and integrated performance support (IPS).

Application frameworks architect-responsible for designing, delivering, and supporting the application framework that provides the overall structure, or template, of the application.

Object model architect-responsible for identifying and resolving modeling issues necessary to achieve a high degree of business reuse and modeling consistency.

Note that the last two roles are especially unique to object-oriented and component-based systems. This means these architects have a learning curve to simply understand what their role means in the organization. Furthermore, the architecture roles require the deepest technical skills and should be staffed with the more experienced resources on the project.

One must be very careful in ensuring that application frameworks are not "over-architected". Experience has shown that many frameworks fall by the way-side for the simple reason that they were not built closely enough in conjunction with the application development. They become too theoretical, complicated and over-engineered making them performance bottlenecks and obstacles to simplifying the application architecture. It has been found that frameworks should "fall out" of the application domain as candidates become obvious. Experienced developers that work closely with the application can quickly identify repetitive constructs and abstract useful frameworks from this context.

Data and Object Model Architects Must Clearly Define Their Roles

One issue that must be resolved early on is the relationship between the role of the data architect and the object model architect. In traditional development environments data architects produce deliverables such as Entity Relationship diagrams. Since an Object Model is a superset of an E/R diagram, it is important to avoid treating the two as separate entities because this can lead to development teams working from two separate schemas. Viewing the object model as the object and data schema is very helpful in solving performance problems later and in promoting an overall understanding of the information schema of the system.

One strategy that has been shown to work is to include the senior data modelers in the object modeling team and give them appropriate object modeling training for their roles. This allows a natural migration of the object model to be the logical schema for the database model. However, this must be carefully managed so that good object model principles are not violated by a strong-minded data modeler who has not transitioned through the paradigm shift.

Greater Collaboration Between Roles is Necessary

Another distinction is the necessary coordination of roles due to the impact reuse has on the organization. In a traditional architecture, modules tend to be larger front-to-back slices of functionality. Reuse is primarily confined to technical services. Thus, functional developers can work independently, relatively speaking. The greater degree of reuse in a component architecture, on the other hand, requires much more coordination of effort.

The Organization Structure Must Enable Specialization and Collaboration

Component development requires a more sophisticated organization structure to support the increased specialization and collaboration of roles. Specialization is generally more important because more is being custom created-and less of the answer is codified as a proven solution. As noted above, well-defined roles are also important to ensure reusable components fit together.

At the same time, the structure must enable adequate collaboration of team members. Too many specialists may result in an organization that requires extensive coordination to deliver anything—e.g., a completed window. The organization must then strike a balance between "vertical" partitioning by function and "horizontal" partitioning by architecture layer. This is a classic management problem at an enterprise or project level.

Vertical Partitioning by Business Function Better Supports Collaboration

FIG. **45** illustrates a traditional organization structure including an activities component **4502**, a credit/collections component **4504**, a billing component **4506**, and a finance component **4510**. This traditional organization for most projects is vertically organized based upon business function with a technology architecture team. In this organization model, there would be one or more developers that are responsible for building a business function end to end. This works well for the following reasons:

aligns well with the business process and software decomposition

enables clear work direction—i.e., complete a window or report, front-to-back

ensures that complete functions work in an integrated, end-to-end fashion

teams better align to application releases

However, there are several potential shortcomings with this approach for an object-oriented system:

may force developers to learn multiple aspects of the framework (e.g., user interface and persistence services) which does not enable as much specialization: of skills

does not easily support consistency and reuse of business logic does not readily enable cross-function leverage of expertise

Horizontal Partitioning by Architecture Better Supports Specialization

Several object-oriented engagements have tried an alternative horizontal, or architecture-based, organization. FIG. **46** provides an illustration of a horizontal organization model **4600**. In this model, one or more developers are responsible for a horizontal layer of the system. Teams may be organized around layers such as technology architecture, frameworks, user interface, component/object model, or data access.

This approach offered the following advantages:

aligned with the object architecture

enabled cross function consistency and reuse of business logic

supported developing and leveraging specialized expertise

However, the following drawbacks were experienced:

over-the-wall problems in coordinating hand-offs of work amongst multiple developers

providing work direction to people became more complicated since they were poorly aligned with the business problem

managing completion of business functions becomes nearly impossible

A Workcell Organization Combines the Two Approaches

FIG. **47** illustrates a workcell organization approach including an activities component **4702**, a credit/collections component **4704**, a billing component **4706**, and a finance component **4710**. This approach combines the two previous approaches into a workcell. The primary orientation can be

aligned either way, but a functional orientation seems more natural for a business application. A cell is comprised of a complete set of specialized skills such as functional analyst, object modeler, application architect, and even user. Cross-cell architects then provide specialized direction for a particular role.

This approach, while adding complexity to the organization structure, has been used successfully on a number of engagements, and has been shown to combine the benefits of the two approaches. Of course, a drawback is simply an added level of organizational complexity—e.g., individuals at times taking direction from two different people.

Additional Effort is Needed to Ensure Consistency Across Workcells Additional effort is needed to ensure that each workcell develops application components in a consistent manner. It is important to define development standards and entry and exit criteria for all workcells. In addition, it can be useful to have a single person or group perform design reviews across the project.

A workcell's architect or frameworks developer can also help application developers better understand the architecture and use it consistently. Furthermore, the workcell architect serves as a good channel to feed new requirements—based on the application developers experiences—back to the architecture team.

Management May Need to Plan for at Least One Major Re-organization

The most effective approach depends on the team size, relative experience, and even the phase of the project. The dependence on development phase implies that management may need to plan for at least one reorganization. Unfortunately, re-organizations create significant team disruption, which must be considered.

Workcell Organization May Be Influenced by Other Factors

Some additional guidelines include the following:

Larger teams generally need to favor increased specialization, because they may almost always have a higher proportion of inexperienced developers. Thus, the specialized model supports developing areas of competency.

Early in an engagement more specialization may be required as an infrastructure of common components: and frameworks is constructed.

Once components are stable and integration of functionality is more important, then a collaborative, functionally-aligned or workcell organization may make sense.

The higher degree of custom development required in the architecture, the more specialization of skills is necessary; likewise, the more stable the architecture, the less important is specialization in favor of supporting collaboration

Complex, non-standard problems that cut across domains lend themselves to increased collaboration. On the other hand, more standardized problems can be solved with the specialized model. This experience is also consistent with management research of macro-organizations for an enterprise.

Workcell alignment may be influenced by the needs of the client. If the client's objective is to develop a highly reusable business component model, then it may be appropriate to have a single team focused on developing the component model. On the hand, if the client is most concerned about delivering business functionality, workcells should be aligned by business function.

The Organization Must Support Informal Structures

Whatever the formal organization, the project must enable extensive informal communications. Component development requires a tighter coupling between functional and technical design, because more commonality is incorporated into the architecture as common services. Thus, few important decisions can be made solely by one group within the project.

One large engagement combined several different strategies to ensure effective communications across organizational boundaries:

cross-cell weekly integration meetings were used to discuss and resolve low-level issues of global concern

temporary, cross-cell teams were formed to address many special issues—e.g., integration with an external system, an approach to handle addresses, etc.

temporary scout teams were formed to pilot the approach for a global change before rolling out to the entire team—e.g., the migration approach for moving subsystems into system test.

It's also important to consider the importance of informal sharing of information when many developers are undergoing training or there are global architecture changes underway. Geographic splits of a team can cause special problems.

Roles are Changed for Personnel at Multiple Levels

There often is not a direct mapping to the traditional roles that individuals expect. Analysts and Consultants may be given tasks with less creative freedom than they expect. For example, an Analyst role may involve less custom coding and more reusing, assembling, and testing of components. Design tasks for a new Consultant may also seem overly restrictive, because the challenge is to do things in a much more consistent, standard manner as dictated by the framework.

On the other hand, because everything is often so new to the entire project team, in some ways everyone is starting together from scratch. Thus, in a few cases, very talented Analysts with prior component experience have assumed lead technical design roles.

Traditional Hand-offs Between Designer and Coder are Problematic

The way roles work together is also different. For example, because of the iteration and coupling required between design and code, hand-offs from designer to programmer generally do not work well. One scenario used to leverage skills involved a lead designer creating the design, prototyping the solution, and stubbing-out methods with comments. The details were then flushed out by a junior developer. Leveraging by review and mentoring has also been key.

Summary

Crafting an organization structure for a component-based project involves balancing many complex factors. The most effective approach may depend upon the size and skill set of the team, the architecture structure and stability, and even the type of the application. Some additional considerations include:

Regardless of the chosen organization, care must be taken to ensure walls do not build up between teams

People's behavior may be influenced by the organization; that is, research has shown that the organization model may be reflected in the software architecture. For example, one engagement experience may shown that individuals may allocate behaviors to inappropriate components to avoid having to collaborate with other developers

Workcells combine the benefits of horizontally and vertically aligned organization structures, and have been used successfully on a number of engagements.

Planning and Managing Development

This section discusses strategies for managing a component-based development process. Two alternative development strategies are:

Waterfall approach

Iterative approach

Much of the one's experience may be with large, mission-critical projects. Moreover, large projects introduce additional, inherent complexity. Therefore, these issues may be discussed primarily from a large project perspective.

A Tension Exists Between the Waterfall and Iterative Development Models

The Waterfall is the Traditional Approach to Managing Software Development

Systems development traditionally relies on a waterfall model. This approach manages development in sequential phases of activity such as analysis, design, code, and test. The waterfall model provides a controlled, orderly process for developing a system. Work is sequenced to ensure that the design addresses the correct requirements, implementation is based on upfront design, and system testing verifies and validates thoroughly unit tested components.

Despite these benefits, the waterfall model introduces potential problems. For example,

Requirements may be difficult for the user to understand without prototyping the user interface or functionality

The design team may not be prepared to specify an effective design without gaining implementation experience

A team may not be positioned to generate reusable components for itself, unless a team works ahead to construct an architecture or component model during the design phase

Iteration Helps a Team Address Risks and Gain Experience

Because of the above shortcomings, much of the OO and component community recommends some variation of iterative development, in which analysis, design, and coding activities overlap to some degree. A theme in these variations is the need to address risk by proceeding further in development sooner. Both the gained information and experience can influence the approach taken in the current phase.

However, iteration also has drawbacks. The team may slip into hacking, by simply skipping design before coding. Or, a team may use iteration as an excuse to not exercise due diligence in completing tasks. Defining and estimating milestones is also hard.

A Project Must Weigh the Tradeoffs Between Waterfall and Iterative Models

Thus, a tension exists. The waterfall promotes discipline and control in the development process. In contrast, iteration proves out assumptions, gains advance experience, and addresses risks. Balancing these conflicting goals is difficult on a large scale.

Distinguish Between the Macro and Micro Process in the Workplan

Both the waterfall and iteration have pros and cons. A hybrid capitalizes on the advantages of both. If they are merged, one or the other must inevitably dominate the structure of the high-level project plan. That is, the plan must start somewhere—either by defining iterations or waterfall-like phases of completion.

For example, defining iterations of the system or subsystem would result in high-level project phases such as:

first working version

refined working version

final, released working version

In contrast, a more traditional waterfall structure would result in high-level project

phases such as:

requirements definition

preliminary design

detailed design and/or coding

testing

A macro plan reflects the high-level development phases The micro plan shows the tasks of a specific phase or team

Distinguishing between a macro and a micro process provides a practical compromise. The pure, traditional waterfall has no distinction. There, the entire workplan and accompanying development approach sequence analyzing everything, then designing everything, then coding and testing everything, with no overlap. The same uniformity between macro and micro processes applies to a pure iterative model. In this case, the workplan reflects multiple iterations of the entire application. However, in either case, such extremism is not necessary. Instead, a plan can merge the two approaches by distinguishing between the:

macro, high-level plan, and

micro, phase or team-specific plan.

In summary, an exclusively waterfall or wholly iterative model are, independently, too simple. A balance is required. Distinguishing between the macro and micro process gives management the intellectual freedom to craft a workplan that reflects a mix of the two styles. The downside is that this introduces significantly more effort and complexity in the planning process.

The Macro Process for Large Projects Should be Waterfall in Nature

Managers are Averse to Iteration, Because it Expects Re-work, Ipso Facto

The previous section laid out two alternatives for combining the macro and micro process. For large, custom development projects, experience has shown that defining the macro process along the lines of a waterfall structure is most effective. Client and firm project management are typically uncomfortable with defining milestones and estimating work with iterations. The common statement is, "How do I know when I finished the current iteration?"This concern is valid—on a large-scale, "complete" can be difficult to define. In addition, most managers have trouble embracing a process that expects and even allows mistakes on such a large scale.

Iteration Does not Scale Well due to Communications Overhead

Aside from these psychological considerations, large projects introduce significant risks due to the complexity of coordination. A large team has a much greater inertia, because of the time delay and errors introduced in human communications. Any change takes much greater effort and time to implement. Correspondingly, once made, changes are more difficult to reverse. Greater reliance on documentation is often necessary to avoid miscommunications.

Many decisions must then be considered from the vantage point of their ease of communication. This complicates iteration. For example, if analysis, design, and code overlap extensively, then by definition, requirements and design change later in the process. Communicating wide-scale changes late in development can be inefficient, wreaking havoc on existing code. Thus, iteration does not scale well to the macro level, because of communications overhead.

It's important to re-state, however, that a pure waterfall introduces many problems for component development due

to its intrinsic reuse and newness. Thus, many of the lessons below emphasize variations of iteration and how they can be merged with a waterfall approach.

Incremental Development May Help Manage Scope and Risk

Incremental Development Partitions the System Roll-out Into Releases

Perhaps the most effective way to mitigate the risks of a large project is to simply avoid being large. Incremental development addresses risk by reducing the necessary team size and scope. "Incremental" and "iterative" development are often used interchangeably, but they are different approaches.

Incremental development partitions the system roll-out into successive releases. For example, the initial release of a customer system might comprise order processing, followed by a subsequent release for billing, and a third release for collections processing. Thus, incremental development adds new functionality, while iterative development continuously refines existing functionality.

Incremental development is often more palatable to managers than iterative development, because there is no explicit notion of repetition. Yet, the desirable benefits of iteration are often realized. For example, releasing consecutive versions of the system creates the opportunity, and often the requirement, to refine the initial release. The early implementation experience can also provide important productivity benefits for subsequent releases. This experience may also help drive out technical requirements for future releases, improving the analysis and design process.

Incremental development avoids the complexity of a big bang integration. Furthermore, although an incremental approach delivers less in each successive release, it can deliver higher priority portions of the system much earlier than a traditional approach, thereby recognizing business benefits in a shorter time frame.

Despite these benefits, incremental development is not a panacea. Many times a big bang conversion has proven necessary, if the cost and risks of having parallel systems and bridges, performing conversion, and rolling out training are high. These costs must balance those introduced by the delayed delivery of business benefits and the risks !implied by increasing scope and team size. The urgency of the business and the desire to manage development size may sometimes favor an incremental approach.

Incremental Development can Also Apply to a Single Development Release

Even when incremental development does not prove feasible for entire application releases, the approach can be effective on a smaller scale. For example, the development and release of a single application may require extensive integration of diverse behaviors in a reusable domain component model. The domain components must be put in place early to allow reuse; then, behaviors are incrementally added as the business use cases are analyzed and designed. As in the previous case, iteration naturally occurs; but, again, incremental proves to be a more acceptable metaphor.

Enable top Down and Bottom up Development

Different Categories of Tasks Should Proceed at Different Rates

Whether applying a more waterfall, iterative, or incremental process, the dependencies between tasks require careful consideration. Different categories of tasks need to proceed from problem-definition through solution at different rates.

FIG. **48** illustrates the Enterprise Information Architecture (EIA) model **4800**. This model adapts to component terminology, with the relatively minor change in layer five from data architecture to domain component model.

Both Top-down and Bottom-up Models are Necessary

This model incorporates the idea of simultaneous top-down and bottom-up development. Much development effort may follow a relatively top-down, sequential approach. This includes analyzing and designing: the business environment and processes, domain model, and then application. Concurrently, an architecture effort proceeds bottom-up. This builds: the technology architecture of platform system software, hardware and infrastructure services; and then application architecture, or frameworks. Top-down and bottom-up efforts then conceptually meet in the middle, integrating the application framework with the application.

Both the Architecture and Component Model Lead Application Development

The need to start architecture implementation early is well-understood for traditional or component-based client/server development. What is different with component-based development, however, is the need for the component model to lead the application and user interface development.

Starting the component model early is essential to enabling reuse of a consistent, cross-functional set of business components. These core domain components must be defined early, at least in preliminary form. Otherwise, the simultaneous integration of functionality from many windows or reports would be extremely chaotic. In addition, developers may implement business logic in the user interface layer, rather than in the business components where it can be reused. Furthermore, early design of the component model before user interface logic improves the odds of creating a pure component model, decoupled from the interface.

Design Efforts Should Focus on Component Interfaces

Interfaces are the contracts for the services that a component provides. Clients of a component are concerned with what the interface specifies, not how it is performed. It is the interface provider that is concerned with the implementation. By correctly defining interfaces during design, it is becomes possible to independently develop components. When interfaces are changed, component assembly becomes much more difficult and rework is often required. Thus, design efforts should pay additional attention to the completeness of interface specifications.

Architecture Development Must Start Early

A Tension Exists Between use Cases and Frameworks

As with client/server, architecture work must start early. As noted above, this is particularly challenging because of the level of application reuse in a well-designed application framework and domain component model. Because of this reuse, the framework must be heavily driven by application requirements, or use cases. Yet, the architecture team must stay one step ahead of application development teams to ensure that the architecture and component model are ready in time to be reused. Thus, a difficult tension exists between use cases and frameworks.

The tension between use cases and frameworks can be simplified to the extent that third-party or standard architectures such as Eagle can be leveraged. In addition, experienced architects may bring their knowledge of which services are common across applications and can be addressed earlier than application-specific architecture services. In any case, the following guidelines should be considered, particularly for custom architectures:

The architecture should be defined and prototyped, if necessary, early in the preliminary design

SAP - Exhibit 1105
SAP v. Valtrus et al. - IPR2025-00420
Page 206 of 281

The architecture should be complete-at the very least, the development architecture and overall framework, prior to developers actually coding; the design must be-in place earlier when functional developers start detailed design; private architecture aspects may be deferred

Time must beplanned for architecture support based upon unforeseen use cases, performance tuning, documentation and developer mentoring

Developing a custom application framework should be estimated as a set of tasks in addition to much of the traditional technology architecture development

Failure to Develop the Architecture Early May Reduce its Efficacy

If the architecture is not completed ahead of the application, developers may have the tendency to build architecture services in the application layer. Clearly, this may lead to diminished reusability and more difficult maintenance. By defining the architecture services early and communicating them clearly to the application teams, these problems can be avoided.

A related problem with object architecture and frameworks is that the line between the application and architecture can become blurred. These architectures may provide so much common functionality that it is difficult for teams to distinguish who is responsible for what. For example, it may not be clear that a function should be provided by the application architecture team, technology architecture team, or application team. This problem can be resolved by better communication and coordination across teams. Workcells are one approach that has proven effective in this area.

Component-based Development Requires More Granular Milestones

The Macro Process Uses Traditional Milestones

The milestones used to track the macro process generally remain the same as for traditional systems lifecycles. Project management may still be interested in monitoring the progress of high-level milestones such as the start and end of design, or the start and end of construction.

The Micro Process may use More Granular Milestones

On the other hand, the micro process may have more granular milestones than traditional systems. Whereas a business function in a traditional system may be composed of single front-to-back module, a component-based system may provide the business function using several collaborating components. Thus, component-based systems inherently have more work objects to track. While the increasing number of work objects may seem to be a management burden, it can provide a more fine-grained reading on the development process.

Another difference from traditional systems is that milestones may be more oriented around elements of the systems (windows, business components, and architecture components), rather ,than business functions. Furthermore, some types of milestones may be more important than others. For example, if there is a significant amount of functionality in the business components, then there may be more milestones associated with the business components than with the user interface.

The Micro Process Should Vary With the Type of Development Role

The Micro Process Must Compliment the Macro Process

Assuming a waterfall-like macro process, as described above, the challenge of the micro process is incorporating an effective level of iteration into this management framework.

Different roles for team members require different development methodologies. For example, possible roles are:

Application developer—responsible for implementing a particular business function, such as accepting bill payment. This focuses on the application-specific design and implementation tasks such as: working with a user to define requirements or use cases, designing the user interface, and implementing application functionality.

Component Model developer—builds, refines, and supports the core, reusable business components in the application.

Frameworks developer—responsible for the application and technology architecture that provide common services and control logic for the application.

These roles do not necessarily correspond directly to organization assignments. Whether these roles formalize as teams, identities within a workcell, or possibly different hats a single person wears is an organization decision that depends on the project size, individual skill sets, and other factors.

Within the Micro Process, More Parallelism can be Achieved

At the micro-level components make it more reasonable to execute more development tasks in parallel. Components enable this by providing more discrete work objects that are more clearly separated by their interfaces. Because interfaces are the contracts through which components interact, the internals of a component can be developed independently as long as the interfaces are respected.

Dependencies on Shared Components Need to be Managed

On the other hand, since some components may be reused throughout the application, it is a good idea to start them earlier to provide a solid base for the rest of the system. Thus, a greater dependency on certain reusable components may require additional planning effort to correctly sequence the work.

Application Developers can Follow a Relatively Formal, Sequential Process

A significant portion of application development can execute in a sequential manner. This excludes the development and maintenance of the core component model and application frameworks discussed below. For the application developer driving out requirements, design, and implementation of window functionality, development can proceed very similar to that of a traditional, client/server GUI project. Particularly early in development, many aspects of the methodology can be very similar such as CAR (Control Action Response) diagrams.

During implementation, detailed design and coding steps may overlap. However, the rules and guidelines for sequencing these should be spelled out in rigorous detail. Note that this does not imply iteration per se, although that may be a desirable side-effect if controlled. Rather, this approach merely suggests tactically interspersing the design and code activities, particularly to aid in-experienced developers in transitioning from design to code.

Define Concrete Milestones With Short Intervals

An important difference in managing efforts with this type of overlap is the need to define much more concrete milestones with shorter intervals. This is necessary because a detailed design or coding phase definition loses meaning if they overlap extensively. Milestones represent more concrete, visible accomplishments, such as:

all basic layout and behaviors designed; complex behaviors identified, but not completely designed

view and application model integrate with domain model window opens

data access from server coded and tested

full detailed design of special processing or complex behaviors

complex behaviors coded and tested

SAP - Exhibit 1105
SAP v. Valtrus et al. - IPR2025-00420
Page 207 of 281

Incrementally add Behaviors to the Reusable Component Model

A previous point emphasized starting the component model development early, because many of these components are reused in many business functions. Thus, their preliminary structure must be available before multiple windows require their use. This implies that many different behaviors may need to be continuously integrated into these components over and over. The component model development, then, is very much event-driven like a factory.

Incremental is a Good Term for Continuous Integration of Behaviors in the Component Model

The salient feature of this development style is that behaviors are incrementally added to the reusable component model throughout the development. Iteration and refinement often occur naturally in this process. However, incremental proves to be a more acceptable term for management.

When developing in this fashion, tracking status is difficult. Management traditionally tracks status by number of windows or reports complete. Yet, in this style of development, the windows complete may fluctuate dramatically. Some windows may not achieve completion until very late in the project, when the component model stabilizes. Yet, many behaviors may indeed have been completed. This creates an illusion that the project is further behind than reality. More sophisticated status tracking is therefore needed.

Iterate to Address Risks or High Degrees of Uncertainty

Prototypes "buy Information "That Reduces Risk

Iteration is required to address risks involving a high degree of unknown. These risks tend to increase with component-based development, primarily because of its novelty. Thus, the need to iterate is often less intrinsic to component-based development and more related to challenges naturally resulting from unfamiliarity. What is now "traditional" client/server development faced similar difficulties years ago.

In some cases, this unknown requires experimentation. For example, a throw-away prototype has the explicit intent to "buy information" for reducing risk. Prototypes are a special case of iteration involving less commitment to salvage the work. Whether the prototype is salvaged or not becomes less relevant, because the primary value is in the information obtained in the process.

Several different categories of risk require iteration. None of these are unique to component-based development But they tend to be more important with component technology because, again, so much of the underlying technology and methodology are new. Some of the types of prototypes are (These are similar to other definitions):

usability, or user interface prototypes

performance prototype

proof-of-concept prototype

pilot process prototype

These categories may be addressed with throw-away prototypes, initial working models which are later refined, or some combination. Use of "prototype" below generically refers to either style.

User Interface Prototypes Help Users Understand Requirements

User interface prototypes address the difficulty that users have in defining requirements without implementation examples. This phenomenon is analogous to the Heisenberg Uncertainty Principle. This law of modem physics states that the simple act of trying to observe the position or velocity of electrons affects the result itself. Likewise, users' perceptions of their requirements may be changed, sometimes dramatically, by observing examples of the potential solution. In many cases, these prototypes have been used as a standard design deliverable with repeated review and refinement with the user.

An important consideration, however, is scope control. There is a very complex management problem when iteration is used to drive out requirements with users. Experience has shown that users may assume that exploring an alternative implies that the functionality may be implemented. Thus, some change control procedures need to be defined and managed, even if they do incorporate some flexibility to incorporate enhancements.

Performance Prototypes Address Global Architecture Issues

Performance prototypes primarily address technology architecture questions. For example, the architecture team may need to decide early on whether to use messaging, remote procedure calls, or shipped SQL statements for distribution services between client and server. A prototype is often the only way to identify the most effective solution.

Proof-of-concept Prototypes Address Complexity

Proof-of-concept prototypes address areas of significant technical or functional complexity. In the most extreme case, the team may be uncertain as to whether they can even develop the logic within the specified quality parameters. Or, it may be too difficult to design a solution upfront, because of a mix of technical, functional, and maintainability issues. In such cases, the team may need to implement alternatives for evaluation.

Pilot Process Prototypes Provide Experience for the Team

Pilot process prototypes are used primarily for the team to gain experience. They typically use a front-to-back, slice of the application. This is similar to incremental development, which delivers the solution to a portion of the business functionality. Such learning benefits are not unique to pilot prototypes. The distinction of a pilot prototype, however, is that gaining experience is the primary purpose of the effort. The leaming may focus on technology, business function, or methodology.

Confine Highly Iterative Tasks to Experienced Framework Developers

Iteration demands very experienced developers, to understand the criteria for completion. Thus, tasks that require a very high degree of iteration, such as technical prototypes or development of reusable components, should be confined to a small team of experienced developers. These individuals usually comprise the architecture frameworks team.

One heuristic is to staff the frameworks team with the most experienced component developers, comprising about 5–10% of the total team size. There is another reason to allow the most skilled developers to iterate more—research has shown that very experienced software developers naturally work more productively this way. Thus, productivity for very talented architects may increase when given freedom to iterate as necessary. On the other hand, anecdotal evidence tells us the opposite is likely true for inexperienced developers.

This is not to say that application developers should never iterate—it's really a question of degree. One approach is to use selected application developers on scout teams that form for one-time efforts and then disband. These efforts may, for example, address an initial pilot process or other type of prototype mentioned above. Even then, these efforts are usually best coordinated by an experienced developer, presumably from the frameworks team.

For Difficult Tasks Plan Three Iterations

For those aspects of the system that require iteration, the question still remains, How do I know when I am done? Experience has shown that three iterations are usually required, for example:

- design and develop initial working model
- refined working model and pilot
- roll-out and support

The need for three iterations has been observed in so many cases that some consider it a magic number. For example, the three iterations defined above parallel very closely a maxim quoted by Kent Beck, a well-known Smalltalk expert, Make it run, make it right, make it fast.

Difficult Components Should be Designed and Then Prototyped

An initial working model phase designs and prototypes the component or framework. Prototyping may be necessary to evaluate two or three alternative approaches. In these cases the initial design represents a strawman, until receiving validation from implementation. Only then can things be finalized and reviewed for sign-off.

Piloting Reusable Components With Developers is Necessary

During the refinement and piloting phase, the component or framework is completed with any remaining functionality and then used in a pilot case. Coordination is often necessary with a pilot developer who is a client of the reusable piece, to ensure that it works in an appropriate use case. Typically, the pilot process generates several refinements or changes. Pilot developers need a flexible, positive attitude to handle potential instability.

The component must then be documented and rolled out for reuse to all developers. In many cases, the roll-out requires a formal group meeting to answer questions. During the support and refinement phase, the component is refined as other use cases generate new requirements, and bugs or performance problems are identified. Although the implementation details of the component should not be widely known, it is critical that developers thoroughly understand the purpose and public behaviors of the component. If they do not, then they may not be able to effectively reuse and debug interactions with it.

Summary

A traditional client/server implementation often incorporates some limited iteration with a waterfall approach. This iteration is usually confined to technology architecture tasks. Component-based systems tend to require somewhat greater iteration for three key reasons:

- Reusability often requires actually reusing the component to ensure the reused piece meets requirements
- Component technology is new, thus iteration helps address greater technical risk
- Component skills and methodologies are emerging, therefore the team often gains valuable experience from iteration

Managing iteration is difficult but possible. Usually the plan must incorporate a hybrid of waterfall, incremental, and iterative models as appropriate. The right process depends on the organization or teams' skills, the degree of technical risk, and the specific application and business requirements.

Testing

Testing typically consumes anywhere from 50–80% of development effort. Despite this relative importance, testing receives little emphasis by component-based methodologies, which focus primarily on analysis and design techniques. This section presents testing lessons consistent with the primary themes in The Testing Process Practice Aid, produced by the Re-inventing Testing Project. These points merit increased emphasis, however, because experience has shown component-based systems increase testing complexity.

Testing is More Complex

While a component-based approach may be simpler to test than a strictly object-oriented approach, testing is still more complex than a procedural system, because component architectures:

- decompose into a much greater number of components than equivalent procedural modules, introducing more complex technical integration achieve a greater level of reuse, which is a blessing once highly reusable pieces stabilize, but remains a substantial challenge while they undergo development
- utilize flexible, messaging between components that creates a larger number of potential test execution paths
- usually develop with some degree of iteration, which jeopardizes the benefits of phase containment

Testing Requires More Phases

The Testing Process defines a three-step process, very similar to traditional Method/I, as follows:

- component test—a test of an individual module or program that is specified and coded as a single unit
- assembly test—a test of a set of programs that communicate with each other via messages or files, usually equivalent to a user interface dialog or a batch string
- product test—a test that verifies the technical and functional implementation supports the business process

Object Systems Require an Initial Atomic Test Phase

When building components using objects, testing can logically follow these same three primary phases, at a high level, preceded by an initial atomic test phase. An atomic test phase is required because a well-factored object system may typically have at least 10 times more objects than procedural modules in a traditional system. This finer granularity requires testing and integrating more units at multiple levels.

Completing a Window Requires Several Stages of Component Test and Integration

A traditional approach often defines the initial component (unit) test as a working window, front-to-back. In a component-based architecture, on the other hand, a window may often utilize behaviors of several components. This results in too much integration for an initial component test. In fact, several stages of integration must occur to complete a single window.

Consider a customer that encapsulates other related components such as credit profile and address. This customer aggregation even represents too much functionality for an initial component test. Simpler components such as the address must be tested first. Testing individual components or tightly coupled aggregations should occur in the initial component test phase.

The assembly phase then tests the integration of these components. This test phase differs from a traditional assembly test, because more components must typically be integrated, particularly for the vertical, front-to-back functionality from window to database. This adds to the horizontal integration of interdependent windows in a dialog. In contrast, a traditional assembly test concentrates much more heavily on the horizontal dialog test, since the front-to-back window functionality is often just a single module.

The timing of the assembly test may vary depending on the development teams organization. If there are a number of

developers working on a functional slice of the application, then early integration helps to ensure that developers are working in concert and simplifies integration later. Conversely, the issues of integration may not be as significant if a single developer is working on an entire business function, end to end.

In summary, the collective atomic, component, and assembly test phases require much more detail in terms of milestone definitions, status tracking, and methodology development.

Testing Component Collaborations Must Occur in Several Phases

The process of testing and integrating behavior of collaborating components must occur at multiple levels. In particular, distinguishing between the component and assembly test phases can seem somewhat arbitrary. A well-factored architecture may have identifiable boundaries, however, as noted above. Thus, coming up with good definitions of aggregation—that is, cohesive groups loosely coupled from other groups, is equally critical to testing as to design. The component aggregation must then support an effective partitioning of the application architecture and team organization.

Testing Requires a Flexible Organization

On large projects, the set of components involved in a business event are often developed by many different people. Thus, the complexity of team integration further complicates the testing effort. Well-defined component boundaries in the software and organization are certainly key. However, the organization must expect the need to support flexible integration testing teams that form to ensure a particular business function works correctly across partitions.

Testing Effort Shifts Earlier in Development

The System Test Phase Should Go Faster

The implications of greater modularity and flexibility discussed above increases complexity in the atomic, component and assembly tests. Once the architecture and highly reusable components in the component model stabilize, however, system test is simplified. Thus, component-based systems require shifting testing effort earlier in the development lifecycle.

Phase Containment Requires Greater Attention

Experience has shown that defects become increasingly more expensive to fix later in the development cycle. Phase containment strives to decrease both the number and cost of fixing errors, by testing steps early in the development lifecycle through verification and validation of work. FIG. **49** illustrates a V-model of Verification **4900**, Validation **4902**, and Testing **4904**. Exit criteria might involve, for example, compulsory detailed checklists or code reviews before work is promoted to the next phase.

While phase containment is not unique to component development, its importance is heightened. Since many portions of the component model may be reused by literally every window developer, quality is critical. That is, a high quality design and implementation for core components increase the productivity of every developer; however, the converse is also true—mistakes tend to penalize all developers. Thus, thorough testing and attention to quality in early development steps is important.

Iteration Complicates Phase Containment

Yet, incremental or iterative development complicates phase containment. Phase containment presumes a waterfall model. For example, a module or component should not be passed to a later phase such as coding until the design has been validated and verified. In contrast, overlapping design and code implies coding starts with an incomplete design. This puts at risk any efforts to define precise milestones so critical to effectively track progress.

Iteration Requires More Detailed Exit Criteria

Thus, iteration requires more detailed completion criteria. For example, different iterations of design must have very explicit scope boundaries to ensure that the completion of an iteration is adequately defined. These must be accompanied by strong adherence to proper procedures as components are promoted through various development stages. Even with such efforts, however, experience has shown that later designs tend to impact previously working code. Significant regression testing must be expected, as discussed below.

Automated Regression Testing is Usually Necessary

Regression testing is necessary because of iteration, inheritance, and extensive reuse

Experience has shown that the higher degree of reuse in an component model and application framework makes it very difficult to protect implemented components from subsequent development. Developers must then verify previously tested components as they incrementally add functionality to the system. Automated regression testing can save time by ensuring that areas that are impacted by changes are properly tested.

Moreover, regression testing capabilities are absolutely essential if an extensive architecture framework is developed. Component-based development allows an application framework to abstract both technical and functional behaviors. This greater level of reuse necessitates that the framework evolve with the development of the application. Unfortunately, this implies changing the technical environment of the application even as it approaches delivery. To effectively support these enhancements requires re-testing at many different levels.

Using Objects Increases the Need for Regression Testing

When developing components using objects, regression testing becomes even more important. For example, inheritance often results in sub-classes coupled to their parent. A parent class may have side effects with subtle implications to children, which are difficult to identify for test cases. Experience has shown that even seemingly innocuous changes to a parent can damage previously tested subclasses.

In general, an inherited feature must be retested in the context of the subclass. Retesting can only be avoided if subclasses are "pure" extensions of their superclasses; that is, if they don't override any methods and do not modify inherited instance variables. Furthermore, test cases usually cannot be inherited when overriding a method. Slight differences in logic and data declarations are indeed enough to invalidate the superclass' test cases, requiring new test definition and input data.

All of the above considerations result in substantial re-testing. Enough so, that a manual approach to regression testing can be extremely cumbersome. In particular, changes to shared components or changes at or near the root class of a deep inheritance hierarchy can have widespread impacts. Thus, automated facilities for testing should be considered a mandatory element of component development.

Combine Automated Testing With An Automated Build Process

Automated testing can also make the configuration management process more efficient. By using an automated test process to verify that the latest version of the application is working correctly, it is possible to give the development and testing teams more stable releases. For example, simple defects such as incorrect interfaces can be detected before the application is even distributed.

GUI Scripting Tools Alone are Usually not Sufficient

Capture-playback GUI testing tools have proven effective. However, for many applications these are not completely sufficient. These tools may only re-validate that the application appears to function properly. Experience has also shown that applications may sometimes use widgets or technical elements of the user interface not supported by a particular tool.

Self-testing Features Should be Built

A more comprehensive testing framework should be considered that incorporates the notion of self-testing components. That is, the component may have behaviors, or indeed contain a tester component, that feeds the class a test suite, and validates the resulting behavior. Note, however, that test components rarely test behaviors of just a single component in isolation, because any meaningful behavior usually cuts across multiple components. The test can still obey encapsulation, though, by testing the group as a single black-box, rather than taking short cuts which may undermine the validity of the test.

Testing Frameworks Requires More Attention

The use of frameworks in component-based systems also increases the complexity of testing. Frameworks add complexity for the following reasons:

- Foreseeing all the uses of a framework is hard a priori. Thus, verifying correct behavior is difficult because the test cases may not be complete.
- The test approach can require extensive scaffolding to support emulating the application intended to use the framework.
- Framework development is usually undertaken early in the project so that it is ready to support application developers. This implies incomplete knowledge of requirements for the frameworks team.
- The stakes are high, because the framework usually provides a reusable structure for many developers.

There are essentially two methods for testing a framework:

- Emulation approach—by building a comprehensive test environment that emulates the application.
- Pilot approach—by using application developers as pilot users in the testing process.

The emulation approach protects application developers from the testing effort, and is generally more consistent with a formalized approach. Not doing so opens the door to rolling out untested frameworks. On the other hand, creating a redundant simulation environment of the application use cases can be expensive.

The pilot approach may be more productive by leveraging real application code. In addition, it encourages training and knowledge transfer to developers. Finally, it helps ensure accurately covering requirements. It is important to use application developers for the pilot, not the architects. This may provide an objective review of the framework's usability. The primary drawback is that it takes a developer away from the application; and, as noted above, may result in frameworks developers feeling relieved from thorough testing. Experience has shown that a hybrid of the two is usually necessary.

Summary

Experience has shown that initial component development projects require more effort in testing. On the other hand, the later stages of testing can be more productive by effectively leveraging encapsulation of components and large-grained components. There is reason to believe that these benefits can be leveraged sooner if the team pays increased attention to testing early in development. Testing should be a part of the entire development process comprising:

- phase containment principles with explicit objectives and exit criteria such as checklists and peer or lead reviews
- automated regression testing capabilities
- self-testing components
- more detailed testing phases and milestones
- comprehensive procedures with disciplined enforcement

Development Architecture Considerations

This section highlights key messages for development architecture teams in regard to supporting teams and tools within a component based development project.

Building Systems That are Dramatically More Responsive to Change Require a Dramatically Improved Development Architecture.

What does it mean to be more responsive to change? The solutions one builds must be more:

- Flexible. Making it possible to replace or modify application components with minimal impact to the other components in the system.
- Scalable. Giving you freedom to distribute and reconfigure application components to meet the scalability requirements of the client.
- Integratable. Allowing you to reuse the functionality within existing systems by wrapping them as components within the new application.
- Adaptable. Giving you freedom to deliver an application to a variety of user types through a variety of delivery channels with minimal impact to the application itself.
- Reusable. Making it easy to quickly assemble unique and dynamic solutions from reusable components.

Component-based development pushes us forward on all of these dimensions, and although it's relatively immature, we're better off than we were before. Metaphorically speaking, we've climbed very close to the top of the mountain that represents traditional development. The view is satisfactory, but we know there is something better, so now we're climbing the mountain that represents component-based development. We have yet to reach the top, but we're already higher than we were before.

On every component-based development project, teams spend time evaluating and establishing the environment in which analysts and developers create the deliverables. A workbench must be established that expedites the flow of deliverables through the different phases of the project. In component- and object-based solutions, these phases are very connected. This is largely because each subsequent phase tends to be an elaboration and refinement of the deliverables completed in previous phases. In addition, there is a strong desire to link deliverables and requirements from the earlier phases to the deliverables from the subsequent phases.

On a typical project one finds the following tools used in the software development process:

- General diagramming tools: Visio, ABC Graphics, etc. for workflow and operation diagrams
- MS Office: Word class and component specification templates, Excel scenarios,
- Object Oriented CASE tool: class and component models, component/class specifications, message trace diagrams

Database design tools: Erwin, Oracle Designer, etc.

Integrated Development Environment(IDE): Visual Studio, Visual Age for Java, JDeveloper, Visual Cafe

Source code configuration manager: SourceSafe, ClearCase

An inordinate amount of time is invested in the macro process of how to capture and link information in a way that it can be used effectively through the course of the project (e.g., moving the models from the CASE tool into the source code of the targeted IDE environment). Teams should tackle early the selection of deliverables in each phase and which tool the deliverable may be created and maintained within. In addition, they should determine whether the deliverable is to continue to be enhanced in subsequent phases of the project through the iteration process.

Today's Dilemma . . . no Easy Answers, Yet

To realize an environment that enhances the productivity of your analysts and programmers is a challenge for any project, but for projects building component-based solutions, it's even more difficult because of the technology's relative immaturity. You won't find any easy answers, yet.

Generally speaking, the resulting environment gets the job done, but consists of tools that are crudely integrated with no central repository. This results in redundant data and manual cross-referencing. It can also cause problems during the transition from Design to Construction ¾ gap that's always been difficult to traverse. Other typical concerns include a tool's ability to meet usability, scalability, and multi-user requirements.

Ideally what would greatly increase the productivity of the development architecture is a seamless integration of tools in the workbench and the ability to "plug in" whatever tool is most appropriate for the capture and communication of a particular deliverable. FIG. **50** portrays a development architecture with a seamless integration of tools which can be plugged in for the capture and communication of particular deliverables. Shown in FIG. **50** is the relationship between a process phase **5000**, deliverables **5002**, tools **5004**, repositories **5006**, and an information model **5008**.

One solution center working on this architecture found that the current state of integration with tools was so constraining that the picture in FIG. **50** had to be resolved with many compromises for new component work. There were many custom scripts created and manual processes defined for leveraging the flow of information between phases and tools.

FIG. **51** shows a design architecture with the compromises made for today's component construction environment. Shown in FIG. **51** is the relationship between processes phases **5100**, deliverables **5102**, tools **5104** and storage **5106**.

Key Considerations

A development architecture should provide an environment for component-based solutions that supports a team through the Analysis, Design, and Construction phases of the development process. It should also serve as a productive environment for the on-going maintenance of an application. Conceptually it should integrate all of the necessary tools through an information model and most ideally through a central repository. The following are considerations that all component development architecture must consider.

1. Support Custom Process. The present invention uses a robust process for developing component-based solutions. It includes deliverables that are above and beyond the Unified Modeling Language (UML). Furthermore, projects often customize it. The environment must provide the ability to extend the information model (i.e., the meta-model).

2. Versioning & configuration management. The environment should provide the ability to version objects within the common information model at any level of granularity, keeping track of these changes over time. It should provide the same ability for composite objects (i.e., configurations of smaller objects).

3. Scalability. The repository-enabled environment must be able to support hundreds of users simultaneously, and hundreds of thousands of repository relationships. It should also scale downward, so that small project can use it. This is a major criterion for usability.

4. Query and impact analysis. As organizations begin to maintain their own component-based assets, they must be able to analyze the impact of change requests (e.g., where-used searches). The ability to trace requirements is also critical.

5. Asset catalog (reuse). As organizations begin to reuse existing assets, it may become increasingly important to provide a catalog of components, frameworks, patterns, etc. The catalog should make it possible to search for relevant assets in a wide variety of ways. It should also provide a means for applying frameworks and patterns.

6. Code generation. The ability to generate the application structure from the model is essential to high productivity. Furthermore, this step should be transparent to the user. As far as the user is concerned, a change to the model is a change to the code.

7. Desktop Tool Integration. The repository-enabled environment must provide integration between all desktop tools (e.g., MS Office, Visio, OO CASE tools, designers, etc.) through component object models such as ActiveX. In addition, these tools must have access to the common open information models.

8. Non-redundant storage. The environment should avoid redundant storage of information, whenever possible. Everything from training to documentation to active components should be automatically updated or notified of changes.

9. Multiple users and locations. Many users may need access to the environment during the course of a development effort. Furthermore, because one supports global communities of practice, there is a strong need to share this information securely and across disparate locations.

A Development Architecture Needs to Support Customization of the Process.

UML & Case Tools in the Development Architecture

Each project using component-based technology determines how to use OO CASE tools to support an object-oriented methodology and its deliverables. These deliverables range from high-level business process documentation in the business-modeling phase to descriptions of classes in the construction phase. UML compliant CASE tools provide a number of the deliverables that most object methodologies uses, however, there are almost always some deliverables that do not fit in the selected tool. There is also a discrepancy with the CASE tools' ability to comply with UML due to the continuing evolution of UML versions.

UML has become so universal now that deliverables from most methodologies form a superset or, in some cases, a subset of the deliverables described by UML. In the case where a deliverable is a close match to a UML deliverable, proprietary scripting is required to allow for complete semantics. This scripting is also used to migrate from the CASE tool to other drawing or word processing tool. Procedures must also defined to effectively use the tool to support the process.

Decide on Supported Deliverables Early in Process

Case tools in recent years have extended their ability to support more of the life cycle and improved their ease of use. In addition, some case tools have improved their integration with the Integrated Development Environments (IDEs) and produce some level of acceptable component code generation. It is important for the development architecture team to determine early exactly which deliverables may be created in each phase of development, which tool they may be captured in and whether links between phases require upgrading deliverables as a result of the transformations and/or enhancements from other phases.

The team must decide how much they may leverage the automated tools to support the build process. An investment in the macro infrastructure can lead to tremendous time savings as the construction process is supported. The team needs to determine early whether they may "build" their custom process into the tool or adjust the process to better integrate with the tool.

Development Architectures Are Often More Heterogeneous Than Traditional Environments

While traditional client/server systems typically required one development tool for programming efforts, component-based systems are often built using several tools and programming languages. The increase in tools is directly related to the improved capability to integrate software components through interfaces that hide the implementation details.

Typically, the more heterogeneous environments may be built upon the open CORBA technology, while applications developed with JavaBeans or COM may tend to be more homogeneous in nature. Thus, it is important to understand the technologies used as the effort to design a cohesive development architecture may be impacted. Plan to spend more time designing and building the development architecture for a heterogeneous environment.

Configuration Management

The advent of client/server has focused significant attention on the importance of configuration management as key to success. Configuration management is more than just source code control. It must encompass the management of the application software components from conception, through implementation, delivery, and enhancements. While the problem is not unique to component and object development, an object-oriented environment presents special challenges discussed below.

Configuration Management is More Complex in a Component Development Architecture

Currently, artifacts versioned with various tools do not know about each other. For example, an object Versioned in a document management tool has no relationship to the source code configuration. In addition, various tools are advertising the advantages of their repository strategies. However, these products are in their infancy and most do not integrate seamlessly with the source code configuration managers let alone the various tools in a development workbench. Models, source code and documentation are not synchronized. The reality is that current versioning in the majority of tools only occurs at the file level and not at the required level of granularity to support development elements. Methods, classes, components, and their respective deliverables should be versioned but only a few products on the market today support this level of granularity and they are not yet integrated with popular case tools.

Object Systems are Decomposed Into More Pieces

Configuration management is more complex with object development because the system is more finely decomposed. Object development realizes the benefits of flexibility and reusability through a greater level of decomposition than was present in traditional systems. While smaller objects have the advantage of making it easier to have pre-defined building blocks, a disadvantage is that large-scale systems have so many elements that managing their relationships becomes difficult.

For example, a key principle of object-oriented design is separation of concern, which decomposes behavior into smaller, more cohesive objects. This strategy strives to prevent changes from rippling through many objects. The implication of this design approach, however, is that the resulting system may comprise many more modular pieces than a traditional architecture. This greater decomposition complicates configuration management.

Not only are there more objects than procedural modules, the relationships and dependencies intrinsic to object development are usually more complex. For example, the relationship between business processes and objects is a complex, many-to-many mapping: a business process is implemented by more than one object; conversely, an object contributes to more than one business process. FIG. **52** illustrates a business process **5200** to object **5202** mapping to illustrate such relationships between business processes and objects. One manager referred to this phenomenon as the web of interdependencies.

To manage this problem determine early what the "unit" of configuration may be and have the development organization aligned with the approach. For example, in the previous maze possible units of configuration could be:

Process **1**depends on:

Object **1**

Object **2**

Object n

This keeps the process component rigorously configured with its dependent pieces.

Configuration Management Requires a Comprehensive Approach

Most Object CASE Tools do not Support a Complete, Integrated Repository

Integrated tools have, thus far, not been found to support cross-referencing window elements, object model attributes and behaviors, and relational database definitions. Thus, large projects must consider crafting a strategy to integrate multiple point tools to provide such cross-referencing.

The tools gap raises the importance of rigorous procedural and organizational models to address configuration management. For example, proper procedures must ensure that rigorous quality and build steps are followed before introducing a new component into the environment; the workplan requires much more detail to track dependencies; and, the organization structure must effectively support more extensive communications to react to changes.

Adopt a Philosophy for Configuration Management that Guides the Development of the Process

There are two fundamentally different approaches to configuration management in the component world. Simply stated, they represent the difference between an optimistic approach versus a pessimistic approach to managing sources. In the optimistic approach multiple users can access and modify the same sources and the tool is leveraged to resolve conflicts when code is merged. In the pessimistic approach a source is locked when it is checked out. Both advantages have pros and cons and some source control managers allow the configuration to choose which approach they may choose.

Define Multiple Levels of Ownership

A traditional, procedural system usually assigns ownership by business function.

Functional developers take on responsibility for a business function that corresponds to a front-to-back window. Technical team developers then take on cross-function architecture responsibility. This approach has obvious benefits in providing straightforward communication points and division of responsibility. A drawback, however, is that business function reuse is much more difficult.

This approach breaks down due to the higher level of reuse in an object-oriented system. Note that a procedurally designed system may also experience this problem to the extent that the team strives for a large degree of business logic reuse.

Owners Must Exist for Every Versionable Component

An object-oriented system must assign component ownership at multiple levels. Business process owners are still necessary; however, clear lines of responsibility must be assigned for the domain object model. Often these two may have a tight relationship. For example, consider a gas utility customer system that provides customer service orders. The service order business process and service order domain object owner should probably be the same person. However, the service order process may also need to collaborate with other key domain components such as the customer and premise. This requires collaborating and communicating with other developers.

Rigid Ownership Boundaries May Not Work

Experience has shown, however, that the level of communications with core business objects such as customer and bill account is so high that the rigid ownership might be ineffective. The resulting communications of requirements may produce inefficient hand-offs and bottlenecks. For large, mission-critical applications, multiple levels of ownership must then be defined. However, this creates a risk of conflicts. Before components mature, the rules of divisions should probably be more rigid. Later, multiple developers can modify common classes, while keeping responsibility to release, or publish, the code in the hands of a single owner.

Thus, ownership roles may overlap, requiring the rules of engagement to be defined. Yet, every scenario cannot be spelled out precisely. The team and leadership must then be very participatory and flexible to adapt to the dynamic requirements.

One Large Engagement Defined Separate, Overlapping Ownership Responsibility for:

- Windows
- Domain object model sub-systems, or components; the model comprised about 350 model objects which were partitioned into about 12 major areas
- Business processes that were particularly complex, highly reusable, and cut across many windows; for example, writing off a bill
- Common architecture framework components
- Separate concept of ownership from developer for increasing productivity

One solution to the above problem is the clear distinction between component ownership and developer rights. This philosophy is supported by tools like Envy/Developer for Smalltalk and Visual Age for Java. Assign owners of classes, packages, and projects and then assign developers to the packages. Any developer may write methods on an open edition or checked out copy of a class. The owner of the class can release the methods to the class, version the class and release to the general development team. In this model editions are open configuration units, versions are any units that have been checked back in and releases are units that have been made visible to the next construct of configuration management.

In this model clients of lower level components can be added as developers in the integration phase. Rather than have to wait for a new version of the component, they can concurrently have an edition opened with which they can modify or enhance and then submit their changes back to the component owner. This practice can keep control with component owners but increase the bandwidth of the development cycle.

Application Packaging Must Have a Clear Release Management Strategy

To support a flexible ownership model requires a detailed technical packaging strategy. Multiple levels of granularity for controlling source code are typically needed. The method and class are obvious targets for versionable components. However, levels of granularity above the class are critical to properly control the cross-class development that occurs.

Typically development may occur on groups of classes which can be versioned together as categories or applications. In Java, for example, these categories are packages. For example, the frameworks development team may generally have a work-in-process version of the framework architecture package to support new development. Application developers would instead have an older version, typically the first version, that had been thoroughly tested and rolled out.

It may also be necessary to version groups of methods together in a class. This has proved beneficial for managing object model development.

Components of the system should also have a well-defined [EWF1]relationship between them. This should occur at each level of granularity and give a definite feel for the dependencies between components. Each instance of a component also needs to know the specific, tested component versions with which they are compatible. It is essential that the relationships between components are non-cyclical or layered and'that a complete dependency inventory be obtainable for every versionable component.

Favor Shorter Shelf Lives to Support Frequent Roll-outs of Reusable Objects

One of the most difficult decisions for object development is how frequently to roll-out reusable components to multiple developers. And a related issue is how long component should sit on the shelf between changes.

For a traditional, waterfall approach the shelf lives may be quite long with few iterations. For example, a module is typically coded and then put on the shelf until string test. The elapsed time ranges from a few weeks to many months. Likewise, once string tested the module may again sit for a long time until system test. These long shelf lives typically work reasonably well unless the underlying data model or architecture changes. In this case, unproductive re-work results.

The Object Model Must be Continuously Rolled Out to the Team

For object development, roll-outs of objects must occur at varying intervals depending on the range of impact. Because the object model is continually evolving as completed windows incrementally add behavior, the model must be continually refined and rolled out to the team. Some of these changes may have a very narrow impact to just one window, where others may have more global implications.

For example, changes may be rolled out in the following intervals:

- Application Interface or Control—nightly
- Narrow Impact Object Model—nightly
- Wide Impact Object Model—coordinated on-demand, no more than 1–3 times per week
- Frameworks—weekly or less frequent depending on impact, maturity of the component, and the complexity of the effort

Structural Object Model—for Data Waves, Once Per Month

Object development also requires shrinking the shelf lives dramatically. Reusable domain model and framework objects generally require a zero tolerance policy for incorrect code. Problems need to be fixed immediately, or at least their impact critically assessed and the fix scheduled. As mentioned earlier, some of this immediacy can be managed with careful process surrounding ownership, editions and developers. In one tool there is a concept of a scratch edition that allows a non-registered developer to access units of control and make changes within his private environment and still be able to post the changes back to the component developers and owner. This eliminates hours or day turn around to correct a critical problem in a versioned component.

Clean-up or fix-it Days Must be Scheduled

Window or view specific behavior can have a longer shelf life, but still not as long as traditional development. Letting items sit for more than even just a few weeks can cause them to become dangerously out of date. Thus, two different large engagements found it necessary to schedule clean-up fix days on a regular basis.

Regression Testing is Key to Effective Configuration Management

Regression testing is essential to support these more frequent clean-up efforts. This approach frustrated management, because these days appeared to be a step backward or treading water. However, keeping the application clean paid dividends in addressing and fixing problems more efficiently. Generally speaking, the longer the problem went unfixed, the more expensive they were to correct.

In summary, a flexible approach is necessary to coordinate and control changes. Some considerations include:

Ability to absorb change—If the developers are overwhelmed with change or learning curve, the shelf life must be expanded to reduce change.

Magnitude of the change—Minor changes may be easy to incorporate and may facilitate immediate turn around. Major changes may be expensive to incorporate except at controlled, regular intervals.

Restart Cost—Each effort to integrate changes into an existing component may incur a start up cost for the developer. This may again be influenced by the magnitude of the change, and the duration of the integration cycle. A rapid integration cycle may keep the behaviors fresher in the developer's memory; a longer shelf life may involve a refamiliarization cost. On the other hand, this must be balanced against the cost of starting and stopping new development to implement fixes.

Stability—As a component stabilizes and matures, the shelf life can be reduced without impacting the rest of the project. Unstable object components cannot be rolled out as frequently, because the turn-around time is longer.

Delivery Capability—The ability of the migration team to provide a "most current" build may also impact the fix versus shelf decision. In C++, the build process may be a major undertaking, where the shortest shelf life may be measured in days. In Smalltalk, the size of the image may likely have a similar affect. In Java the adherence to clearly defined packages improves the delivery capability.

Configuration Management may Require 5–10% of Development Team

Configuration management clearly requires more effort with object development. These roles are often hard to justify to management, because they appear to be pure overhead. The tasks may also appear unclear. For example, tasks such as "Manage environment" and "Communicate changes" do not to have a start and a finish.

These tasks should be controlled and managed by a centralized effort. Several people sharing the effort in their spare time may not exercise enough caution and due diligence. Furthermore, a centralized effort may often result in automation of tasks producing significant productivity improvements.

At least 5% of the development team should be completely dedicated to the on-going configuration management effort. When setting up and defining the environment even more resources may be necessary. Of course, there are limits. Stacking the team with too many resources may result in wasteful development of an overly elaborate tools architecture.

Another approach is to make the configuration process implicit in the entire development process. In other words, by ensuring that an owner of a class must version and release his work before it can be seen by a containing package the owner is required by the process to be thinking about the configuration process in all of his work. Subsequently, the package owner, generally a more experienced developer, must ensure that all classes are versioned within his package, version his package and then release it for general consumption. This would work the same for a project which tends to be centered around increasing units of capabilities (i.e. business activities and finally whole applications).

Scaling to Large Teams

Despite the advice to use small teams, enterprise applications are large and often require in the aggregate a large number of developers. Development architectures must be constructed in such a way as to support sometimes hundreds of users with many, sometimes hundreds of thousands of development artifacts and their relationships with each other.

All of the major software development tool providers (i.e. IBM, Microsoft, Oracle, Unisys, etc) have announced repository strategies. These repository strategies are much more comprehensive then the proprietary repositories that are represented by a source tool repository such as in Clear Case for source code or the proprietary repositories shipped with Case Tools or development environments like Envy Developer and Forte. These repositories allow for information to span tools and strive for integration between not only tools provided by a single vendor or from a host of third parties as well. Many case tool and IDE providers have announced support for this new generation of component repositories.

The new strategies all espouse either de facto standards (Microsoft's Open Information Model) or eventual conformance to a repository strategy (OMG's Meta Object Facility—MOF). These repositories, although encouraging, are very immature and may require a few years to deliver on their promises. In the mean time development architectures must decide on their own how they may provide the necessary facilities to promote large team development progress.

Query & Impact Analysis

Tools are necessary to identify categories of similar behavior such as the class hierarchy, where used, senders of, implementors of, etc. Today, many environments for C++, Smalltalk, Visual Basic & Java provide robust browsers with this comprehensive functionality. Additionally case tools also provide search capabilities. Unfortunately every tool uses a different method for finding artifacts, such as text searches for documents, menu provided searches in case tools, and where used and senders of within browsers.

As mentioned in an earlier section many of the language based IDEs provide sophisticated browsers and explorers that allow for searches for "where used" and "senders of" for messages and objects. These facilities are extremely important in component leveraged architectures. They allow developers to more effectively look for things to reuse rather than always re-inventing what they need. One important practice to help the searching process is naming standards. They should be put in place early in the process to enable a principle ParcPlace was fond of calling, "the principle of least astonishment". Because of polymorphism developers become very agile in locating classes and methods because their interfaces are so common like all objects responding to the "toString( )" method.

One of the problems in current development architectures is the redundancy of the facilities. For example, rather than be able to rely on the repository where the information should be stored in a common location developers may search in Rational Rose and in the source code manager for references of a given type.

One way to mitigate this issue is to publish information to a common location to make it accessible to everyone through a common interface, preferably a web browser. Tools like. JavaDoc and Microsoft Word (which can transform documents into HTML) make it possible to leverage the web server's index server to locate artifacts from various locations. This practice is being more widely adopted, as shown by the release of IBM's JCentral tool.

Asset Catalog (Reuse)

One key improvement in component-based development from traditional development is the use of components to assemble solutions. This is very different from libraries. Because of the reflective nature of components, runtime binaries can be dropped into the development environment, their interfaces exposed and then integrated into the current solution space. This is done through the Java Reflection mechanisms within class files and type libraries in the Active/X world.

Currently reuse tends to be at entire source code branches rather than component-oriented. This has been provoked by poor version support in most development environments and tools inadequate for managing the assembly and configuration of components into solutions. Some component manager tools that are being released onto the market today support either the ActiveX or JavaBean component models but its not clear how they may be received, used and integrated into development and design environments.

To maximize reuse requires the assembly and configuration of run-time components in addition to being able to construct new components as part of the software construction process. A new breed of tools supporting black box reuse referred to as "Component managers" should be considered one of the primary tools provided with the environment to 1) support transformations between tools where this may continued to be a requirement, 2) enable component views of reuse allowing configuration from both run-time and development components and 3) give component developers security features preventing users from modifying and/or reusing certain components if they desire. It requires the ability to categorize components and search components according to property descriptions in a way that can be ascertained without the viewing of source code.

Code Generation

In the past code generation was crude, had to be customized, and was hard to keep synchronized once source code was emitted. This awkwardness was caused by other related factors like the lack of a common information model, little coupling with the IDE and no common repository sources. In addition, the ability of the CASE tool environment to comprehend the run time environment was poorly supported in most tool environments. The most damaging problem is the failure of CASE tool providers to "own" the code integration and generation produced from the model. Some of the efforts to integrate with IDE's via Add-Ins are a step in the right direction but, some key issues, such as identity integrity across multiple environments, have not yet been addressed to ensure its success.

That being said, code generation via case tools at the structural level can greatly increase the productivity of a team when a rigorous model is adhered to in mapping the domain model constructs to code or schema in the target environment. Two areas have been used to some degree of success from component engagements 1) generation of DDL from object schema—the domain model and 2) generation of the object structure or domain model to the target language.

One analogy has been made with Layout Managers or Screen Builders. A decade ago people were comfortable with coding windows by hand. Some even felt that form designers were too constraining and got in the way of developing a really usable interface. However, no one today would think of generating forms by coding them by hand. So with the standardization of UML and the maturing of object model semantics developers should be reticent to code class structures by hand. Oracle refers to this as "one source of truth". A change to the class structure in the source code is a change to the model and vice versa.

Desktop Tool Integration

Desktop tools today generally include an office suite, drawing tools, case tools and more recently, a web browser. For example, one might find a tool selection like Microsoft Office for documentation, Visio for custom deliverables, Rational Rose for models, and Internet Explorer for viewing HTML versions of the documentation. VBA has become the glue for extending and connecting the information between these tools. Other strategies have included using Notes as a repository for all of the deliverables that users could access information.

ODM has many predefined deliverable templates that are targeted towards this suite of tools including Word, Excel and Visio templates. Often times management underestimates the start up cost of integrating the tools in such a way as to improve the flow of information between phases and for ensuring that information is published to the team in a way that is accessible and plentiful. However project experience teaches that this investment can yield many returns down the road if the development architecture includes processes and infrastructure to support this flow of information.

This desktop tool integration strategy needs to take into account the comprehensive approach used by the configuration management strategies. In other words, relevant documents need to be associated with the components and business processes they update so that key stakeholders can subscribe to alarms that may make them aware of when sections of documentation need updating. This process may help ensure that the publishing model is dynamic and current.

Many Users and Multiple Locations

Solution Centers and engagements often have many users and multiple locations involved in solution delivery. It is very important for development architecture teams to solve the problems of concurrency within tools and ownership across locations. Strategies need to be developed for how

components may be exported and imported, and supported across locations. In addition, an approach for receiving feedback for improvements needs to be established. Most projects have found that ownership is even more important in a distributed development environment. This allows for the using of master/slave assignments on components and dictating either who is allowed to make changes to the component or who is responsible for merging changes. As one technologist from Sun stated, if distributed development is not managed carefully it can be like herding cats.

Summary

Although there are new challenges with development architecture in a component environment there are also additional opportunities for increased productivity. A team that understands the additional considerations may weigh the opportunities that tool integration can bring to the project against the practical gap in the market place and customize their development architecture accordingly. Wise planning and a clear understanding of the strengths and limitations of the tools available to a team may contribute greatly to the success or failure of a project.

Managing Performance

Component-based technology is often sold on benefits such as reduced maintenance, increased reuse, or flexibility to absorb change. Performance, on the other hand, is usually viewed as a significant drawback. However, resilience to change and performance do not have to be mutually exclusive.

Component technology can enable better performance

Component-based systems have advantages that can actually enable better performance—but only if proper design techniques are used. This chapter discusses the correlation between performance-tunability and a well-designed component-based system, and the implications this has for project management.

The timing of when to address performance may initially appear trivial. "Design performance in from the start" is one often-repeated rule. The opposing viewpoint is expressed by computer scientist David Knuth who said, "Premature performance tuning is the root of all evil". Timing when to address performance is actually a complicated management issue. The competing forces and their possible resolution are discussed further below.

Define performance goals in terms of the business

An old saying goes, "Cheap, fast and good—I'll give you two out of three". Many of clients may react negatively to this philosophy, because they would certainly like excellence in all three areas. Yet, the fact remains that difficult tradeoffs exist between performance, quality, and the cost of the system. For example, no one intentionally designs a slow system. Thus, it is critical to define performance goals in business terms based on cost/benefit analysis.

Consider service level agreements for online performance, which are often based on the average wait time between screens. This makes sense in a technical environment using 3270 display devices. However, this may lead to poor business decisions for a non-modal, windowing GUI.

A GUI may support a more rich set of processing than a 3270-based design. This can result in response times much slower per window; however, the time for completing the business transaction such as a customer order may be equivalent or even less. Yet, to tune the performance for an equivalent level of window-to-window response time may simply not make economic sense. Thus, the requirements should be based on how efficiently the system completes the pure business event, encompassing potentially multiple windows, rather than a more technical measure of window-to-window navigation.

Measure Performance

Any effort to effectively address performance requires thorough measurement capabilities. There are two reasons for this. First, the team must understand where the specific risks reside, before they can effectively attack them. Is the application I/O or compute bound? Is database or network I/O a bigger issue? Are there obvious bottlenecks? These are all key questions.

Performance metrics focuses attention and provides confidence

Second, just the simple act of measuring and tracking performance focuses attention in a positive way. Tools such as language profilers and memory-leak checkers are critical. A rich set of tools can give the team more confidence in the quality of their development and technology.

Confidence is particularly important to object-oriented and component-based systems development, because a delicate balance is necessary between addressing performance risks without detracting from good object-oriented design. For example, fear of messaging overhead may lead developers to avoid altogether factoring behavior into smaller methods and objects. Yet, such factoring is critical to application reusability and quality.

Fear of object messaging overhead can be overstated

A potential source of misunderstanding is equating object messages with network or operating system messages. Actually, object message sends are often more comparable to function calls, albeit slower. And the overhead of message sends compared to function calls can be unimportant compared to the application I/O. That is, most applications are I/O bound, not compute bound. On the other hand, it is important to understand the frequency of component messaging since it may cross network or process boundaries. Thus, when looking at messaging characteristics it is important to distinguish between component messaging and object-messaging.

Address architecture performance risks early

As with a traditional client/server system, performance risks should be addressed early. Performance requirements often have a severe impact on the technology architecture including the infrastructure design and the platform systems software and hardware. For example, the architecture team may need to decide whether to use messaging, remote procedure calls, or shipped SQL statements for distribution services between client and server. Performance may also impact fundamental platform decisions such as the choice of language, DBMS vendor, operating system, network, or hardware configuration.

Usually these parameters cannot truly be understood without constructing a benchmark prototype. In cases where the underlying platform is affected, the benchmark should be planned and conducted at the outset of the project. These measures are important, because intuition may often be wrong as to where the problems lie.

In addition, the technologies that make up the foundation of a component architecture may be new and unproven. To minimize risks, look for a reference application that is similar in complexity and size. If a similar application can't be found, then it may be necessary to develop a proof-of-concept prototype for the architecture. Such a prototype may address areas such as the middleware, application architecture, or hardware platforms.

SAP - Exhibit 1105
SAP v. Valtrus et al. - IPR2025-00420
Page 217 of 281

Performance is balanced against encapsulation and software distribution

Performance Is Frequently Balanced Against Encapsulation and Software Distribution

As with any system, there are design trade offs that can be made to achieve better performance. With component-based systems, some of the most significant performance trade offs are made against encapsulation and software distribution.

The encapsulation of data forces applications to access data through a component's interface. Unfortunately, encapsulation may many times result in excessive messaging, sometimes across a network, between components. Thus, performance can often be improved by breaking encapsulation to directly access data.

Software distribution is often simplified by utilizing centralized application servers. However, a centralized approach may result in diminished performance due to the network messaging. Performance can often be improved by distributing software closer to the point of usage.

Selecting the right balance between performance, software distribution, and encapsulation is not easy. Achieving the right balance may be driven by system's requirements. Performance tuning can be deferred with object-oriented frameworks

Object-oriented frameworks enable performance tuning to be deferred

Smalltalk columnist and consultant Kent Beck espouses the philosophy "Make it run. Make it right. Make it fast." At a glance, this advice may seem counter to the previous recommendation to address performance risks early. However, they do not have to be mutually exclusive. An application should be prototyped—i.e., made to run, early to address broad architecture performance risks. Later, proper design should be the focus before performance, because a well-designed application enables more productive performance tuning. Optimized code is simply very difficult to maintain. And prematurely optimizing code may incorrectly assume what problems are most important, thus wasting effort.

Object-oriented system development, in particular, allows for a deferred attention to performance. The component design goal of encapsulating implementation details tends to lessen the impact of major change to the application. This allows sweeping changes to be made late in application development. FIG. **53** is a diagram which illustrates a graph **5300** of resilience to change.

This graph illustrates the belief that through a good object-oriented design, changes related to performance tuning may be made much later in the development lifecycle than would generally be possible with traditional structured design. With an emphasis on good object-oriented design, the degree of radical change possible late in development is surprisingly high.

Non-object-oriented systems should be performance tuned throughout development

When components are not built using object-oriented frameworks, it may not be as feasible to defer performance tuning. Without frameworks that provide a well-layered and factored architecture, it may not be possible to make small, localized changes that result in dramatic performance improvements. Instead, it is better to performance tune as the system is being built so that there is time to make changes. Furthermore, it becomes even more important to establish design guidelines early in the project so that detailed designs can be reviewed against them. This can help ensure that performance problems are avoided before components are implemented.

Leverage points

The value of reuse is frequently perceived as "less to code". While often true, a sometimes overlooked, more valuable aspect is "less to maintain". This is notably significant when performance-tuning a system. It is generally worthwhile to spend more time upfront determining how to reuse existing components than it is to spend less time developing a new solution. Similarly, it is usually more worthwhile spending more time generalizing a component so it may be reused than it is spending time to develop a specialized solution.

A leverage point is factored out behavior that enables leveraging global performance gains

A leverage point is processing common to multiple components which may be factored out and reused when needed. In performance tuning, these points are identified, profiled and tuned, thereby leveraging any performance gains against all components which use them. In general, the less actual processing an application-specific component (i.e. non-architecture) performs indicates the more performance leverage may be gained from it by tuning architecture processing.

For example, a business event controller class in a system must somehow specify the relationships between its relevant business components and the widgets which may interact with them on the application window. There are two fundamental approaches in specifying these relationships. The first is for an initialization method to be invoked in the controller which may perform the processing required to define these relationships. The other is for that business class to specify the bare minimum information required to infer these relationships such that a common architectural component can perform the actual processing required to define the relationships during runtime.

The latter approach provides a leverage point for performance tuning the initialization of the window. The processing may be tuned to use a more efficient algorithm; the results of the initialization may be cached during application packaging and read during initialization; or, efficient initialization methods may be generated and maintained automatically from the information by a code generator once it becomes clear what the most efficient implementation is. In any case, the flexibility provided by this leverage point allows many more possibilities to be considered during performance tuning. Note that all three optimizations could be achieved without manually visiting a single of perhaps hundreds of windows which share the initialization processing.

The pursuit of leverage points must be considered in every architectural design decision, and followed with discipline in application design.

Communication via interface definitions—specify what not how

On a component-based project where the development should reuse extensively, the name of a component and its methods are perhaps the strongest medium of communication between the original developer and a developer interfacing with, or maintaining that component. A fundamental grammatical naming standard is the means to ensure clear communication between developers. This standard must be well-defined, strongly enforced, and supported by leadership.

A weak standard of interface definition often results in code requiring extra processing which could be avoided by making assumptions based on a strict interface definition. Performance tuning is easily complicated by generic interfaces supported by vague assumptions. Redefining such interfaces late in development is often prohibitively expensive relative to the low cost of clear initial interface definition.

An example of a poorly-defined interface is a method definition which may accept several unrelated types as its parameters. The result leads to type-checking of parameters and decreased flexibility in tuning the implementation of the method. The strong definition of interface parameters allows fundamental assumptions to be made in tuning the implementation of the interface.

A grammatically-based naming standard differentiates between methods that do versus methods that get. In a traditional approach, procedures or functions do routines and algorithms. The unique blend of data and behavior in component-based development, on the other hand, allows components to collaborate, asking each other for data, as well as directing each other to perform processing. This requires the addition of nouns and the inclusion of verbs to the vocabulary of interface definition. That is, the interface should specify what, not how.

For example, if a customer component provides a public interface that allows another component to ask it to query the database for its credit profile, a common mistake is to define a method getCreditProfile or retrieveCreditProfile in customer. If, however, performance tuning required caching the customer may already have the credit profile. This would leave development with the choice of either changing the method name in all referencing components, or create documentation to explain why the method getCreditProfile didn't really get anything, but just provided access to another component.

This example illustrates the importance of naming to ensure encapsulation. The implementation changes required to achieve radical performance improvements are feasible only through diligence in the pursuit of encapsulating implementation.

Along with good design organization, clear interface definition is key in achieving valuable tunability.

Limit knowledge of data and object relationships

Developers with structured programming experience often tend to perceive objects as data, manipulating them within the context of objects, effectively distributing behavior associated within components amongst all the objects which interact with it. This becomes very difficult to performance tune due to the combination of duplication of code, and the wide impact any such tuning could have on application classes. A much greater degree of performance tuning can be achieved when object responsibilities are respected and objects or collaborations of objects can be tuned in isolation with minimal impact to their embedded system.

A simple example of "data-ifying" objects is when object A manages a group of other objects, yet other objects ask object A for its managed objects and manipulate them freely. Generally loop iterations are prime candidates for significant performance improvements. If the iterations are distributed over every object that interacts with object A, little performance improvement of component A may be gained without high impact. By restricting interfaces such that only object A may iterate over its own managed objects, the iteration code can be tuned with little impact outside object A.

Performance improvements can always be identified. The difficulty is in the cost of actually implementing them. The strong pursuit of encapsulation allows bottlenecks to be identified more easily (i.e. in one place), and tuned with minimal impact.

Leverage functional and technical tuning

Though tuning of a component-based application can be deferred until late in development, eventually it must be done. At this point it is important to realize the difference between functional tuning and technical tuning.

Functional tuning involves a combination of cognitive and measured tuning. It consists of looking at the functional design of a component and determining which portions of processing can be deferred, cached, etc. It demands a developer who is functionally knowledgeable about the desired behavior, whether it be architecture or application. It often results in reorganizing or redesigning portions of code. The performance gains realized during functional tuning are generally the most significant gains.

Technical tuning is a lower-level approach to tuning, developing more efficient techniques to achieve the same functionality. Technical tuning demands a developer who, though not necessarily intimate with the functional requirements, has a strong familiarity with tricks and techniques of the development platform. It can involve better use of memory, language idioms, base class modifications, etc. Technical tuning should require little or no changes to application code, and narrow changes to architecture.

Opportunities for performance tuning are found both in bottlenecks and in distributed inefficiencies. There are generally many tools available in detecting bottlenecks. Distributed inefficiencies are usually more difficult to identify with tools. Whether performance optimizations are realized through cognitive analysis, or tool-assisted profiling, it is important to measure the gains against a baseline performance level.

Few performance improvements are gained by eliminating completely useless code. Gains are usually achieved by trading speed for size, or chronologically reorganizing processing. Improvements in one area may weigh in a different area. For example, runtime processing is often sped up by increasing initialization time. When making such changes, measuring the affected runtime processing is insufficient. It is necessary to measure also the areas impacted to determine that the optimization does not push another area into unacceptable response.

Summary

Performance is an acknowledged risk in developing complex systems with today's maturing component technologies. To reduce risk and uncertainty, it may be necessary to develop" prototypes that validate the architecture approach. When components are built using rich object-oriented frameworks, it is possible to tune a component-based system more effectively and later in development, than its structured counterpart. Other more traditional approaches to components, may require tuning throughout the development cycle.

BASE SERVICES (**1020**)

Batch processing is used to perform large scale repetitive processing where no user involvement is required. Batch support is an often overlooked area in architecture and component design. When first client/server and then component technology hit the scene, the emphasis on GUI and communications was so strong that many thought of batch as dead. Today, one is wiser about including batch in the scope of both architecture and application efforts. One also finds that many of the principles and concepts that applied to batch twenty years ago also apply today.

In general, batch still has the following fundamental characteristics:

Scheduling—Services are required to manage the flow of processing within and between batch jobs, the interdependencies of applications and resources as well as to provide integration with checkpointing facilities.

Restart/Recovery—Batch jobs must be designed with restartability in mind. This implies the need for a batch restart/recovery architecture used to automatically recover and re-start batch programs if they should fail during execution.

Controls—Run-to-run and integrity controls are still required to ensure that all data is processed completely.

Reporting—Services are required to handle configurable report creation, distribution, printing and archiving.

These services are typically not provided through component technologies. They can be provided by third-party products or custom implementations.

How is batch different in a component-based system?

A system's batch architecture can be easily overlooked since it is not a part of the system that is visible to end-users. Regardless, it is critical to design components with both on-line and batch requirements in mind. Combining both sets of requirements is necessary to ensure that your components can be used in both environments. This will allow the batch programs to act as just another client to your components.

In addition, since many on-line systems are expected to be available on a 24×7 basis, there may be a limited window available for exclusive batch processing. This requirement can have a tremendous impact on your batch architecture. In these environments, it is necessary to design batch programs that make efficient use of resources and have a low impact on on-line users.

A component-based batch architecture must support batch programs that read transactions that are really messages. These message transactions are read either from a flat file or from a database. The program must then locate the component for which the message is intended and pass the message to that component for processing. In many cases, these will be the same components that process messages from on-line (GUI) applications. The function of the batch "program" in this environment is fairly limited. It reads the input messages, controls the packaging of database units of work, and sends requests to the business component that performs the actual business logic associated with the messages. Batch architectures usually "commit" on intervals that are designed to optimize database resources. Thus, it is important to design components that can participate in a logical unit of work that is controlled outside of the components.

How do the patterns in this section help?

The patterns described in this section represent some initial attempts to capture basic concepts that are useful in the design of a component-based batch architecture. They are by no means exhaustive but represent building blocks in a complete solution.

The Batch Job pattern describes a method of structuring batch components so that common architectural services are implemented uniformly across all of these components. In a way, this is the component-based analog to the concept of shell designs and skeleton programs which have been a recurring feature of robust batch architectures for many years.

The Batch Unit of Work (BUW) pattern, on the other hand, represents a method of structuring the work to be processed by the batch components so that it too can be treated uniformly by all components. An abstraction such as this forms the basis for distributing batch workloads in a number of useful ways. It also enhances the capability of the architecture to support evolutionary change.

The Processing Pipeline pattern describes a way of structuring batch activities so that they can be easily reconfigured as processing requirements change. This pattern directly addresses the issues of scalability and reuse in a component-based batch system.

The Abstraction Factory pattern has a much broader applicability than just batch systems. It represents a way to encapsulate diversity such that only those parts of the system that need to understand the difference between two objects have to deal with those differences. To use a typical batch example, a file is a file is a file. Only those components that require knowledge of the contents of a file should need to deal with those contents in other than a very generic way.

What are some other considerations in developing a component-based batch architecture?

Because batch processing executes on a server and requires limited user interaction, many of the services used for on-line architectures are not needed. For example, the services used for distributing components—naming, distributed events, bridging, trader, etc—are not needed for a batch architecture. In fact, the interfaces that encapsulate components and provide location transparency can add significant overhead to a batch architecture. To avoid the expense of unneeded services, the component stubs can be wrapped with a layer of indirection that short-circuits the normal distribution mechanisms. This will provide performance that will approximate local function calls.

Typically, business objects have to be instantiated from a relational database (RDBMS) before the batch application can make use of them. This extra overhead is a very real concern. It is an unfortunate fact that in many ways the more "object-oriented" your design is, the worse it fits into the relational paradigm of rows and tables. For one thing, these designs tend to have lots of objects with embedded instances or references to other objects. And the primary reason that such designs have RDBMS performance problems is that in the database, resolving such an object relationship requires joins or recursive queries. When mapping from your object model to the RDBMS, there is a tendency to "normalize" your object over many tables, and the performance can easily plummet.

Is efficient component-based batch hopeless? No. But if you have stringent batch performance requirements, you may need some specialized design. There are several techniques you can use to improve your batch speed.

Reduce (eliminate, if you can) batch. This may sound simple and stupid, but is often overlooked and is by far the cheapest way to improve your batch yield. Lots of reports can be obtained on-line, lots of them are not useful or used, "trigger transactions" can simply become spawned sub-processes that run in the background, same for printing bills, the only thing that must be done in batch is a database reconciliation, which requires a time window with no other activity. If you can engage in discussions for eliminating batch, by all means do.

Pool (recycle) objects. Each time you dispose of an object, instead of destroying it put it in a pool of recyclable objects; and every time you create a new object, look in the pool to see if there is one that can be recycled. Keep separate pools for each class of objects. Allocating objects is a lot more expensive than one tends to think, and recycling can improve your batch performance dramatically without affecting your design.

Cache and sort. This technique is well known to "traditional" batch designers, but it is so obvious that they don't even think of it. However, it has a correspondent object implementation. Keep a small cache of objects you have just read from the database. Most of the times,

one instance of each is plenty. Whenever you need to access an object on the database, look to see if it is already in the cache. If not, read it and put it in the cache too. Encapsulate all this logic in a technical "Table" object—not in the business objects. At the same time, organize the processing of your data in a sequence that maximizes cache hits. Again, this technique does not affect your "business objects" design. The processing cost of this technique is so low that you can keep it enabled also for on-line, thus simplifying your technology architecture.

For some applications, an LRU caching policy might not be the right choice; a more complicated scheme with multiple cache levels might be necessary. For this reason it would be best to make the caching policy itself be an object (consider the Strategy pattern for making an object from an algorithm) so you can change the policy on demand.

Cache operations and accesses. One of the reasons component-based batch performs so poorly is due to the fact that, in order to maximize modularity and preserve encapsulation, a lot of operations are performed redundantly. For instance if a balance is implemented as a calculation, and if it is needed by six different objects it is recomputed six times. These situations are very easy to identify with a performance monitor that tells you where the program spends most of its time; it is not uncommon to find that most of the time is actually spent in very few methods. For these methods (and only for them!) cache the result in an instance variable. Every time the method is invoked, check if the instance variable contains an answer: if not, compute it and store it there; if yes, just return it. Of course, each operation on the object that invalidates the result of the computation must invalidate the cache too! This technique has a very small impact on your object design and typically leaves the interface unchanged.

Cache objects. Typically, this would involve leaving recently referenced objects instantiated in memory for some length of time after their last use. Then, if the object is accessed again, you check the memory-resident cache before re-loading the object from the DBMS. Usually you would construct this cache as a hash table keyed by object ID, and use a LRU policy to keep the cache size manageable. Expect degraded performance if you do anything to destroy the utility of the cache. For some applications, LRU might not be the right choice; a more complicated 'scheme with multiple cache levels might be necessary. For this reason it would be best to make the caching policy itself be an object (consider the Strategy pattern for making an object from an algorithm) so you can change the policy on demand.

Make use of " lazy" or "deferred" loading. That is, don't do a "deep" instantiation until you know you're going to use the associated parts of the object. Instead, load selected sub-objects only when first referenced This can save on memory overhead as well as DBMS access. In some cases you can use a hybrid strategy: do a "shallow" instantiation by default, but provide the client program with a way to build the complete object on demand to provide more deterministic performance. One thing to be careful of with this approach is that if you really do tend to use most parts of the object during high-volume processing, loading it in piecemeal can actually worsen the performance, because of the overhead of maintaining the load state and because of the smaller DBMS transactions sizes. These techniques have a very small impact on your object model.

De-normalize your database where possible. Typically when one does object-to-relational mappings, one tends to make every unique object type a separate table. This is best from a design perspective. But in cases where you know you have a fixed set of "private" associations (meaning physical aggregation with no possibility of shared references), then fold that sub-object data into the enclosing object's RDBMS table. It's not pretty, but it can save lots of extra loading time. Also, look at ways to do aggregate loads based on some unique object ID. For example, if you have collection-valued sub-components, insert the object ID of the enclosing object in the sub-object tables and do aggregate loads in code, rather than doing a "point-of-use" instantiation for each one separately. Of course, these optimizations can have a more substantial impact on your object model.

Consider making "light" versions of some of your objects. That is, for performance critical situations, create alternate implementations of your business objects that don't have all the baggage of the first-class objects. Yes, this can be ugly and more difficult to maintain. But for many batch processing applications you might find that you can drop a lot of the (persistence-related) complexity of an object without affecting the batch processing at all. Then create fast hand-tuned routines to instantiate the "light" objects from the database.

As can be seen, there are a lot of opportunities for improvement in component-based batch performance. However, in order to manage risk early, remember that the areas in which you will have trouble are those in which batch excels (predictability, repetitiveness) and component-based design trades off performance for flexibility and encapsulation. Message overhead and similar language related issues are unlikely to be critical. Obviously, before doing any of these things you should do some serious benchmarking to see where you're coming up short on performance. Often the overhead comes from surprising places. Don't twist your object model all out of shape without first having some solid performance measurements.

Abstraction Factory

FIG. **54** illustrates a flowchart for a method **5400** for providing an abstraction factory pattern. Data is received and transformed into a plurality of concrete objects in operations **5402** and **5404**. Each of the concrete objects is associated with an abstract interface in operation **5406**. A map of the association between the concrete objects and the abstract interface is created in operation **5408**. In operation **5410**, when request is received which includes an identifier for one of the concrete objects and an identifier for the abstract interface. The map is consulted to locate the concrete object that has been identified in operation **5412**. An abstract object is then created that corresponds to the located concrete object in operation **5414**.

The identifiers may be included with a single request. In another aspect of the present invention, the abstraction factory pattern may be written in a C++ programming language. As an option, the located concrete object may also be inserted into the abstract object. With this option, the abstract object may operate as a handle.

It is desirable to separate concerns between architecture/framework and implementation details. One way to do this is to exploit the power of polymorphism, using an abstract interface to a concrete object which implements that interface. How, then, is one to create these concrete instances and manipulate them within a framework while preserving the framework's independence?

In any complex information processing system, there will be a variety of different types of information, with a corresponding variety of actions which must be taken to process that information. One of the difficulties in this task involves taking an information source and creating an appropriate internal representation for it. The typical approach to this problem takes the form of a large switch/case statement, where each case deals with one of the information types. The switch/case approach leads to components that are very difficult to maintain, extend, debug, etc. and also leads to a procedural programming style. This approach also makes it extremely difficult to properly manage dependencies so that the details depend on the framework and not vice-versa.

With this in mind, some alternative approach must be used which will allow a framework to handle multiple information types in a way which encourages good style, modularity, extensibility, and framework independence.

Therefore, one transforms the various types of raw data into a corresponding variety of concrete object types, all of which share a common abstract interface. This transformation will be encapsulated within an Abstraction Factory.

The primary interface to the Abstraction Factory is:

"abstractType produceForKey(key)"

where "abstractType" is the type of the common abstract interface, and key is a piece of information which identifies the appropriate concrete type. (This could be the same piece of information used in the switch/case statement; there could be a variety of ways to get it). When this method is invoked, the Abstraction Factory consults its internal mapping and creates an "empty" object of the proper concrete class. The factory then casts the concrete object into the abstraction and returns it to the method's client. This client (a framework most likely) will then instruct the abstraction to initialize itself from the incoming data stream.

At the end of this process we have an abstract handle to a concrete object which a framework may then manipulate generically.

Benefits

Software Quality. Exploiting this pattern can allow us to avoid one of the major pitfalls of procedural programming, the switch/case statement. Done properly you get better modularity, testability, maintenance and extensibility.

Frameworks. The layer of abstraction introduced allows us to build frameworks which follow the open/closed principle, that is to say, they are open to extension by the addition of new concrete types, but are closed to the necessity of risky and costly modification

Implementations of this pattern will vary widely depending on the selection of language. For example, in C++ a generic factory, based on templates can be constructed, and key—concrete type pairs can be registered to the appropriate instantiation of the class. This might require manual coding in other languages. The key interfaces, however are:

Abstraction Factory:

Abstract Type produce For Key(key)

Abstract Type:

init(some data stream)

The Abstraction Factory can be fully coded in C++. It is very re-usable as it stands. In addition, it has been extended to perform "Java Loader-like" dynamic linking if the proper code cannot be found already within the factory.

Factory, the well know pattern from Gamma, et. al

BUW, in which the objects created by the factory can be dealt with generically in terms of independence, scalability, parallel processing, etc. Component Solutions Handbook.

Batch Job

FIG. **55** illustrates a flowchart for a method **5500** for representing a plurality of batch jobs of a system.each with a unique class. In operations **5502** and **5504**, an abstract class of abstract data required by a plurality of batch jobs is provided and a plurality of batch job sub-classes are defined. Each batch job sub-class includes batch specific data, and logic for processing the abstract data and the batch specific data upon the execution thereof. Each of the batch job sub-classes is represented with an object in operation **5506**. In operataions **5508** and **5510**, one of the objects is identified and the logic of the batch job sub-classes associated with the identified object is thereby executed.

In one aspect, the data may include a name, a current status, messages encountered during a run, various times, and a priority. In another aspect, the abstract class may include default logic for running a batch job.

In an additional aspect, the abstract data and the batch specific data may be stored separately. In a fourth aspect, the logic of the batch job sub-classes may be executed by a scheduler.

A set of logical operations may need to be initiated through some "batch" scheduling means. This requires a set of common services such as activation, logging, and error handling that will have to be applied across all jobs. How can these common services be distributed to all types of batch jobs?

Most business systems today include some sort of batch processing. Batch processing is the execution of a series of instructions that do not require any interaction with a user to complete. Batch jobs are usually stored up during the day and executed during evening hours when the system load is typically lower.

Once a batch job begins, it continues until it is complete or it encounters an error.

An architecture that supports batch jobs usually has certain characteristics. It must be able to support checkpoints and rollback, restart and recovery, error handling, logging, scheduling, and resource locking

Most systems, including those that are component-based, require this sort of architecture. A difficulty arises when considering component-based systems though.

In a component-based system, the application architecture is usually very separated from the business application classes. In many cases, the business classes and components are built without regard to the surrounding architecture.

It is expected that the business components will execute in some environmental container that will provide many of the architectural services (like batch services).

Some natural representation of the batch architecture must be developed and transparently integrate with the existing business components and still support all of the architectural requirements.

Therefore, represent each type of batch job in the system as its own class. An abstract class (BatchJob) will exist from which all specific types of batch jobs will derive from. The abstract BatchJob contains data that all batch jobs require: name, current status (pending, started, finished, deleted), messages encountered during its run, various times (submission, start, completion), and a priority, for example. It also should provide some default behaviors including running the job and logic to execute before and after the run.

SAP - Exhibit 1105
SAP v. Valtrus et al. - IPR2025-00420
Page 222 of 281

FIG. **56** illustrates a class diagram of the batch job hierarchy.

Various system batch job classes can subclass from the abstract BatchJob **5600** and add their own required attributes and behavior. A "Bill Customer" batch job may need the identifier of the customer to bill and the time period for which to bill. These should be attributes added to the concrete subclass: BillCustomerBatchJob **5602**. In addition, the concrete class needs to supply the actual logic that the batch job performs (along with any pre- and post-run logic).

Finally, the concrete batch job class should provide some way to start-up all of the pending jobs in its class. A class method is implemented on the abstract class to start all pending jobs. This method can be overridden by any concrete extension of the BatchJob superclass.

By implementing batch job instance as any other type of object, the batch architecture may then take advantage of the same system services available to all other business objects in the system (persistence, transaction management, error handling, logging, security, etc.).

Benefits

Natural Representation. Each type of batch job is represented by an object. This allows it to interact with the rest of the system in a natural way.

Extensibility. By providing an abstract superclass, adding new types of batch jobs only require adding a new concrete class to the system.

Architectural Separation. Batch Jobs that are not implemented "inside" the object-oriented environment can still be tracked by the batch job objects. The rest of the system is unaware of the batch job objects.

FIG. **57** illustrates an object interaction graph of a possible implementation of the example of FIG. **56**. FIG. **57** illustrates a batch scheduler **5700** which interfaces a BillCustomer Class component **5702** which in turn interfaces a BillCustomer BatchJob component **5704**

ISO New England energy exchange. A net-centric internet system build for managing functions associated with a competitive energy market. The energy exchange is implemented in Java across client and server components and using CORBA as a communications architecture.

Batch processes are often highly resource intensive. In many cases the required throughput demands the use of multiple processors, possibly distributed, to provide scalability. How, then, should one structure one's batch workload to facilitate a robust and scaleable system?

BUW

One of the primary techniques used to achieve scaleable batch applications is parallel processing. There are many different types of parallel processing, but the simplest and easiest to exploit occurs when the problem domain contains many independent work items. In this case, the work can simply be divided among the available processors, providing nearly linear scaling.

Happily, this is exactly the situation encountered in many batch systems. However, also given the nature of batch processing, the variety among the various work items will likely be large. This of course leads to the need to treat some items differently than others, and there goes the nice clean scaling model.

With this in mind, some alternative approach must be used which will allow a cleanly scaleable framework to handle multiple heterogeneous work types.

Therefore, one creates an abstraction which represents a batch unit of work. Now the design tension comes in. Clearly one common abstraction is easy to parallelize, but not very interesting to manipulate due to its very generic nature (at least if you're interested in type safety). This quickly leads us to design a shallow tree of more interesting abstractions, and again one's clean scaling model seems threatened.

The key is to treat the work units as top level abstractions while they are being routed among processing nodes and to treat them as more interesting derived abstractions when internal to a node. Treating them as topmost abstractions between nodes provides a good lever for robust processing, as typical actions like IO/persistence, recovery, auditing, etc. can often be treated uniformly for all types.

Treating work units as derived abstractions while internal to a node is achieved by actually creating the abstractions within the node. See the Abstraction Factory pattern for details on one way to achieve this.

So are we safely scaleable now? Not necessarily. There is still the danger that a given processing node will be presented with a unit of work it cannot deal with.

Kind of like asking a parking meter for a hot pastrami. This situation can be avoided with proper workflow, or with sufficient structure, a dynamic library loading version of the Abstraction Factory could, in effect, tell the parking meter how to fix sandwiches. This of course, has the effect of a one time performance hit as the processing node is instrumented with new capabilities.

Implementations of this pattern will vary widely depending on the selection of languages and technical architectures. The key is that the all work units in the system are derived from a single abstraction. This abstraction contains key interfaces that are appropriate at the workflow level. Derived abstractions add interfaces as needed functionally.

Abstraction Factory, in which concrete objects are created by the factory and returned to the Factory's client as an abstraction. CSH.

Processing Pipeline

FIG. **59** illustrates a flowchart for a method **5900** for structuring batch activities for simplified reconfiguration. In operation **5902**, a series of processing steps are prepared for performing on input objects being streamed into a batch processing system. Each of the processing steps is encapsulated within a filter in operation **5904**. The input objects are received and processed in the filters in operation **5906**. In operation **5908**, results are delivered from the filters incrementally during the processing of the input objects for reducing latency and enabling parallel processing. In operation **5910**, connectors are utilized for connecting at least two filters each having a processing step for creating a process. One of the filters is an input filter of the process and another of the filters is an output filter of the process. Connectors are also used in operation **5912** for connecting input and output filters of different processes for forming a scalable system.

There may be several instances of a particular type of filter running in parallel. A portion of the filters may be active and a portion of the filters may be passive. In such a situation, the active filters may pull input data and data may be pushed into the passive filters. Additionally, the input filter of the process may be an active filter and the remaining filters of the process may be passive filters.

The connectors may perform the steps of acting as a choke point for data to be pulled from a filter, connecting serial

SAP - Exhibit 1105
SAP v. Valtrus et al. - IPR2025-00420
Page 223 of 281

filters defined as independent processes, and/or multiplexing to demultiplexing from several filters of the same type running in parallel. As another option, one of the filters may be positioned between the input and output filters of the process for translating an output of the input filter into an input type of the output filter.

How do I define a disciplined strategy to structure the components performing processing steps within a batch system so that the system is cleanly partitioned while maintaining performance and scalability goals?

Often batch processing systems perform a series of processing or transformation steps on input objects that are streamed into the system. Implementing such a system as a single component is not feasible for several reasons: portions of the component must be developed by several developers, requirements are likely to change and it is difficult to cleanly partition the modules resulting in a highly coupled system.

Compounding the difficulty in implementing the system is the fact that most batch systems must satisfy the following challenging requirements:

Must be able to satisfy extremely stringent performance criteria.

The system must scale to meet client's volume.

The system must be flexible enough to be adapted to various contexts.

These requirements are difficult to meet for any system, and batch systems' stringent demands often lead developers to think they cannot use component technology. Building a procedural batch system to satisfy the requirements listed above may result in a complicated set of modules that are difficult to maintain as the system is scaled. By utilizing component technology's ability to manage complexity through encapsulation, a component-based batch system can more easily be defined with clean partitioning than when using a procedural paradigm. Defined with foresight, this partitioning enables the system to scale to meet difficult performance requirements.

Therefore, encapsulate each processing step within a filter component. A filter consumes and delivers its results incrementally, rather than consuming all of its input before producing output. The incremental nature of filters allows them to significantly reduce latency and enables true parallel processing.

A supplier provides the input to each filter, and the filter's output flows to a consumer. Suppliers and consumers may be objects that read files, databases or queues, other filters or any type of object supplying or accepting data. In order to produce a flexibly arranged system, connect the initial supplier, the filters and the final consumer with pipe components that are responsible for implementing the data flow between adjacent filters.

As a result of filter processing's incremental nature, one or more filters, tied together with pipes, define a process's Logical Unit of Work (LUW); i.e., the filters defining the steps of the process are sandwiched by the beginning and ending of the transaction. Expanding this model, each subsystem representing the LUW can be modeled as a filter with input and output that encompasses the internal filters. These filters are then tied together through the use of pipes to represent the system. In this manner, the Processing Pipeline pattern offers a consistent way to view the system that scales to whatever size and degree of complexity the system grows.

Benefits

Scalability. Each filter performs its data processing and transformation independently of other filters. By leveraging off some pipe forms' multiplexing/demultiplexing techniques, there may be several instances of a particular type of filter running in parallel.

Partitioning. As a result of encapsulating each processing step within a filter component it becomes easier to manage the balance between coupling and cohesion since there are disciplined and well-defined interfaces surrounding the components.

Flexibility. Since filters make little assumptions about the world around them, they can be arranged in any manner; several filters can be combined together and wrapped by a larger-grained filter; filters can be dynamically assembled at run-time depending on some context, etc.

Filters

At a high level, there are two types of filter components: active filters and passive filters. An active filter pulls input data from its suppliers, processes the data and outputs the result to its associated consumer. In contrast, input data is pushed into a passive filter, which then performs its processing step and outputs to its consumer.

Typically a system is defined by an active filter at the beginning of the Processing Pipeline, that pulls input data from the data source and initiates further processing by pushing the data to a chain of passive filters situated down the pipeline. Often the active filters are only responsible for pulling data into the system, while the core business functionality is performed by passive filters.

Because active and passive filters demonstrate different levels of pro-activity, it is useful to further break down the type of consumers and suppliers into four general types: push suppliers, pull suppliers, push consumers and pull suppliers. These four simple abstract interfaces help segregate the fundamental, yet disparate, behaviors. Active filters inherit both from PullConsumer and PushSupplier. Active filters' sources inherit from PullSupplier, and their destinations inherit from PushConsumer. Passive filters inherit from PushConsumer and PushSupplier. Passive filters' sources inherit from PushSupplier, and their destinations inherit from PushConsumer.

Pipes

While filters define the basic processing steps, pipes define how to flexibly configure the system. Pipes can be used to connect filters in a wide range of configurations:

Acting as a choke point for data to be pulled from an active filter

Connecting serial filters defined as independent processes

Multiplexing to/demultiplexing from several filters of the same type running in parallel

Pipes may use buffering, multiplexing and de-multiplexing techniques in order to transfer data between filters. Some examples of useful pipe implementations include:

Channeled Pipes. Perhaps the most generally useful form of a pipe is based on the CORBA Event Channel object, which can connect any number of Push/Pull Suppliers to any number of Push/Pull Consumers.

Multithreaded Pipes. These pipes route data to one of several filter threads. The data can then be joined back to the primary thread on the other end of the filter with a demultiplexing pipe.

Database Queue Pipes. These pipes wrap around a database queue to enable seamless data transfer between processes.

The various command shells enable filter programs to be tied together into a Processing Pipeline.

Collaborations

Abstraction Factory. Often filters will need to produce new data objects from input but are only aware of the data's abstract interfaces. As a result of this generality, the filters will need to utilize an abstraction factory to produce concrete objects without knowing their concrete class types.

BUSINESS LOGIC SERVICE PATTERNS (1024)

As is stated in the Component Technology Architecture Framework, "Business components are the core of any application, they represent concepts within the business domain. They encapsulate the information and behavior associated with those concepts. Examples of business components include: Customer, Product, Order, Inventory, Pricing, Credit Check, Billing, and Fraud Analysis." These are the components that in many cases have been the most elusive for reuse but hold the highest promise for attacking the cost of development. In this area there are at least three targeted categories of business components, Common Business Components, Common Business Services and Common Business Facilities.

Common Business Components are those components from the preceding list that encapsulate key business concepts. At one level these components represent cross application components that are common to a plethora of applications. These include concepts like Customer, Company, Account, Shipment, etc. These common components normalize how basic behavior surrounding common business concepts can be normalized. Common Business Components are very concerned about the validity of the relationships they have with other components and ensuring that the information relationships are maintained correctly.

Common Business Services deal with the higher level services that abstract out the "Business Unit of Work" or more transactional aspects of business processing. Having components that capture key processing concepts normalizes the processes for handling business events. These are services like credit checks, ordering, servicing problems, shipping, product selection, etc. They tend to capture business practices and when reused enable a company to increasingly leverage the value of those practices.

Common Business Facilities are those services that deal with areas of more engineering component type reuse. These include base common facilities like reason codes, currency management, telephone and address manipulation and validation of these common business types.

How do the Patterns in This Section Help?

The patterns described in this section represent some initial attempts to capture basic concepts that are useful in the area of Common Business Facilities. They are by no means exhaustive but represent building blocks in a complete solution. Both provide tremendous value in solving two key challenges which appear on every engagement.

The Constant Class pattern describes a facility for ensuring correct data at the attribute level.

The Attribute Dictionary describes a facility for encapsulating architectural mechanisms within business objects.

Attribute Dictionary

FIG. **58** illustrates a flowchart for a method **5800** for controlling access to data of a business object via an attribute dictionary. A plurality of attribute values for a business object are stored in an attribute dictionary in operation **5802**. A plurality of attribute names are provided in the attribute dictionary for the stored attribute values in operation **5804**. Next, in operation **5806**, it is verified that a current user is authorized to either set or get one of the attribute values upon a request which includes the attribute name that corresponds to the attribute value. The attribute value in the attribute dictionary is obtained or updated if the verification is successful and an indicator is broadcast upon the attribute value being updated in operations **5808** and **5810**.

In one embodiment, a list of the attribute names may be outputted in response to a request. Additionally, the list may also include only the attribute names of a portion of the attribute values of the business object that are present.

In one aspect, the attribute values may be obtained for auditing or rollback purposes. In another aspect of the present invention, a dirty flag may be set upon the attribute value being updated.

Typically, business objects include "getter" and "setter" methods to access their data. How can I support value-added processing, such as logging events for changes, without impacting application code?

Typically, business objects store attributes in instance variables. The application code for a typical setter for an attribute is depicted as:

```
public void setBalance(Float newBalance) {
    my Balance = newBalance;
    return;
}
```

Initially, this is straightforward. However, after all of the attribute setters and getters have been coded, the need may arise for an event to be broadcast each time an attribute is updated. The code for a simple setter would need to change to become:

```
public void setBalance(Float newBalance) {
    myBalance = newBalance;
    this.notifyChanged("Balance");
    return;
}
```

Now each attribute setter must contain the call to the 'notifychanged' architecture method. This implementation forces architecture mechanisms to be intrusive to application code. Moreover, addition or extension of architecture processing should not impact business logic. One new line of code alone may not seem like a large burden on application developers. However, many other architecture requirements might later affect each setter or getter.

As another example, before updating an attribute, a check may be required to determine if the current user has security rights to update attributes. Also, after successful update, a dirty flag may be set, or an audit log may be performed. The code for each setter now looks as follows:

```
public void setBalance( Float newBalance ) {
    // keep track of my original balance,
    // for post-change processing, then do
    // some pre-processing to check
    // that the user has access rights
    Float oldBalance = myBalance;
    this.assertCanSetAttribute( "Balance" );
    // finally update the balance, then
    // broadcast, set the Dirty Flag,
```

-continued

```
        // and log
        myBalance = newBalance;
        this.notifyChanged( "Balance" );
        this.makeDirty();
        this.logChanged( "Balance", oldBalance);
    }
```

Thus, each added architecture framework for gets and sets must be manually added to all getters and setters. Such changes impact application developers during coding and maintenance. Moreover, they also complicate business logic with technical details. Therefore, the application architecture should control access to a business object's data. This will separate out reusable, technical, architecture details. Business objects should use an Attribute Dictionary to provide an architectural hook for attribute getters and setters. Moreover, this framework should handle all architectural processing related to the update and access of data, transparently to application logic.

Rather than using instance variables, the Attribute Dictionary holds all attribute values for the object. This dictionary is a collection, keyed by attribute names. Then the architecture can provide generic architecture methods to get and set attributes in the dictionary.

Business objects could each delegate directly to the Attribute Dictionary within the attribute getter and setter. However, rather than having each business object talk directly to the Attribute Dictionary, simple helper methods can be created in a superclass for business objects. This simplifies the interface for application developers, who do not need to know about the Attribute Dictionary. This also allows for business object specific logic to also be added prior to and after the dispatch to the Attribute Dictionary.

The code for a simple setter now would look like:

```
class Account extends BusinessObject {
public void setBalance( Float newBalance) {
    // set my balance with the new value
    // passed in. The architecture will handle
    // any technical details related to
    // setting the data.
    this.setAttribute( "Balance", newBalance);
}
}
```

The architecture superclass will then perform the following:

get the original value, perhaps for auditing or rollback purposes

check if the user has security access to set the attribute

update the attribute on the Attribute Dictionary

if successful, broadcast and log the change

The Attribute Dictionary would then contain the code to:

update the value for the given attribute name

set the dirty flag

This illustrates that both the superclass facade and the Attribute Dictionary can have different processing. In general, one generic location for getting and setting attributes supports (but is not limited to):

logging

broadcasting

dirty flag

security checking

NULL field value handling

This logic will be either in the facade methods (for any code that is business object specific), or the generic methods on the dictionary, thereby shielding developers from this added complexity.

Benefits

Maintainability. Architecture code can be added and changed in one place for all objects, without change to the application code.

Flexibility. The implementation of the storage mechanism can be changed as needed to improve performance.

Readability. The methods used in application code to retrieve and update fields on the object are generic. These methods do not have excess architecture code to detract from the purpose of the method.

Object Model

FIG. **60** illustrates the manner in which the AttributeDictionaryClient **6000** is the facade which delegates to the AttributeDictionary **6002**. For example, business objects would inherit this behavior. AttributeDictionaryClient **6000** probably wouldn't be the immediate superclass, but it would be somewhere in the hierarchy. In this manner, stateful business objects, like Account or Customer, can easily take advantage of the Attribute Dictionary.

The attributeValues attribute on the Attribute Dictionary is shown as an instance of the HashMap class **6004**, which stores key value pairs. The HashMap Collection is used to provide access to attribute values based on the attribute name. This is required for a direct lookup of values associated with attribute names. Such lookup can use string representation of the attribute names.

Object Interaction Diagrams

There are four interactions for this framework: Simple Attribute Getter, Simple Attribute Setter, Failed Attribute Getter, and Retrieval of Attribute Names. FIG. **61** illustrates the internal implementation of the dictionary.

FIG. **61** depicts the use of the containsKey( ) method **6100** on the HashMap to ensure that the value will exist before the get( ) method is used. This proactive search for the value ensures that the nullPointerException is not thrown from the AttributeDictionary. The performance of such methods will be checked during testing. If such processing is not performant, the code can be altered and the call to containsKey( ) removed. In that case, the HashMap will need to wrap a try-catch block around the get( ) method. FIG. **62** illustrates a method **6200** that dictates that any nullPointerException that is thrown would be caught and rethrown as the more user-friendly exception in the attribute dictionary pattern envronment. FIG. **63** illustrates the Get the Attribute Names method **6300** in the attribute dictionary pattern envronment.

Public Interface

The following are methods on the AttributeDictionary. The AttributeDictionaryClient exposes similar public methods.

public Object getAttribute(String attributeName) raises AttributeNotFoundException;

The return value of getattribute( ) is typically a wrapped primitive, or Java type, for most attributes. This includes, for example, an account balance (Float) or account number (String). The return value of these wrapped primitives must be cast, as illustrated in the following example:

SAP - Exhibit 1105
SAP v. Valtrus et al. - IPR2025-00420
Page 226 of 281

```
class Account extends BusinessObject {
    public Float getBalance() {
    // get my balance using the superclass facade
    // cast the return value before returning it
    return (Float)( this.getAttribute( "Balance" ));
    }
}
```

Other methods on the AttributeDictionary include:

```
public void setAttribute( String attributeName, Float attributeValue);
public void setAttribute( String attributeName, String attributeValue);
public void setAttribute( String attributeName, BusinessObject
attributeValue);
. . .
```

These overloaded methods create a generic interface to the AttributeDictionary for attribute setters. They ensure type checking, such that no attributes will be set to a value other than those for which an overloaded method exists.

public String[ ] attributeNames( );

The method attributeNames( ) returns a collection of the names of only those attributes that have been populated (or set) on the dictionary. This might be useful for other frameworks, which may want to iterate over all attributes. At any particular time, a business object may not contain all of its attributes (e.g., because of partial retrieval from the database). So this may be a subset of the full attribute list for the object.

Constant Class

FIG. **64** illustrates a flowchart for a method **6400** for managing constants in a computer program. In operation **6402**, a plurality of constant names are provided with each constant name having a corresponding constant value. The constant names are grouped into constant classes based on an entity which the constant values represent in operation **6404**. Access is allowed to the constant values in operation **6406** by receiving a call including the corresponding constant name and corresponding constant class.

In one aspect, the constant values may be changed upon being accessed. In another aspect, the constant value may also include an enumeration. Also, in one embodiment, accessor logic modules may be assigned to a plurality of the named constants with the accessor logic modules being executed upon the accessing of the corresponding constant value via the accessor logic module. Also, the accessor logic modules may be edited per the desires of a user. Additionally, the constant values may be accessed without the accessor logic modules.

Literals are hard-coded constants referenced in multiple places. How can source code refer to literals in a maintainable fashion?

The concept and value of named constants have been realized for quite some time. The idea can date back to Assembler language naming memory locations where data was stored. The purpose is to give the ability to refer to fixed values by the name of what they represent rather than by the quantity they are set to.

Named constants allow a programmer to "parameterize" a system. This allows a programmer to change a constant's value in a single place rather than every place the constant is used. In addition to the maintenance gain, readability is also increased.

Many languages offer mechanisms to implement named constants. These include PoolDictionaries (Smalltalk), enums (C and C++) and public static final declarations (Java). Difficulties arise during implementation of these mechanisms with respect to type constraints, visibility, and type checking.

Using these traditional approaches, results in global namespace for these literals. This can result in name collisions. For example, the name HIGH to define a large magnitude could translate into different values for different uses. A HIGH temperature could be 95 while a HIGH altitude could be 39000.

In addition, constants often belong to logical groupings. For instance, STOCK, BOND, and OPTION are all types of financial instruments. These belong in a some sort of collection.

A consistent, quality method to represent constants in an object-based system is required.

Therefore, represent named constants in a separate class, grouping categories of constant values together within one name space. Constants tend to naturally fall into logical groupings. Each grouping should be represented by its own class. For instance, all of the constants used by a PhoneNumber object to capture the various types of PhoneNumber (i.e. home, business, fax, cell, pager, etc) can be accessed through a PhoneTypeConstants class.

If constants are obtained by other means than explicit language constructs like "public final int HOME_ADDRESS" than public accessors are used to insulate a client from changes in how the constant is obtained. In this case the values of each of the constants should be defined privately inside the Constant Class. Public accessors are then provided for clients to obtain the constant values. This allows for "changing constants". Business-related values that may seem constant at design and construction time very often are not. Some of these "constants" may eventually require some logic to determine their value. If clients obtain constants through accessor methods, no changes (except within the accessor) will have to be made if the logic is added. This is a particularly safe practice when programming rules dictate all constants to be stored and retrieved from database tables.

In the case where constants are defined within the class itself most OO languages, excepting Smalltalk, allow for some type of const definition. In this case by using a const construct (i.e. static final int PhoneNumberType FAX=new PhoneNumberType( )) it is not necessary to have public accessors and private definitions. Declare the class type, create static final instances of the type and do not provide a public constructor. This ensures the type safety and provides easy to access members in the Eiffel style.

Moreover, public accessors in either strategy provide for type-safe enumerations. Enumeration is a special type of constant that deserves attention. A TypeConstant class can provide enumeration by implementing some key methods that provide for supporting iteration over the elements of the enums. In Java, for example, this entails implementing the Enumeration interface.

Benefits

Maintainability. Groups all valid values together and ensures they can not be created or passed as parameters by any other method.

Type Safety. Enumeration values can be type-checked by a compiler in method parameters and return values.

A common application pattern where this use of constants was applied was in the modeling of instances vs instance

SAP - Exhibit 1105
SAP v. Valtrus et al. - IPR2025-00420
Page 227 of 281

types where the types added no additional behavior. In two different customer care applications this came through as the objects like PhoneNumber, PhoneNumberType, RatePlan & RatePlanType, etc. This example has not yet been updated to JavaBeans.

```
package Party;
import java.util.*;
public class PhoneNumberType {
    static final Vector types = new Vector();
    static final PhoneNumberTypeFAX = new PhoneNumberType(0,
"Fax");
    static final PhoneNumberType CELL = new PhoneNumberType(1,
"Cell Phone");
    static final PhoneNumberType HOME = new PhoneNumberType(2,
"Home");
    static final PhoneNumberType WORK = new PhoneNumberType(3,
"Work");
    static final PhoneNumberType PAGER = new PhoneNumberType(4,
"Pager");
    private final int phoneNumberTypeOrd;
    private final String typeId;
    private PhoneNumberType(int i, String id) {
        phoneNumberTypeOrd = i;
        typeId = id;
        types.addElement(this);
    }
    public final static Enumeration elements() {// allows for enumeration
        return types.elements();
    }
    public static void main(String args[]) {
        Enumeration elements = PhoneNumberType.elements();
        PhoneNumberType pt;
        while (elements.hasMoreElements()) {
            pt = (PhoneNumberType)elements.nextElement();
            System.out.println(pt.toString());
        }
    }
    public String toString() {
        return typeId;
    }
}
```

This type partition is used by PhoneNumber. See maino for uncommenting a line that demonstrates the type safety protection through the use of static final and private constructors.

```
package Party;
import java.io.PrintStream;
import java.io.StringWriter;
public class PhoneNumber
{
        private PhoneNumberType phoneNumberType;
        private String areaCode;
        private String prefix;
        private String suffix;
        public PhoneNumber() {
        areaCode = null;
        prefix = null;
        suffix = null;
}
public PhoneNumber(String aPhoneNumber)
{
        parsePhoneNumber(aPhoneNumber);
        setPhoneNumberType(PhoneNumberType.HOME);
}
public PhoneNumber(String anAreaCode, String aPrefix, String aSuffix)
{
        areaCode = anAreaCode;
        prefix = aPrefix;
        suffix = aSuffix;
}
```

-continued

```
public String areaCode()
{
        return areaCode;
}
public void areaCode(String anAreaCode)
{
        areaCode = anAreaCode;
}
public boolean equals(String aPhoneNumber)
{
        PhoneNumber tempPhoneNumber = new
PhoneNumber(aPhoneNumber);
        return equals(tempPhoneNumber);
}
public boolean equals(PhoneNumber aPhoneNumber)
{
        if (areaCode() == null && aPhoneNumber.areaCode() != null ||
        aPhoneNumber.areaCode() == null && areaCode() != null)
            return false;
        if (areaCode() != null)
        {
        if (areaCode().equals(aPhoneNumber.areaCode()) &&
prefix().equals(aPhoneNumber.prefix()) &&
            suffix().equals(aPhoneNumber.suffix()))
            return true;
        else
            return false;
    }
    if (prefix().equals(aPhoneNumber.prefix()) &&
suffix().equals(aPhoneNumber.suffix()))
        return true;
    else
        return false;
}
public static void main(String argv[])
{
    PhoneNumber aPhoneNumber;
    System.out.println("Testing construction & comparison!");
    if (argv.length == 0)
    {
        System.out.println("Test with no area code - no masks ");
        aPhoneNumber = new PhoneNumber("5579203");
    aPhoneNumber.setPhoneNumberType(PhoneNumberType.FAX);
        System.out.println(aPhoneNumber.toString());
        System.out.println("Test with area code - no masks");
        aPhoneNumber = new PhoneNumber("2065572039");
    aPhoneNumber.setPhoneNumberType(PhoneNumberType.WORK);
        System.out.println(aPhoneNumber.toString());
        System.out.println("Test with normal masks");
        aPhoneNumber = new PhoneNumber("(206)557-3920");
    aPhoneNumber.setPhoneNumberType(PhoneNumberType.PAGER);
        System.out.println(aPhoneNumber.toString());
        System.out.println("Test equality 5578215 557-8215");
        PhoneNumber temp1 = new PhoneNumber("5578215");
        temp1.setPhoneNumberType(PhoneNumberType.CELL);
        PhoneNumber temp2 = new PhoneNumber("557-8215");
        temp2.setPhoneNumberType(PhoneNumberType.CELL);
        // temp2.setPhoneNumberType(new PhoneNumberType(6,
"TOY")); // enum type safety w/private ctor
        System.out.println(temp1);
        System.out.println(temp2);
        System.out.println("temp1 == temp2: "+
temp1.equals(temp2));
        return;
    } else {
        aPhoneNumber = new PhoneNumber(argv[0]);
        System.out.println(aPhoneNumber.toString());
    }
}
private void parsePhoneNumber(String aPhoneNumber)
{
    StringBuffer aStr = new StringBuffer(aPhoneNumber.length());
    int i = 0;
    do
        if (Character.isDigit(aPhoneNumber.charAt(i)))
            aStr.append(aPhoneNumber.charAt(i));
    while (i++ <aPhoneNumber.length() - 1);
    String tempString = new String(aStr);
    if (aStr.length() == 7)
```

-continued

```
    {
        prefix (tempString.substring(0, 3));
        suffix(tempString.substring(3, 7));
        return;
    }
    areaCode(tempString.substring(0, 3));
    prefix(tempString.substring(3, 6));
    suffix(tempString.substring(6, 10));
}
public String prefix()
{
    return prefix;
}
public void prefix(String aPrefix)
{
    prefix = aPrefix;
}
/**
* This method was created by a SmartGuide.
* @param sw StringWriter
*/
public void printOn (StringWriter sw ) {
    sw.write(((areaCode != null || areaCode =="") ?("(" + areaCode() +
")": ""));
    sw.write(this.prefix());
    sw.write("-");
    sw.write(this.suffix());
    return;
}
public void setPhoneNumberType(PhoneNumberType pnt) {
    phoneNumberType = pnt;
}
public String suffix ()
{
    return suffix;
}
public void suffix(String aSuffix)
{
    suffix = aSuffix;
}
public String toString()
{
    return new String(phoneNumberType.toString() + ":" + ((areaCode
!= null || areaCode == "") ? ("("+ areaCode() + ")") : "") +
prefix() + "-" + suffix ());
}
}
```

Alternatives

Smalltalk allows for grouping logical constants in PoolDictionaries as in TextConstants. This is simply a global dictionary with key value pairs that simplifies and improves readability by using well understood names like "Space" and "Tab". However, they are global variables and they are not automatically recreated when you file in code that depends on them.

When constants are implemented in a class within Smalltalk accessors must be used. There is no real language notion of final or const in Smalltalk that would allow for accessing member variables.

COMMUNICATIONS SERVICES PATTERNS (1008)

An original tenet of component-based design has been simplified distribution of functionality. A ccording to the original argument, up-front definition of component boundaries and their interfaces would simplify the configuration of functionality on the network. Even though component-based design has simplified distribution, it has not guaranteed success. Networks introduce performance issues and failure conditions that didn't exist in non-distributed solutions. If a solution is to be successful, these issues can't be ignored.

Each pattern in this se ction addresses a difficulty associated with distributed computing. Every pattern reflects a problem and a solution to issues encountered by other development teams

Legacy systems running on mainframes, Unix boxes, etc. are an important part of today's client projects. The majority of today's clients have existing computer systems that cannot be easily rewritten or ignored. Integration of these older systems with the newly developed applications is often imperative to the success of the project. Any newly developed components must leverage the existing functionality on these Legacy systems. The Legacy Wrapper pattern addresses this problem. It describes a common pattern for tackling the integration issues associated with reusing functionality from existing systems.

Server-side components implement services for use by the Clients in an application. These components should clearly specify the interfaces an services the provide, but how should they make them available? A well-known central service, e.g., a Name Service or Trader Service can be used to make the interfaces available to all Clients, but is that always warranted or prudent? Should every Client have access to every service? The Locally Addressable Interface (LAI) and Globally Addressable Interface (GAI) patterns describe two approaches to this problem.

The performance characteristics of remote components are very different from "in process" components. The cost of requesting and transmitting data between remote components is much higher and should be considered in a distributed solution. As a result, distributed solutions often call for communication patterns that improve upon the performance aspects most important to the system. The Structure Based Communication pattern addresses the "chattiness" associated with distributed applications. It helps reduce network load and increases system response time. The Paging Communication pattern addresses the common need to retrieve and display large lists of data. It shows how incremental fetching can be used to provide much better perceived responsiveness in GUI based applications.

The cost of locating a remote service and establishing a connection to that service can also be a costly endeavor. The Refreshable Proxy Pool pattern describes a robust and efficient way to minimize this "lookup" activity.

Most recent component-based systems use middleware such as CORBA, DCOM or Java RMI to specify the interfaces provided by components and the associated data types. However, such middleware is not always available, or directly applicable. In such situations the Stream Based Communication pattern, or one of its descendants, the Fixed Format Stream and SeyDescribing Stream patterns might be applicable. These patterns describe different techniques for efficiently streaming data between processes. While they all share a common solution to a common problem, the solutions present different tradeoffs between implementation simplicity, performance and flexibility.

A Null value represents the "empty set" and is an important value in distributed component solutions.

Some languages, such as Java, support Null as a specific value, whereas other languages do not (e.g., C++ which uses zero and context to determine Null). This language mismatch can cause problems in distributed systems that use more than one language. The Null Structure pattern describes this problem and proposes a solution.

Fixed Format Stream

FIG. **65** illustrates a flowchart for a method **6500** for providing a fixed format stream-based communication system. In operation **6502**, a sending fixed format contract on

interface code is defined for a sending system. A receiving fixed format contract on interface code is also defined for a receiving system. A message to be sent from the sending system to the receiving system is translated in operation **6504** based on the sending fixed format contract. The message is then sent from the sending system and subsequently received by the receiving system in operations **6506** and **6508**. The message received by the receiving system is then translated based on the receiving fixed format contract in operation **6510**.

In one embodiment of the present invention, information in the translated message received by the receiving system may also be stored in a relational database. In one aspect, the fixed format contracts may be included in meta-data of the message. Also, in another aspect, the message may include an indication of a version thereof.

In one situation, one of the systems is an object-based system and one of the systems may be a non-object-based system. In another situration, both of the systems are may be object-based systems. In a third situation, both of the systems may be non-object-based systems.

Stream-based communication is a very effective pattern for relaying data, data structures, and meta-data. Meta-data is information about the data like data structure, data types, etc. using a shared, generic format. How can the message format be shared between systems so as to create the most straightforward and best performing stream-based mechanism?

Often, it is determined that a stream-based communication mechanism should be used to transport information between systems. Stream-based communication is a pattern where information is transported from one system to another using a simple stream and a shared format to relay the data structure and meta-data information.

FIG. **66** illustrates two systems **6600** communicating via a stream-based communication **6602** and using a common generic format to relay the meta-data information.

However, when implementing Stream-based Communication, a number of factors influence the method for enabling each system with a "shared format." The "shared format" provides the meta-data information needed to interpret the raw data in a stream. This shared format is like a secret decoder ring for systems sending and receiving messages. It allows the systems to convert structured data (objects, strings, etc.) into raw data and raw data back into structured data. This is needed to transmit the structured data across the network.

On many projects, the following factors influence the details of communicating using a stream.

- High performance—System performance is always a factor, but sometimes it is one of the most important factors in a system.
- Short development time—The system must be operational in the shortest possible timeframe.
- Stable information characteristics—In some solutions, the data and the structure of the data are stable and unlikely to change.

In cases like this, how can one optimize the benefits of stream-based communication and implement only the most basic capabilities that one requires?

Therefore, use the Fixed Format Stream pattern to create a stream-based message that uses fixed format contracts to share the formatting information and meta-data between systems.

Fixed format contracts are maps that contain the meta-data information such as the data structure, data separators, data types, attribute names, etc. They describe how to translate Fixed Format messages onto a stream and off of a stream.

FIG. **67** illustrates an example of a Fixed Format message **6700** associated with the fixed format stream patterns. The location and size of each attribute in the message is fixed and known at design time. In the example below, it is known that the command will be in bytes 1–9, the first name will be in bytes 10–29, the last name in bytes 29–49, etc. This information (meta-data) is used by the Fixed Format contracts to convert Fixed Format messages from data structures to raw data and back again.

FIG. **68** depicts the complete Fixed Format Stream pattern associated with the fixed format stream patterns. A data structure on System A **6800** is translated to a Fixed Format message (raw data) using a Fixed Format contract. The message is put in the stream **6802** and sent to System B **6804**. System B **6804** receives the Fixed Format Message (raw data) and uses its Fixed Format contract to recreate the data structure. The same process works in reverse when System B **6804** responds to the message request.

Benefits

- Performance. Because there is no time spent on look-ups or dynamic translation of the message, performance is better than with other variations of Stream-Based Communication.
- Small Message Size. Each Fixed Format message contains only data to be sent to the other system. These messages contain no meta-data and are smaller than those in Self-Describing Streams.
- Simplicity. Translating and parsing information onto and off of the stream is straightforward and easier than with the other variations of Stream-Based Communication. The behaviors for the Fixed Format Streaming are contained in the fixed format contracts on the interfaces of the sending and receiving systems and thus easy to find.

Object Friendly. This pattern is very straightforward to implement in object based systems. The objects contain the fixed format contracts and manage the translation and parsing onto the stream. These objects can access their own private behaviors which makes the interface much simpler.

Implementing this pattern is very straightforward. Define corresponding fixed format contracts on the interface code of both the sending and receiving systems. FIG. **69** illustrates fixed format contracts **6900** containing meta-data information for translating structured data onto and off of a stream.

In non-object systems, define a fixed format contract on the parsing interface module of the sending system. The interfacing module on the sending system can use the contract as a map for how to translate and write the data onto the stream. Define a corresponding fixed format contract on the interface modules of the receiving system. The interface module on the receiving system can use the contract to read and translate the data off of the stream.

In object-based systems, make each object responsible for its own fixed format contract. Using this contract, each object is able to retrieve and parse its attribute values onto a stream as strings (streamOn) and each object class should be able to parse attributes off of a stream and put them into a new instance of an object (streamOff). Also, it is good practice to include the version of the format within the stream so that concurrent format versions can be accommodated in the design.

Below is a pseudo-code example of an object-based system communicating with a non-object system using stream-based communication and a fixed format contract.

SAP - Exhibit 1105
SAP v. Valtrus et al. - IPR2025-00420
Page 230 of 281

FIG. **70** illustrates a Customer object **7000** in an object-based system **7002** streaming itself into a stream **7004**, the stream being sent to a non-object system **7006**, this stream being read and the data inserted into a relational database **7008**.

1. The CustomerObject with attributes name, sex, and age has a method "streamOn: aStream." It is invoked with an empty stream as the argument 'aStream'. The CustomerObject "streamOn:" method goes through each of the object's attributes and parses each values as a string onto the stream.

The fixed format contract here is embodied in the order that this method parses the attributes onto the stream. A pseudo-code example in Java is the following: Note-Assume that "asString( )" converts the receiver to a string and that "padWithSpaces( )" pads the string with spaces and makes the string the length specified.

```
/** Stream my attribute values on aStream **/
public void streamOn (OutputStream aStream)
{
    aStream.write(this.getName().asString().padWithSpaces(10));
    aStream.write(this.getSex().asString().padWithSpaces(7));
    aStream.write(this.getAge().asString().padWithSpaces(3));
}
```

2. The stream is then put into a message communication mechanism like MQSeries or MessageQ and sent to the non-object system.

3. Once at the non-object system, interface code reads through the stream, parses the values off of the stream, converts them to the appropriate types if required, and puts them in a copybook with the appropriate structure. In this example, the fixed format contract is embodied in the structure and type of the WS-SHARED-FORMAT-CUSTOMER working-storage copybook. Refer to the pseudo-COBOL example below.

```
. . .
DATA DIVISION.
FD FILE-STREAM-IN
    RECORD CONTAINS 20 CHARACTERS
. . .
WORKING-STORAGE SECTION.
* * * THIS COPYBOOK CONTAINS THE COMMON FORMAT OF
THE
* * * CUSTOMER IN THE DATA STRUCTURE AND DATA TYPES
01 WS-COMMON-FORMAT-CUSTOMER
03 WS-COMMON-FORMAT-NAME          PIC X(10).
03 WS-COMMON-FORMAT-SEX           PIC X(7).
03 WS-COMMON-FORMAT-AGE           PIC 999.
* * * THIS COPYBOOK IS THIS SYSTEMS VIEW OF A CUSTOMER
O1 WS-CUSTOMER
03 WS-NAME                        PIC X(10).
03 WS-AGE                         PIC 999.
03 WS-SEX                         PIC X(10).
. . .
PROCEDURE DIVISION.
. . .
* * * OPEN THE FILE STREAM AND PUT THE CONTENTS IN THE
* * * WS-COMMON-FORMAT-CUSTOMER COPYBOOK.
OPEN FILE-STREAM-IN
READ FILE-STREAM-IN INTO WS-COMMON-FORMAT-
CUSTOMER
    AT-END CLOSE FILE-STREAM-IN
END-READ.
* * * MOVE THE VALUES INTO FROM THE COMMON FORMAT
INTO
* * * THE WS-CUSTOMER VARIABLES.
MOVE WS-COMMON-FORMAT-SEX        TO WS-SEX.
MOVE WS-COMMON-FORMAT-AGE        TO WS-AGE.
```

-continued

```
MOVE WS-COMMON-FORMAT-NAME       TO WS-NAME.
. . .
* * * CALL A SQL MODULE TO SAVE THIS INFORMATION IN
THE
* * * RELATIONAL DATABASE
CALL "SAVE-CUSTOMER-IN-DATABASE" USING WS-
CUSTOMER.
. . .
STOP-RUN.
```

Conversely, a stream could be created by a non-object system (or another object-based system for that matter) and sent to one's object-based system. In this case, CustomerObject could use a "streamOff: aStream" method and instantiate a new instance of an aCusiomerObject and populate it with the appropriate attribute values.

Eagle Architecture Framework: Uses Stream Based Communication in a number of ways. First of all, it uses it to embed tracing information in CORBA distributed requests. Second of all, it is used to replicate state, between fault-tolerant services.

MCI: Invoice Development Workbench. This workbench helps MCI create error-free invoice definitions for the various Local Bells. Stream-based communication was used as part of an efficient, lightweight persistence mechanism.

Java Serialization: This is a Java defined fixed format for streaming objects.

Object Request Brokers (ORBs) that use CORBA, DCOM, or Java Remote Method Invocation (RMI)—ORBs that use one of these standards implement this pattern. They define an Interface Definition Language (IDL) that is the format or contract of the stream and use stream-based communication as the communication medium.

Collaborations

Stream-based Communication. This is the parent pattern to Fixed Format Stream. In this pattern, information is transmitted using a simple stream and a shared, generic format. The Fixed Format Stream is a more specific implementation of Stream-Based Communication.

Structure Based Communication—This pattern uses a Fixed Format Stream to transmit data structure between systems. It is often used to obtain data from a Server for display in a Client UI.

Bridge (from the Gamma book Design Patterns) describes a way to de-couple an abstraction from its implementation so that the two can vary independently. The Bridge pattern is often used to define collaborations between a business object and a format object while decoupling the business object from its specific stream format.

Abstract Factory (from the Gamma book Design Patterns) is a pattern for creating families of related classes. This could be used with the Bridge pattern to retrieve the format dynamically based on non-static information.

Alternatives

Self-Describing Stream. This pattern is a specific implementation of Stream-Based communication where the messaging format is parameterised and stored on the stream. A message language is used to read and write the format of the message from the stream.

Downloadable Format Stream—This pattern is a specific implementation of Stream-Based communication where the messaging format is stored at a central location and is downloaded by the communicating parties when needed.

FIG. **71** illustrates a flowchart for a method **7100** for delivering service via a globally addressable interface. A plurality of interfaces are provided in operation **7102** and access is allowed to a plurality of different sets of services from each of the interfaces in operation **7104**. Each interface has a unique set of services associated therewith. Each of the interfaces is named in operation **7106** with a name indicative of the unique set of services associated therewith. The names of the interfaces are then broadcast to a plurality of systems requiring service in operation **7108**.

The access may be allowed via structured-based communication. As another option, the names may be broadcasted using a naming service. Also, the naming service may provide the systems requiring service with a location of the interface on a network. In addition, the systems requiring service may be capable of looking-up the interfaces using the naming service.

In a client-server environment, a client makes requests of services on a Server. In such an environment, how might a Server expose its services for use by one or more clients?

In a typical two or three-tiered client-server application, the services are maintained away from the users (Client) on separate Server machines. Whenever a user needs to use a service, the user must send a request across the network to the Server machine.

Before a client can utilize a service, it must find the service. If the client is unable to find the service, it can't ever use it. FIG. **72** depicts a client **7200** that is unable to find the services provided by a server **7202** via a network **7204**.

The client could look for services in a naming service. However, if the services don't exist in the lookup or naming service, the client still can't find and use the service.

Therefore, use a Globally Addressable Interface to expose services to all available clients.

A Globally Addressable Interface builds upon the Interface pattern and the Naming pattern. When implementing a Globally Addressable Interface, a Server's operations are bundled into logical groups using the Interface pattern.

FIG. **73** illustrates the grouping of services **7302** using interfaces **7304**.

For example, all the operations for accessing and viewing customer information (Get Customer, Get Customer Address, etc.) could be bundled into one interface. All the operations for changing customer information (Change Customer, Address, Change phone number, etc.) might be bundled into another interface. Keep in mind, this is an example "bundling" of operations and not the definitive method for bundling operations. Once all the operations have been grouped into an interface, the interface is given a name appropriate to the operations it bundles. Then the interfaces are announced using the Naming pattern. The Naming pattern enables registration of interfaces in a globally available naming service. FIG. **74** illustrates a customer server **7400** publicly announcing its interfaces **7402**.

Until that time, a client can't find the operations and can't use them. Thus, the Server must use the Lookup or Naming pattern to register its interfaces (not methods). Once the interfaces have been registered with such a service, any client can go to the Naming Service, locate an interface, and access an operation in that interface.

Benefits

Public Addressability—Every Globally Addressable Interface is publicly available for use by any client. As a result, any Client can find these interfaces and access these operations.

Stateless Load Balancing. Globally Addressable Interfaces are generally implemented for stateless Servers. When Stateless Servers are used, it is a lot easier to balance the incoming load. Since state or context is always passed into the Server, any call can be directed to any Server that supports a particular operation. If one is busy, the Client can be forwarded on to the next one.

The following is a message trace diagram depicting the interactions associated with a Globally Addressable Interface.

The Message Trace diagrams depict a common Client-Server scenario. A client requests customer data from a Server. The Server finds the data in a database and forwards it back to the client. The Client can then display the data in a User Interface for a user.

The scenario was broken into two message trace diagrams. The first message trace sets the stage for the second. In the first message trace, the Server registers two Globally Addressable Interfaces with a Naming Service. The Client then "looks-up" an interface and establishes a connection to that interface.

Assumptions

CORBA ORB connects Client and Server

CORBA Naming Service used to lookup GAIs

FIG. **75** illustrates a method **7500** including the registering and then locating of a globally addressable interface.

Collaborations

1a. "Bind" the interface name (Update Interface) with it's Remote Object Reference (network location) in a Naming Service. This will allow clients to "lookup" the interface. Once the Interface is registered in the Naming Service, it has become globally addressable. Any client can find the interface and access a operation.

1b. "Bind" the second interface in the same manner as the first.

2. The client instantiates a Proxy (Browsing Interface Proxy) to the Browsing Interface on the Customer Server.

3. The Proxy "looks up" the network location of the Browsing Interface. It makes a request of the Naming Service. It requests the network location of the Browsing Interface.

4. The Naming Service returns the Remote Object Reference (network location) for the Browsing Interface. The Proxy now has all the information it needs to access an operation on the Browsing Interface.

The second message trace builds upon the first. In this message trace diagram, the Client calls the Server through a Globally Addressable Interface. The server finds the appropriate customer data and returns it to the Client. The Client can then display it in the UI.

FIG. **76** illustrates the present invention using a method **7600** wherein a globally addressable interface is used to obtain data from a server. The steps associated with the method **7600** of FIG. **76** will now be set forth.

Collaborations

5. The Client asks the Browsing Interface Proxy for the data associated with customer **1234**.

6. The Browsing Interface Proxy forwards the request across the network to the Browsing Interface.

7. Same as 6.

8. The request is forwarded to the Customer Server. The Customer Server requests the customer data from the Database.

9. The Database returns the customer data for Customer **1234**.

SAP - Exhibit 1105
SAP v. Valtrus et al. - IPR2025-00420
Page 232 of 281

10. The Customer Server creates a structure and populates it with the customer data.

11. The Customer Structure is forwarded through the Browsing Interface, across the network and back to the Browsing Interface Proxy.

12. The Browsing Interface Proxy forwards the Customer Structure to the Client. The Client can now display the data in a UI for a user.

IDL Interfaces and Structures

The following IDL defines the two Interfaces and Structures used in the message trace diagram s above.

```
module CustomerServer
{
// CORBA IDL for the Update Interface
    interface CustomerUpdateInterface
    {
        void changeCustomer( long anId,
commonDefs::CustomerStructure aCustomer);
        void changeAddress(long anId,
commonDefs::AddressStructure aNewAddress);
        string  changePhoneNumber(long anId, string
aNewPhoneNumber);
    };
    // CORBA IDL for the Browsing Interface
    interface CustomerBrowsingInterface
    {
        commonDefs::CustomerStructure    getCustomer(long anId);
        commonDefs::AddressStructure     getAddress(long anId);
        string  getPhoneNumber(long anId);
    };
};
// This module defines the structures passed through the two // customer
interfaces.
module commonDefs
{
    struct AddressStructure
    {
        string  street;
        string  city;
        string  state;
        string  zip;
        string  phoneNumber;
    };
    struct CustomerStructure
    {
        string  id;
        string  status;
        string  firstName;
        string  lastName;
    };
};
```

Sample Code

The following is some Sample Java code for the Skeleton portion.

```
//Pass all requests on to the Component for processing
public CustomerStructure getCustomer(
String aCustomerId)
{
    CustomerComponent aCustomerComp = this.getComponent( );
    return(aCustomerComp.getCustomer(aCustomerId));
    }
//Pass all requests on to the Component for processing
public String getPhone(String aCustomerId)
{
    .
    .
    .
}
```

The next snippet of code is for the Customer Component (Server). The interface delegates the processing to the component.

```
// Get the Customer's data and return it.
public CustomerStructure getCustomer(String aCustomerId)
{
    // Go to the database and get the
    // customer with the appropriate ID
    Customer aCustomer = ...
    // Create a structure and populate it with the
    // customer data retrieved from the DB.
    CustomerStructure aCustomerStructure   = new
CustomerStructure( );
    aCustomerStructure.id                  = aCustomer.getId( );
    aCustomerStructure.status    = aCustomer.getStatus( );
    aCustomerStructure.firstName           =
aCustomer.getFirstName( );
    aCustomerStructure.lastName            =
aCustomer.getLastName( );
    return (aCustomerStructure);
}
public String getPhone(String aCustomerId)
{
    .
    .
}
public AddressStructure getAddress(String aCustomerId)
{
    .
    .
}
```

Additional Considerations

The GAI class is actually represented by two different classes (and the Component itself). Each GAI is made up of a Proxy and a Skeleton. The Proxy represents the interface on a Client while the Skeleton represents the interface on a Server.

Collaborations

Proxy—The proxy pattern is generally used to communicate from a Client to a Globally Addressable Interface on a Server.

Structure Based Communication—Often times, a client needs to display data in a UI for a user (e.g. Customer Information, Order Information, etc.). When communicating through a Globally Addressable Interface, this data is transmitted from the Server to the Client using Structure Based Communication.

Load Balancing—When a number of servers implement the same Globally Addressable Interface, the Load Balancing pattern is used to balance Client requests between these Servers.

Proxy Pool—The Proxy Pool pattern helps balance the cost of instantiating Remote Proxies and retaining Proxy "freshness." The Proxy Pool pattern can be used to create a pool of Proxies to Globally Addressable Interfaces.

Locally Addressable Interface—Locally Addressable Interfaces are private interfaces that aren't easily located. Generally, a well-known interface (like a Globally Addressable Interface) is used to find a LAI. A Client easily find and access a service on a Globally Addressable Interfaces and request a reference to a Locally Addressable Interface in return

Interface—The Interface pattern defines methods or functions or services rather than implementation. The Interface pattern is expanded upon by the Globally Addressable Interface pattern.

Naming—The Naming pattern describes a pattern for registering and finding services or objects etc. where they can be easily found in an application. The Naming pattern is

often used to register Globally Addressable Interfaces so they are publicly available

Alternatives

Locally Addressable Interface—The Locally Addressable Interface pattern is both a collaborating and alternative pattern. It can be used to retrieve information from Servers instead of Globally Addressable Interface.

Legacy Wrapper

FIG. **77** illustrates a flowchart for a method **7700** for affording access to a legacy system. A plurality of components coupled to a client via a component integration architecture are provided for servicing the client in operation **7702**. A legacy system is interconnected to the client via the integration architecture using a legacy wrapper in operation **7704**. In operation **7706**, the legacy system and the client are interfaced via the legacy wrapper by communicating with the client by way of a first protocol and by communicating with the legacy system by way of a second protocol.

As an option, the legacy wrapper may include a legacy wrapper component coupled to a component adapter which, in turn, may be coupled to a legacy adapter via a legacy integration architecture. In this aspect, the legacy adapter may also coupled to the legacy system. As another option, the component adapter may also reformat call parameters of the message into an acceptable format for the legacy system.

As an additional option for this aspect, the legacy wrapper component may also include a pure legacy wrapper component. As even a further option to this aspect, the legacy wrapper component may include a hybrid legacy wrapper component. Also, the interfacing may further include: sending a message from the client to the legacy wrapper component via the component integration architecture; sending the message via the component adapter to the legacy integration architecture; forwarding the message to the legacy adapter; formatting the message to match an application program interface (API) of the legacy system; executing calls on the legacy system based on the formatted message; executing function of the calls and returning results to the legacy adapter, legacy integration architecture, component adapter, and legacy wrapper component which reformats the results; and forwarding the reformatted results to the client via the component integration architecture.

Legacy systems pose a unique situation for developers of component-based solutions. Commonly hosted on mainframes, Legacy Systems often communicate through proprietary protocols, have no standard data or process APIs and don't integrate easily with component based systems. How does a developer access a Legacy System in a component-based solution?

A legacy system is an existing system that does not conform to the technology, architecture and standards of the current project. A large IBM 3090 Mainframe running programs to calculate automobile insurance rates is an example of a Legacy System. It is large, very important to an insurance company, and runs on older, proprietary hardware.

Legacy Systems generally utilize different communication protocols than those used by newly developed component systems. As a result, communicating between a newly developed component and a Legacy System is very difficult.

FIG. **78** depicts the communication difficulties associated with Legacy Systems **7800** attempting to communicate with a client **7802** via a component integration architecture **7804**. The newly developed application (client and components) communicates through a different protocol than the existing Legacy System. FIG. **78** illustrates heterogeneous Interfaces from Components.

Legacy Systems are critical to an organization and usually represent a significant investment. They are tightly controlled to reduce the incidence of system failures and clients may be unwilling or unable to replace these older systems.

New applications could be developed on the mainframe system, however, this generally is not considered strategic and takes a lot of time and effort. Organizations want to add new functionality (new processes) without investing in the old legacy system.

As a result, the current Legacy Systems represent significant investments, are often crucial to a business and aren't easily replaced. Investing in new Legacy applications isn't practical or strategic and Legacy Systems can't communicate with newer componentized systems.

Therefore, the component-based solution should use a Legacy Wrapper to communicate with the existing Legacy Systems. The Legacy Wrapper is a component built to adapt the front end of a legacy system to the rest of the component-based solution.

This solution encapsulates the concerns of a Legacy System away from the new application. It allows other components in the solution to communicate with the legacy component in the exact same manner as the rest of the component-based solution. Further, this solution can also be used to partition the existing Legacy System functionality. FIG. **79** illustrates homogenous interfaces from components **7900** which rectify the problems with Legacy Systems **7901** attempting to communicate with a client **7902** via a component integration architecture **7904**.

Benefits

- Reuse. The Legacy Wrapper pattern allows reuse of an existing Legacy System. New component applications can be developed that leverage the rich store of business processes and data that already exist on the Legacy System.
- Migration. Allows for slower migration of functionality from the Mainframe to components. By continuing to use the functionality of the existing legacy system, the immediate need to build the same functionality in a pure component-based solution is lessened.
- Encapsulation. Provides a separation of concerns between the new system and the Legacy System. By encapsulating the Legacy System, the impact of host changes is largely limited to the Legacy Wrapper.

The implementation of Legacy Wrapper is usually very specific to the type of Legacy System it is integrating. The implementation in this section attempts to give a high level overview of the components of typical legacy systems.

FIG. **80** shows how a Legacy Component is integrated into a component-based model **8000**.

The upper part of FIG. **80** depicts the main units **8002** of a component-based solution. The lower part of the picture depicts the Legacy Component **8004** in greater details.

The following is a description of the participants in the upper portion of FIG. **80**.

- The Client **(8006)** is the application running on the user's machine. It is responsible for UI presentation, local business objects, and communication using client resident proxies.

The Component Integration Architecture **(8008)** is the component that allows clients to communicate and remotely invoke functions on the server components. Typically this is based on some middleware standard (e.g., CORBA or MTS).

The Components **(8010)** in this FIG. **80** represents the server components. These are the business entity components and the business process components. They are invoked from the Client via client proxies.

SAP - Exhibit 1105
SAP v. Valtrus et al. - IPR2025-00420
Page 234 of 281

From the outside, the Legacy Component **8004** looks identical to any other component. However, internally is performs a very specialized function.

The lower part of FIG. ure **80** expands the Legacy Component **8004**. The expansion shows the individual elements, which comprise the Legacy Component **8004**. These elements are:

- The Legacy Wrapper Component **(8012)** is responsible for presenting the same functionality provided by the legacy system to the rest of the component-based solution. Other components of the new component-based solution will interact and communicate with this component. Although this component wraps the existing legacy system, it should behave as any other component in the newer solution.
- The Component Adapter **(8014)** is a custom component responsible for the translation from the Legacy Wrapper Component to the particular implementation of the Legacy Integration Architecture.
- The Legacy Integration Architecture **(8016)** is responsible for sending and receiving messages between the server and host machines. This architecture is usually based on some specific communication implementation. Examples of this include message queues and common databases accessible by both legacy systems and component-based solutions.
- The Legacy Adapter **(8018)** is a custom component responsible for translation from the particular implementation of the Legacy Integration Architecture to the Legacy System.
- The Legacy System **(8020)** is the existing system that will be accessed by the newer component-based solution. Changes to the Legacy System should be minimized when accommodating the new component-based solution.

The application on the host is responsible for translating messages between the Legacy Integration Architecture and the Legacy System. For example, the application must know how to format calls to CICS appropriately, as well as interpret results and reformat them in a way appropriate for the Legacy Wrapper server component.

The degree to which the wrapper components are specialized to partition the functionality of the existing legacy system can vary.

Pure Legacy Wrapper Component

One type is the Pure Legacy Wrapper Component. This component simply adapts the legacy system to the new component-based solution. No new business processes are added. The interface methods on the Legacy Wrapper Component "pass through" to the legacy system, as shown in FIG. **81**. FIG. **81** illustrates Legacy Wrapper Components of a Pure Legacy Wrapper Component including a Legacy Wrapper Component **8112**, a Component Adapter **8114**, a Legacy Integration Architecture **8116**, a Legacy Adapter **8118**, and a Legacy System **8120**.

Hybrid Legacy Wrapper Component

Another type of Legacy Wrapper Component is the Hybrid component. FIG. **82** illustrates a Hybrid Component type of Legacy Wrapper Component. As shown, the hybrid includes a Legacy Wrapper Component **8212**, a Component Adapter **8214**, a Legacy Integration Architecture **8216**, a Legacy Adapter **8218**, and a Legacy System **8220**.

It is a mix of legacy system adapter and some new business processes built in a single component. Some of the interfaces **8222** of the wrapper component **8212** "pass through" to the legacy system, while other interfaces communicate with objects, which may in turn call the legacy system.

There are potentially more variations, including use of an Event Service to allow the mainframe to initiate work from the wrapper components.

Example

FIG. **83** shows an abstract example of the control flow in a Legacy Component. Although, the example is at a very high level, it should provide some insight as to how the Legacy Component functions and how it invokes work on the legacy system.

From the example of FIG. **83**, the following steps are shown:

1. The Client component wants to invoke some functionality, which is located on the legacy system. The Client sends a message via the Component Integration Architecture (e.g. ORB) on the way to the Legacy Wrapper Component.
2. The Component Integration Architecture (e.g. ORB) forwards the call to the appropriate Legacy Wrapper Component.
3. The Legacy Wrapper Component sends the call via the Component Adapter to the Legacy Integration Architecture. When necessary, the Component Adapter reformats the call parameters into an acceptable format for the Legacy System.
4. The Legacy Integration Architecture receives a call for the host-based Legacy application and forwards it to the Legacy Adapter.
5. The Legacy Adapter receives the message from the Legacy Integration Architecture and formats it to match the API of the Legacy System. It makes the appropriate calls on the Legacy System. The Legacy System executes the function and returns the results to the Legacy Adapter
6. The Legacy Adapter receives the results and returns them to the Legacy Integration Architecture.
7. The Legacy Integration Architecture receives the result and forwards it to the Legacy Wrapper Server Component through the Component Adapter.
8. The Legacy Wrapper Component receives the result, reformats the parameters for the component system and forwards it to the Component Integration Architecture.
9. Finally, Component Interaction Architecture receives the result and forwards it to the Client

Collaborations

Message Queued Legacy Integration is a specific implementation of this pattern. It uses message queues as the legacy integration architecture.

Adapter (from the Gamma book Design Patterns) describes at a more abstract level how to convert the interface of a class into another interface that clients expect.

Proxy—This pattern is documented in Design Patterns by Gamma, Helm, Johnson and Vlissides. The proxy pattern is often used to communicate with server components in a distributed environment. The Proxy would be used to communicate across the Component Integration Architecture to a Legacy Wrapper.

Alternatives

Screen Scraping is a more specialized version of legacy wrapping. It describes how to convert a user interface to that of the server (i.e., the legacy system in this case). In this solution, the host-based application generates 3270 type screens and then passes them to CICS. The advantage of this

solution is that it is non-invasive to CICS and reacts as if it were just another terminal interacting with CICS. This may be necessary with legacy systems which must be leveraged, but can not be modified and provide no common API set.

Locally Addressable Interface

FIG. **84** illustrates a flowchart for a method **8400** for for delivering service via a locally addressable interface. In operation **8402**, a plurality of globally addressable interfaces and a plurality of locally addressable interfaces are provided. Access is allowed to a plurality of different sets of services from each of the globally addressable interfaces and the locally addressable interface in operation **8404**. Each interface has a unique set of services associated therewith. In operation **8406**, the globally addressable interfaces are registered in a naming service for facilitating access thereto. Use of the locally addressable interfaces is permitted only via the globally addressable interfaces or another locally addressable interface in operation **8408**.

In an option, the use of the locally addressable interfaces may be facilitated by structured-based communication. As another option, the access may be allowed via a customer interface proxy, a customer server and a database of the globally addressable interface.

In one embodiment, a request may be received by the customer interface proxy for a reference to one of the locally addressable interfaces. The request may then be forwarded across a network to the database of a server of the globally addressable interface. Also, data from the database may be returned in response to the request. Additionally, an object may be instantiated and populated it with the data by the server of the globally addressable interface. The object may also be associated with one of the locally addressable interfaces. Also, the locally addressable interface may be forwarded to the globally addressable interface. As even a further option, a reference may be forwarded to the locally addressable interface across the network and to the customer interface proxy. In addition, the use of the customer interface proxy may be also used to access the locally addressable interface across the network.

In a client-server environment, a client makes requests of Services on a Server. In such an environment, how might a Server expose its services for use to a client in a tightly controlled manner?

Quite often a component wants tight control over the visibility of its interfaces or does not have a need to make its interfaces widely available. Examples of such situations include:

- Security—A component may provide multiple interfaces, some of which have sensitive operations that should not be exposed to all clients. For example, an insurance company's customer service desktop application gets full access to all interfaces and services on a Customer component, but an Independent Agent application has restricted access to services.
- Interface Design—From a design standpoint it may make sense to limit access to some interfaces. For example, a system operations interface might allow clients to query Server components for the number of requests being serviced, or disable future requests on a particular Server. In this type of situation, it's best to limit access to the appropriate user group. In this case, the operations tools specifically designed for administering a system.
- Large number of interfaces—If the component design calls for a large number of interface instances (objects), then it would be detrimental to use the GAI pattern. The sheer number of interfaces could overcrowd and overburden the Naming or Trader service. The Naming or Trader service would slow down as it searched its large list of entries. Additionally, the system would slow down as every client attempted to access the Naming or Trader service for every interface.

Thus, it's sometimes best to keep interfaces with limited appeal out of a Naming or Trader Service.

- No need—If a particular interface or service only has one client, why bother registering it globally? It doesn't make sense and causes additional administration.

FIG. **85** illustrates Problems with Globally Addressable Interfaces in a system **8500** including clients **8502** and servers **8504** with a plurality of interfaces **8506**.

The last couple of points are quite common for stateful components. The above samples clearly do not call for the GAI pattern—an alternative manner of making interfaces available to clients is required.

Therefore, the Locally Addressable Interface pattern should be used to control access to interfaces in an efficient manner.

FIG. **86** illustrates the manner in which the present invention uses a Locally Addressable Interface **8600** to hide functionality and lessen the load on the Naming or Trading Service **8602**.

All components maintain a Globally Addressable Interface **8604** This interface is registered with a Naming or Trader service **8602** and can have any of its services accessed by any client on the network. The services on a GAI **8604** are generally stateless and potentially shared by many clients.

Locally Addressable Interfaces **8600** are not registered with a Naming or Trading service **8602** and can only be obtained through a Globally Addressable Interface **8604** or another Locally Addressable Interface **8600**.

FIG. **87** illustrates the manner in which the present invention obtains a Locally Addressable Interface **8700**.

Globally Addressable Interface **8702** services typically are used to obtain Locally Addressable Interfaces **8700** by providing some key information to the service, trigger global changes to all of the component's member objects, or to obtain component-maintained data that is not represented by a Locally Addressable Interface **8700**.

It is important to note that member business objects are never directly exposed to the client but, rather, communicated with through a component interface (global or local). This allows for changes to be made to the internal structure of the component without disturbing the way a client interfaces with the component. Encapsulation is preserved.

Benefits

- Tight control. Servers providing LAIs have full control to determine which clients will receive them. This control could, for example, be based on client type, access rights or server load.
- No central bottleneck. The pattern does not rely on a centralized service to hand out interfaces. This leads to a scalable architecture that can handle many interface instances.
- Useful for stateful components. Stateful components often contain many objects, each accessed through a separate interface instance. The LAI pattern is very useful in such circumstances.
- Complex server side relationships. The LAI pattern is better for managing complex object relationships than most alternatives. If an object is associated with a lot of other objects (an order holds a customer and an address and a line item etc.), it isn't practical to copy all of the objects to the client.

The following is a message trace diagram depicting the interactions associated with a Locally Addressable Interface.

The Message Trace diagrams depict a common Client-Server scenario. The Client would like to interact with a specific Customer on the Server. The client requests a Locally Addressable Interface to a Customer Object on the Server and communicates with that object.

The scenario was broken into two message trace diagrams. The first message trace sets the stage for the second. In the first message trace, the Server registers a Globally Addressable Interface with the Naming Service. The Globally Addressable Interface will be used to get the Locally Addressable Interface.

Assumptions

CORBA ORB connects Client and Server

CORBA Naming Service used to lookup GAIs

FIG. **88** illustrates the method in which the present invention registers and then locates a Globally Addressable Interface **8800**. The various steps shown in FIG. **88** are set forth hereinbelow.

Collaborations

1a. "Bind" the interface name (Customer Interface) with it's Remote Object Reference (network location) in a Naming Service. This will allow clients to "lookup" the interface. Once the Interface is registered in the Naming Service, it has become globally addressable. Any client can find the interface and access a operation.

2. The client instantiates a Proxy (Customer Interface Proxy) to the Customer Interface on the Customer Server.

3. The Proxy "looks up" the network location of the Customer Interface. It makes a request of the Naming Service. It requests the network location of the Customer Interface.

4. The Naming Service returns the Remote Object Reference (network location) for the Customer Interface. The Proxy now has all the information it needs to access an operation on the Customer Interface.

The second message trace builds upon the first. In this message trace diagram, the Client calls the Server through a Globally Addressable Interface. The server finds the appropriate customer data and instantiates an object with the data. A Locally Addressable Interface to the specific Customer object is then returned to the Client.

The Client can then directly access the specific Client through the Locally Addressable Interface.

FIG. **89** ilustrates the manner in which the present invention uses a Globally Addressable Interface **8900** to obtain a Locally Addressable Interface **8902** to a specific Customer Object **8904**. Note the steps set forth below.

Collaborations

5. The Client asks the Customer Interface Proxy for a reference to a Locally Addressable Interface for Cusomer **1234**.

6. The Customer Interface Proxy forwards the request across the network to the Customer Interface.

7. The request is forwarded to the Customer Server. The Customer Server requests the customer data from the Database.

8. The Database returns the customer data for Customer **1234**.

9. The Customer Server creates instantiates an object and populates it with the customer data. The Customer object is associated with a Locally Addressable Interface (Update Interface).

10. The Locally Addressable Interface is forwarded to the Customer Interface.

11. The Customer Interface forwards a reference to the Locally Addressable Interface, across the network and back to the Customer Interface Proxy.

12. The Customer Interface Proxy instantiates an Update Interface Proxy with the reference to the Update Interface.

13. The Customer Interface Proxy forwards the Update Interface Proxy to the Client.

14. The Client sends a new address for the customer to the Update Interface Proxy.

15. The Update Interface Proxy forwards the information across the network to the Update Interface.

16. The Update Interface forwards the new address to the Customer Object. The Customer Object updates its address based upon the new information.

Collaborations

Proxy—The proxy pattern is generally used to communicate from a Client to a Locally Addressable Interface on a Server.

Interface—The Interface pattern defines methods or functions or services rather than implementation. The Interface pattern is expanded upon by the Locally Addressable Interface pattern.

Globally Addressable Interface—Locally Addressable Interfaces are private interfaces that aren't easily located. Generally, a well-known interface (like a Globally Address able Interface) is used to find a LAI. A Client can easily find and access a service on a Globally Addressable Interface and request a reference to a Locally Addressable Interface in return

Structured Based Communications—Often times, a client needs to display data in a UI for a user (e.g. Customer Information, Order Information, etc.). When communicating through a Locally Addressable Interface, this data is transmitted from the Server to the Client using Structure Based Communication.

Alternatives

Globally Addressable Interface. The Globally Addressable Interface pattern is both a collaborating and alternative pattern. It can be used to retrieve information from Servers instead of Locally Addressable Interface—the right choice will depend on the context.

Null Structure

FIG. **90** illustrates a flowchart for a method **9000** for communicating a null value. A query is first communicated in operation **9002** from a first system to a second system to determine whether a data structure is a null value. Next, in operation **9004**, a response to the query is received from the second system indicating whether the data structure is a null value. A request for the data structure is sent from the first system to the second system in operation **9006** only if the response indicates that the data structure is not a null value. Subsequently, the data structure is received from the second system in operation **9008**.

As one option, the response may be a Boolean indication. As another option, the response may be determined based on an attribute of the data structure. As a further option, the data structure may represent a set of a plurality of values. Also, the first system may, optionally, be a client and the second system is a server.

When transmitting data across a network between a client and server application, the middleware's "type system" does not always support null values. How can a remote service send or receive null values over a communications medium that does not support them?

It is expected that distributed Business Components will collaborate with other Business Components via some sort of communications medium. Communications between components is not usually handled by the components

SAP - Exhibit 1105
SAP v. Valtrus et al. - IPR2025-00420
Page 237 of 281

themselves but rather by some communications middleware (like an object request broker, or ORB).

A "null" value is a frequently used value in object-based systems. A "null" represents the empty set. It is often returned from a service that is unable to find the requested elements or is used as an optional parameter in a distributed service. For example, a Client might request all the customers with a last name of "Smith." If no Customers exist with a last name of "Smith," a "null" value would be returned.

Some legacy systems return −999 or 0 when no data exists. This is not an ideal solution as the system is using data to represent non-data. What if −999 or 0 are valid responses to a request? Instead, a "null" could be used to better represent this case. A "null" value provides extra flexibility since a specific data value need not be reserved to represent the empty set.

However, middleware cannot represent every data type that exists in every language. Since middleware is "language neutral", it can only represent the least common denominator of every language accessible via the middleware. Due to this constraint, "null's" often can not be represented in middleware. FIG. **91** illustrates the problem associated with sending a NULL across many types of middleware **9100**.

A system should be able to take advantage of this important value and use middleware that may not support it.

Therefore, use the Null Structure pattern to pass a structure with an isNull attribute across the middleware. Unlike a "null", a structure can be passed across the middleware.

FIG. **92** ilustrates the manner in which the present invention passes a "null" structure across the middleware **9200**.

The extra attribute on the structure then determines whether or not the structure represents a "null" value. The structure can be queried to determine whether or not it represents a "null" value. FIG. **93** depicts conversations **9300** with a "null" data structure **9302**. FIG. **94** depicts conversations **9400** with a non-"null" data structure **9402**.

The isNull attribute could be added as shown in the IDL example below.

```
structure contract
{
    boolean isNull;
    long buyerIdentifier;
    long sellerIdentifier;
    double rate;
};
```

Benefits

Flexibility. This pattern allows for "null" values to be utilized by distributed components.

The following example assumes a CORBA implementation. In order to pass Null Structures across an ORB, a structure must be defined in the ORB's interface definition language (IDL). The following IDL defines a structure that will represent an Integer or a "null."

```
struct CommonInteger
{
    long    value;
    boolean isNull;
};
```

In the code that prepares the data to be sent over the ORB, a check of the data is made and the structure is populated appropriately. If it is null, the isNull flag is set, otherwise it is cleared. Refer to the following code example:

```
public CommonInteger convertIntegerForORB(Integer anInteger)
{
    CommonInteger integerStructure = new CommonInteger( );
    if(anInteger == null)
    {
        integerStructure.isNull = true;
    }
    else
    {
        integerStructure.isNull = false;
        integerStructure.value = anInteger.intValue( );
    }
    return integerStructure;
}
```

The receiving code that obtains the data from the ORB does the same conversion in reverse as shown in the method below:

```
public Integer convertIntegerFromORB(CommonInteger anIntegerStructure)
{
    Integer anInteger = null;
    if(anIntegerStructure.isNull = false)    // structure not null
    {
        anInteger = new Integer(anIntegerStructure.value);
    }
    return anInteger;
}
```

Collaborations

Proxy. A Proxy is a placeholder that can accept requests meant for another object. This is typically used in distributed systems when one component wants to send a request to another. Thus, a proxy is often used to make requests of servers that may return null structures.

Client-Server. Client-Server is a type of architecture that separates the Client portion of an application from the business logic or database portion of an application. When implementing a Client-Server application, the Client and Server often communicate across a middleware (like CORBA) that doesn't support "nulls."

Alternatives

Invalid Value. Determine an "invalid value" for each data type in the particular application. Return the "invalid value" when ever a null should be returned.

Paging Communication

FIG. **95** illustrates a flowchart for a method **9500** for transmitting data from a server to a client via pages. In operation **9502**, pages of data sets are built from data in a

SAP - Exhibit 1105
SAP v. Valtrus et al. - IPR2025-00420
Page 238 of 281

database of a server. Upon receipt of a first request from a client for the data in the database of the server in operation **9504**, a first one of the pages of the data sets is sent to the client over a network in response to the first request in operation **9506**. When a second request from the client for the data in the database of the server is received in operation **9508**, a second one of the pages of the data sets is then transmitted to the client over the network in response to the second request in operation **9510**.

The second request may be sent to the server with an identifier of a last entry of the first page. Also, a size of the data sets of the pages may be defined dynamically. As an option, the pages may be displayed by the client upon receipt from the server. Also, a size of the data sets of each of the pages may be determined based on a user interface of the client. As another option, a size of the data sets of each of the pages may be determined based on an amount of data capable of being displayed at once by the client.

In a client-server environment, a client often needs to display or process a long list of data. Finding and transmitting this list of data can take a long time and negatively impact the user's response time. How can a client and server interact to improve the user's response time when retrieving a large list of data?

The speed with which a UI can respond to a "user initiated" request is important. This is generally called the UI response time and is an important attribute of every application. FIG. **96** depicts the response time **9600** for a User Interface **9602** to display a list of customers in a list box **9604**.

Users expect an "acceptable" level of UI response in their applications. Applications that don't meet this criteria, will not be successful.

Many UIs allow users to query databases for lists of data. In FIG. **96**, for example, the user clicks the "Get Customers" button to initiate a database query. The query will retrieve every customer from the database and the UI will display the customers in a list box. The user can then scroll through the data and select a particular entry for further investigation.

FIG. **97** shows a request that returns a large amount of data. As shown, in a three-tiered client-server environment, each query must travel from the client UI **9700**, across a network **9702**, to a Server **9704**, and eventually to a Database **9706**. Then, the result of the query must travel all the way back to the client.

When the query results in a large amount of data, the time to search the database and return the data across a network can become prohibitive. As a result, the UI response time will quickly degrade to an unacceptable level.

To make things worse, the average user only looks at half of the data returned from the database. The user is just as likely to find their data in the first half of the list as the second half of the list. As a result, the user may wait a long time for data that is not used.

FIG. **98** shows a graphical depiction of a paging communication pattern **9800**.

Therefore, provide Paging Communication between the client and server tiers of an application. Paging Communication describes a pattern for transmitting a large amount of data while maintaining an acceptable UI response level.

Rather than send all of the data at one time, a subset or "page" **9802** of data is transmitted. When the client needs more data, another "page" **9802** of data is transmitted. This continues until the client has seen enough data or all the data has been transmitted.

Benefits

More Responsive UI. This pattern improves upon the user's response time. The server only retrieves and transmits a "page" of data at a time. This is a lot faster than retrieving and transmitting all of the data at one time. The pattern breaks-up the total search and transmission time into smaller page-sized chunks. This greatly improves upon the user's perceived performance.

Additionally, the Server searches the database for a "page" of data at a time until the user finds what they are looking for. As a result, unless the needed data is in the last page of data, the search is limited to a portion of the total search.

Configurable Page Size. The page size can be "tuned" to best fit the application. As a result, the page size can be altered to best fit a particular network, application design, etc.

Stateless Servers. The paging mechanism can be managed from the client-side requestor. Thus, this pattern can be used with stateless servers just as easily as with stateful servers.

UI Tunable. The page size can be changed to match a particular User Interface.

List Box Friendly. A list box can only display a limited amount of data at one time. As a result, it isn't as important to have all of the data immediately available for the list box. The List box can display a page of data, and then request additional pages of data as the user scrolls through the list.

Scenario: A user is searching for a particular customer. The user doesn't remember the exact name of the customer, but the user believes they will recognize the name when they see it. Thus, the user requests a list of all customers.

Technical Parameters:

Static Page Size=4

List Box can only display 2 lines of data at a time.

FIG. **99** illustrates a message trace diagram showing the interactions between a Client **9900** and a Server **9902** using Paging Communication to satisfy the previously mentioned scenario.

Definitions

Starting Key

The Starting Key is the initial starting point for the search., The database will begin searching for data (customers in the message trace above) at the Starting Key. An example starting key could be "A*".

Last Found Key

The Last Found Key is used to request subsequent pages of data from the Server and the database. The "last found key" defines the starting point for the next data request. The Server will begin searching for data at the "last found key" and continue until it has retrieved a full "page" of information.

When all of the data has been retrieved from the Server and Database, the Last Found Key is left blank. This notifies the Client that all the data has been sent.

Intermediate Page

An intermediate "page" is returned for every request but the last. When a client receives an intermediate page and a "last found key", the client knows more "pages" of data exist on the server.

In order to obtain an intermediate "page," a "last found key" must be passed from the client to the server. When the Server has retrieved a full "page" of data, the new "last

found key" is saved. It is then passed back with the intermediate "page." The new "last found key" defines the starting point for the next data request.

Last Page

When the Server has retrieved all of the data meeting the search criteria, the Server builds the last "page." When the last page is returned to the client, the "last found key" is left blank. This notifies the client the search is complete and no more data matching the search exists on the Server. Note that the last page is usually smaller than the other pages.

Empty Page

When no data are selected from the search criteria, the server builds an empty page signaling to the client no more data exist on the server.

Static or Dynamic Page Size

The page size can be defined statically or dynamically. The message trace diagram in FIG. **99** depicts a static page size.

If you'd like a dynamic page size, the client must pass an additional parameter with each request to the Server. The additional parameter would be the page size.

The steps associated with FIG. **99** will now be set forth.

Collaborations

1. The user "clicks" the "Get Customers" button on the User Interface. The Client UI makes a getAllCustomers request of the Server and passes a Starting Key as a parameter. Since the user wants to view all of the customers, a Starting Key of spaces is used. Message sent=getAllCustomers(" ");
2. The Server receives the request from the Client. The Server realizes the Starting Key is blank and knows this is a new request. Thus, the Server requests first four customers (the page size) from the database.
3. The database returns the first four customers (Albert Abraham, Ned Abraham, Sally Abraham and Alice Allen) and a "Last Found Key" ("Alice Allen") to the Server. The "Last Found Key" denotes the last entry found during the search. It will be used for subsequent searches.
4. The Server builds a page with the four customers retrieved from the database. The Server returns the page and the Last Found Key to the Client.
   Page Type=Intermediate
   Page="Albert Abraham", "Ned Abraham", "Sally Abraham" & "Alice Allen"
   lastFoundKey="Alice Allen"

The Client receives the "page" of data. The Client sends the data to a UI List Box for viewing by the user. The User can see the first two customers (Albert Abraham, Ned Abraham).

The User clicks the "scroll down" arrow twice and can now see two additional customer (Sally Abraham, Alice Allen).

5. The User clicks the "scroll down" arrow again. No more data exists on the Client so the Client must request another page from the server. The Client UI makes a getAllCustomers request of the Server and passes the Last Found Key of Alice Allen. Message sent=getAllCustomers("Alice Allen");
6. The Server receives the request from the Client. The Server requests the next four customers (page size) after Alice Allen. Message sent=getPageOfcustomer ("Alice Allen")
7. The database returns the next four customers (Jason Allen, Fred Allen, Sam Allen & Zack Allen) and a "Last Found Key" ("Zack Allen") to the Server.
8. The Server builds a page with the four customers retrieved form the database. The Server returns the page and the Last Found Key to the Client.
   Page Type=Intermediate
   Page="Jason Allen", "Fred Allen", "Same Allen" & "Zack Allen"
   lastFoundKey="Zack Allen"

The Client receives the "page" of data. The Client sends the data to a UI List Box for viewing by the user. The User can see the first two customers and one new customer (Alice Allen, Jason Allen).

The User can now scroll through the next three customers. When scrolling past customer Zack Allen, the Client will request another page of data from the Server. It will follow the same basic pattern as described in steps 5–9.

Eventually, the end of the list of Customer will be reached

n-**3**. Once again, the client clicks the "scroll down" arrow and no more customers exist on the client. The Client must request another page from the server. The Client UI makes a getAllCustomers request of the Server and passes the Last Found Key of Jim Ziegler. Message sent=getAllCustomers("Jim Ziegler");

n-**2**. The Server receives the request from the Client. The Server requests the next four customers (page size) after Jim Ziegler. Message sent=getPageOfcustomer ("Jim Ziegler")

n-**1**. The database can only find two more customers. The database returns the final two customers (Sam Ziegler and Ziggy Ziegler) and no Last Found Key.

n. The Server builds a page with the two remaining customers retrieved from the database. The Server returns the page and the blank Last Found Key to the Client.
   Page Type=Last Page
   Page="Sam Ziegler", "Ziggy Ziegler"
   lastFoundKey=" "

The Client receives the final "page" of data. The Client sends the data to a UI List Box for viewing by the user. The User can see the following two customers (Jim Ziegler, Sam Ziegler).

The User clicks the "scroll down" arrow once and can now see the final two customers (Sam Ziegler, Ziggy Ziegler) in the List Box.

Subsequent "clicks" on the scroll down arrow no longer request data from the Server. The Client knows (due to the blank last found key) that it has already received all of the available data.

Additional Details

Context isn't generally stored on the Server when implementing Paging Communication. As a result, it is important to request a minimum collection of data from the server. Most of the relational database are using a count mechanism that defines the maximum number of data to search. That will minimize CPU and memory usage.

As explained in the example, page size may be adapted to the client requirements, however that does not mean the page size must exactly fit the widget size. Ideally the client application will anticipate future user actions and request more than one page.

Collaborations

Proxy—The Proxy pattern is often used to communicate between Clients and Servers in a distributed environment. A Proxy is often used to make requests for a "page" of data from a Server.

Interface Control Model—The ICM pattern addresses the separation of the Interface (Viewing portion) from the Control from the Model (the data portion) in an application.

SAP - Exhibit 1105
SAP v. Valtrus et al. - IPR2025-00420
Page 240 of 281

Paging Communication is often used when implementing this separation of functionality. A user through the Interface uses the pattern to retrieve large lists of data from the Model for viewing.

Globally Addressable Interface—Globally Addressable Interfaces are often used to obtain a Page of data from a Server for display in a Client UI.

Alternatives

Paging with Server Caching—This pattern builds upon the Paging Communication pattern. Rather than querying the database for a "page" of information, the Server would retrieve all of the data at one time. Then the Server would pass the data to the Client one page at a time.

Refreshable Proxy Pool

FIG. **100** illustrates a flowchart for a method **10000** for interfacing a naming service and a client with the naming service allowing access to a plurality of different sets of services from a plurality of globally addressable interfaces. In operation **10002**, the naming service calls for receiving locations of the global addressable interfaces. As a result of the calls, proxies are generated based on the received locations of the global addressable interfaces in operation **10004**. The proxies are received in an allocation queue where the proxies are then allocated in a proxy pool (see operations **10006** and **10008**). Access to the proxies in the proxy pool is allowed for identifying the location of one of the global addressable interfaces in response to a request received from the client in operation **100010**.

The proxy pool may employ load balancing. As another option, the proxies in the proxy pool may be renewed based on an age thereof. As a third option, a handle may interface the proxy pool and the client. This handle may additionally interface a plurality of the proxy pools and the client.

In distributed systems with many clients, it is important to establish connections with remote servers in an efficient manner. In a manner where clients evenly utilize the available servers. How can this be performed in a consistent manner for all clients?

In production systems it is quite common for long-lived clients to "stay up" for days and interact with a collection of different servers. Oftentimes, a client process will establish connections to the same type of server a bunch of times during its lifetime. The lifetimes of the client and its servers are often different as a result of server maintenance and failures. However, such failures should have minimal impact on the client. Clients in such systems usually retrieve a Globally Addressable Interface (GAI) from a naming or Trader Service (see GAI Pattern).

A GAI retrieved by a client will usually go through three phases: 1) Initial retrieval of a GAI from the Trader Service that is subsequently wrapped up in a proxy, 2) Invocations of businesses functions supported by the GAI and 3) Release of the GAI proxy. This often means a long-lived client will repeatedly ask the Trader Service for the same type of interface during its lifetime.

FIG. **101** illustrates repeated requests to the Trader Service **10100** for the same interfaces are neither efficient nor necessary

Repeatedly requesting the same interface from a Trader Service is neither efficient or necessary. However, the Trader Service does maintain load balancing information and allocates the least busy interface at any given time. Thus, some value exists in repeated reallocation.

Therefore, use a Refreshable Proxy Pool mechanism that standardizes the usage, allocation and replenishment of proxies in a client's pool. Initially, the Proxy Pool will allocate a bunch of Proxies to remote services using some sort of a Lookup Service (e.g. Trader Service, Naming Service). The Proxy Pool will hold onto these Proxies and allocate them to Clients as they need them. When the client asks the Proxy Pool for a proxy, the pool will hand out a new Proxy.

FIG. **102** illustrates how a pool **10200** can be created that reuses GAI proxies.

In order to balance the system load evenly, the Proxy Pool should implement a Load Balancing approach (e.g. Round Robin) for handing out proxies within the pool.

Each proxy in the pool has a "retirement age." The "retirement age" determines the time to refresh a given Proxy. When a Proxy reaches its retirement age, it is taken out of the pool and replaced with a freshly allocated Proxy. This ensures the Proxy Pool is refreshed regularly with new GAIs retrieved from the Trader Service. The retirement mechanism helps dynamically balance the systems load.

Benefits

Performance. Establishing connections to servers will take less time because clients will go to the trading service less often.

Balanced Load. This ensures all clients are implementing the same strategy for utilizing available servers.

Standard. One single approach to pooling increases maintainability and predictability across the enterprise and decreases confusion.

Ease of use. Client developers do not have to design and implement their own version of pooling.

Maintenance. This mechanism allows for centralized development. When a bug is found it can be fixed and distributed to all build centers.

Dynamic. Client threads will not have to worry about allocating retired or bad proxies.

Robustness. By pooling GAI location information, clients are less susceptible to Trader Service failure. This is because the Proxy Pool can operate using the GAIs it has information on for as long as those references main valid.

The Proxy Pool should be packaged and distributed to client developers so that it is non-intrusive and easy for them to use.

Parameters such as pool size and retirement age should be configurable.

The client thread using the proxy should not pay a penalty for pooling or allocation.

The pool should recover gracefully from server failure.

The pool should recover gracefully from Trader Service failure. FIG. **103** illustrates the implementation of a Refreshable Proxy Pool **10300**. The Refreshable Proxy Pool is based on a pool-queue approach. In this design, the pool holds allocated proxies **10302** while the queue allocates and replenishes the pool with proxies. To handle the allocation and replenishment, a worker or allocation thread **10304** runs on the queue and makes calls to the Trader Services as needed.

There can be numerous proxy pools, but this implementation supports typed pools Using C++ templates, i.e., each pool will only contain proxies of one type. This allows the client to create a class that is passed to the proxy pool and supports client specific properties in the pool such as pool size, retirement age, etc. Also, due to synchronization issues with the rest of the architecture, there can be only one allocation queue.

Clients who wish to use a pooled proxy will create a handle as a wrapper. This handle wrapper takes care of the problems associated with sharing resources across threads such as lazy initialization, reference counting, allocation, and de-allocation.

Handles are classes that abstract the users away from the implementation. Handles are generally stack based and exist for the lifetime of a method invocation or an object. The handle destructor insures that the underlying proxy is dereferenced.

Suggested Classes

FIG. **104** illustrates the class relationships between the patterns primary classes.

Class	Description
PooledProxy (10402)	This is the base class for the pooled proxy. It actually acts as a wrapper for a Proxy and maintains all usage and reference counting information.
ProxyPool (10404)	This is the proxy pool, where clients go to retrieve a proxy. It should be thread-safe in that multiple threads are automatically synchronized. This pool should only contain valid proxies that have been allocated by the AllocationPool. When a proxy is requested, the usage count is incremented. After the "usage" passes retirement age, the proxy is remove from the pool and placed back into the allocation pool.
AllocationPool (10406)	This is the pool that actually does the proxy allocation. This pool is populated with unallocated proxies and a "reader" thread will allocate them. Since there will only be one, this class should be implemented as a Singleton. This pool however can allocate proxies of any type.
ProxyHandle<T> (10408)	This is the Handle that clients should use to manage pooled proxies. The handles must use a static_cast<T> (C++ template) to retrieve the correct proxy. T is defined by a client template instantiation, and assumes the client knows exactly what type of proxy the pool is actually holding. Clients must take care to assure that pools only contain proxies of one type.

Collaborations

Globally Addressable Interface—This is a pattern for making interfaces publicly available. Distributed connections to Globally Addressable Interfaces can be pooled using the Refreshable Proxy Pooling pattern.

Proxy—This pattern is documented in the book "Design Patterns" by Gamma, Helm, Johnson and Vlissides. The proxy pattern is often used to communicate with server components in a distributed environment. Proxies are pooled using the Refreshable Proxy Pool pattern.

Trader—The Trader service defines how distributed architectures locate components based on the types of services they provide. The allocation queue interacts with a Trader Service to allocate the correct type of proxy.

Naming—The Naming Service provides a mapping between names and object references. A Naming Service could be used to store the GAI references that the Refreshable Proxy Pool pattern requires.

Alternatives

Single Use—As opposed to pooling connections to a remote server a client can request a new connection each time a GAI is needed. This would work best when a client infrequently needs GAIs.

Proxy Pool—This pattern addresses the pooling of proxies without periodic refreshing. It is a simpler version of the Refreshable Proxy Pool that may be of use when server load is fairly constant.

Self-Describing Stream

FIG. **105** illustrates a flowchart for a method **10500** for providing a self-describing stream-based communication system. Messages are sent including data between a sending system and a receiving system in operation **10502**. Meta-data is attached to the messages being sent between the sending system and the receiving system in operation **10504**. The data of the messages sent from the sending system to the receiving system is translated based on the meta-data in operation **10506**. The meta-data includes a first section that identifies a type of object associated with the data and a number of attribute descriptors in the data. Also included is a second section that includes a series of the attribute descriptors defining elements of the data.

As an option, the sending system and receiving system may each be equipped with logic for interpreting the meta-data of the messages. As another option, the elements may be defined in terms of size, type, and name. Versions of the present invention include a version where one of the systems may be an object-based system and one of the systems may be a non-object-based system, a version where both of the systems may be object-based systems, and even a version where both of the systems may be non-object-based systems.

Stream-based communication is a very effective pattern for relaying data, data structures, and meta-data. Meta-data is information about the data, such as data structure, data types, etc. using a shared, generic format. How can the message format be shared between systems so as to create the most flexible stream-based communication mechanism?

Often, it is determined that a stream-based communication mechanism should be used to transport information between systems. Stream-based communication is a pattern where information is transported from one system to another system using a simple stream and a shared format that relays both the data and meta-data information.

FIG. **106** illustrates two systems **10600** communicating via Stream-Based Communication **10602** and using a shared generic format to relay the meta-data information.

However, when iinplementing Stream-based Communication **10602**, a number of factors influence the method for enabling each system with a "shared format." The "shared format" provides the meta-data information needed to interpret the raw data in a stream. This shared format is like a secret decoder ring for systems sending and receiving messages. It allows the systems to convert structured data (objects, strings, etc.) into raw data and raw data back into structured data. This is needed to transmit the structured data across the network.

Many additional factors influence the detailed design of this communication mechanism. Some systems support volatile and constantly changing object models, data models and data structures. In these systems, flexible, de-coupled communication is extremely important.

In a constantly changing system, a statically defined "shared format" doesn't work very well. Every change to the object model, data model of data structure causes a reimplementation of the "shared format." Each reimplementation results in a redesign, recompile, and retest of the changed code

FIG. **107** illustrates an object-based system **10700** with a frequently changing object model **10702** communicating via Stream-Based Communication **10704**.

SAP - Exhibit 1105
SAP v. Valtrus et al. - IPR2025-00420
Page 242 of 281

FIG. **107** depicts a constantly changing system. Initially, the object-based system **10700** is designed to send Poodle objects through a stream to a non-object system **10702**. As time passes, the system requirements change. Now, the object-based **10700** system must send German.Shepherd objects through a stream to the non-object system **10702**. If the "shared format" for converting dog objects to raw data is inflexible, this will break the system.

In cases like this, it would be better to implement a communication mechanism or "shared format" that can better handle changes to the systems.

Therefore, use a Self-Describing Stream and create a stream that contains message data AND descriptive meta-data. Then use a message language to read the formatting information and meta-data off of the stream.

FIG. **108** illustrates a stream-based message that contains both message data **10800** and descriptive meta-data **10802**.

FIG. **108** depicts: a message sent using a Self-Describing Stream. The first 30 bytes contain descriptive meta-data **10802**. This meta-data **10802** describes the formatting of the "real" data **10800** in the remainder of the message. It describes the data type, attribute names, location in the message, etc. of the "real" data **10800** in the message. The remaining 70 bytes are the "real" data **10800** transmitted between the two systems.

Additionally, each system must implement a message language. The message language defines the rules for writing and interpreting the descriptive meta-data. It describes how the meta-data is parametarised and embedded in the message.

These Self-Describing messages usually contain three distinct sections: a generic header, an attribute descriptors section, and a data section. The header portion contains generic information about the message. It contains such information as the type of object, the number of attributes descriptors, the target environments, etc. The attribute descriptors section contains a series of attribute descriptors that define the various data elements of the information. The number of these attribute descriptors is usually defined in the header section. The last section contains only data.

FIG. **109** illustrates the manner in which a message language defines how to parameterise the meta-data **10900** and put it on the stream.

Benefits

Greater Flexibility. Because the information about the structure of the data has been parameterised and stored as additional data within the message, changes to the data structure would have no effect on this interface mechanism. This means the interface mechanisms will not need to be re-designed/re-built/re-tested/re-deployed, etc for each change in meta-data.

Interfacing systems are better de-coupled. Because the message format is embedded in the actual stream, this format does not need to be stored or kept in synch across different systems. It can be "discovered" at run-time when the interface is invoked.

For object-based systems, the implementation is quite straightforward. Simply make each object responsible for implementing streaming behaviors based on the format and message language. Each object should know how to get and parse its attribute values onto a stream as string values (streamOn) and each object class should know how to parse attributes off of a stream and put these values into a new instance of the object (streamOff).

Below is an example of a Self-Describing stream. It is used to stream an object's information from an object-based system to a non-object system.

FIG. **110** illustrates a Customer object **11000** in an object-based system **11002** streaming itself into a stream **11004**, the stream **11004** being sent to a non-object system **11006**, this stream **11004** being read and the data inserted into a relational database **11008**. The steps illustrated in FIG. **110** will now be set forth.

1. The CustomerObject with attributes name, sex, and age has a method "streamOn: aStream." It is invoked with an empty stream as the argument 'aStream'. The CustomerObject "streamOn:" method goes through each of the object's attributes and parses each value as a string onto the stream.

In the Java pseudo-code below, the message language defines the format of the header, the format of the attribute descriptors, and the delimiter used in the parsing.

Note: Assume that "asString( )" converts the receiver to a string and that "padWithSpaces( )" pads the string with spaces and makes the string the length specified.

```
/** Stream my attribute values on aStream **/
public void streamOn (OutputStream aStream)
{
    // CREATE THE HEADER
    aStream.write('CUSTOMER '); // This is a customer object
    aStream.write('003'); // with three attributes
    aStream.write('001'); // this is the format version
    // DESCRIBE EACH ATTRIBUTE
    aStream.write(Stream.Delimiter);
    aStream.write('NAME ');
    aStream.write('STG ');
    aStream.write('010');
    aStream.write(Stream.Delimiter);
    aStream.write('SEX ');
    aStream.write('STG ');
    aStream.write('007');
    aStream.write(Stream.Delimiter);
    aStream.write('AGE ');
    aStream.write('NUM ');
    aStream.write('003');
    // WRITE OUT THE ATTRIBUTE VALUES AS DATA
    aStream.write(Stream.Delimiter);
    aStream.write(this.getName( ).asString( ).padWithSpaces(10));
    aStream.write(this.getSex( ).asString( ).padWithSpaces(7));
    aStream.write(this.getAge( ).asString( ).padWithSpaces(3));
}
```

2. The stream is then put into a message communication mechanism like MQSeries or MessageQ and sent to the non-object system.
3. Once at the non-object system, interface code reads the stream, parses the values off, converts and moves the values into a copybook with the appropriate structure, and saves the information in relational database. A pseudo-COBOL example is listed below. In reality, this interface code would be more dynamic than depicted in this example.

```
...
DATA DIVISION.
FD FILE-STREAM-IN
    RECORD CONTAINS 100 CHARACTERS
...
WORKING-STORAGE SECTION.
01  WS-FILE-STREAM-IN      PIC X(100).
O1  WS-SHARED-FORMAT-HEARDER
O3  WS-HEADER-OBJECT-TYPE              PIC X(10).
03  WS-HEADER-NUM-OF-ATTRIBUTES       PIC X(7).
03  WS-HEADER-VERSION-OF-FORMAT       PIC 999.
O1  WS-SHARED-FORMAT-ATTRIBUTE
```

-continued

```
O3  WS-ATTRIBUTE-NAME      PIC X(5).
03  WS-ATTRIBUTE-TYPE      PIC X(5).
03  WS-ATTRIBUTE-SIZE      PIC 999.
O1  TEMP-VARIABLES
    O3  WS-INDEX           PIC 9999.
...
01  WS-CUSTOMER
03  WS-NAME                PIC X(10).
03  WS-SEX                 PIC X(7).
03  WS-AGE                 PIC 999.
...
88  LT-HEADER-SIZE         PIX 99 VALUE 20.
88  LT-ATTRIBUTE-DESCRIPTOR-SIZE      PIX X(1) VALUE 14.
88  LT-DELIMINATOR         PIX X(1) VALUE "|".
88  LT-STRING              PIX X(1) VALUE "STG ".
88  LT-NUMBER              PIX X(1) VALUE "NUM ".
...
PROCEDURE DIVISION.
...
*** OPEN THE FILE STREAM AND READ IT INTO THE
TEMPORARY ***
*** VARIABLE WS-FILE-STREAM-IN ***
OPEN FILE-STREAM-IN.
READ FILE-STREAM-IN INTO WS-FILE-STREAM-IN
    AT-END CLOSE FILE-STREAM-IN
END-READ.
*** MOVE THE HEADER INFORMATION INTO THE HEADER
COPYBOOK****
MOVE (WS-FILE-STREAM-IN FROM ZERO TO LT-HEADER-SIZE)
    TO WS-SHARED-FORMAT-HEADER.
*** FIND WHAT BYTE THE DATA STARTS AT AND SET THE
INDEX ****
MOVE (LT-ATTRIBUTE-DESCRIPTOR-SIZE * WS-HEADER-NUM-
OF-ATTRIBTES)
TO WS-INDEX.
*** PARSE THE APPROPRIATE OBJECT STRUCTURE OFF ***
*** THE STREAM                                  ***
IF WS-HEADER-OBJECT-TYPE EQUALS "CUSTOMER " THEN
    PERFORM 1000-PARSE-CUSTOMER-STREAM THRU
        1000-PARSE-CUSTOMER-STREAM-END).
ELSE IF WS-HEADER-OBJECT-TYPE EQUALS "EMPLOYEE "
THEN
...
ELSE IF
...
ELSE
*** END THE PROGRAM
RUN-STOP.
END-IF.
1000-PARSE-CUSTOMER-STREAM.
*** READ WHICH VARIABLE IT IS AND POPULATE THE
CORRECT ***
*** VARIABLES ***
IF (WS-FILE-STREAM FROM WS-INDEX TO (WS-INDEX +5)) =
"NAME"
THEN
MOVE WS-INDEX TO START-INDEX.
*** FIND THE DELIMINATOR AFTER THE NAME STRING AND
***
*** MOVE THE NAME VALUE INTO THE SEX VARIABLE ****
PERFORM
    VARYING WS-INDEX
    FROM START-INDEX
    BY 1
    UNTIL (WS-FILE-STREAM-IN AT INDEX) = LT-
DELIMINATOR
END-PERFORM.
MOVE (WS-FILE-STREAM FROM START-INDEX
TO WS-INDEX) TO WS-SEX.
    PERFORM    1000-PARSE-CUSTOMER-STREAM
THRU 1OOO-PARSE-CUSTOMER-STREAM-END.
ELSE IF (WS-FILE-STREAM FROM WS-INDEX TO (WS-INDEX +
5)) = "SEX "
THEN
*** FIND THE DELIMINATOR AFTER THE SEX STRING AND
MOVE ***
*** THE SEX VALUE INTO THE SEX VARIABLE      ***
MOVE WS-INDEX TO START-INDEX.
```

-continued

```
PERFORM
    VARYING WS-INDEX
    FROM START-INDEX
    BY 1
    UNTIL (WS-FILE-STREAM-IN AT WS-INDEX) = LT-
DELIMINATOR
END-PERFORM
MOVE (WS-FILE-STREAM FROM START-INDEX
TO WS-INDEX)TO WS-SEX
    PERFORM    1000-PARSE-CUSTOMER-STREAM
THRU 1OOO-PARSE-CUSTOMER-STREAM-END.
ELSE IF (WS-FILE-STREAM FROM WS-INDEX TO (WS-INDEX
+5)) = "AGE "
THEN
*** FIND THE DELIMINATOR AFTER THE AGE STRING AND ***
*** MOVE THE AGE VALUE INTO THE AGE VARIABLE ****
MOVE INDEX TO START-INDEX.
PERFORM
    VARYING WS-INDEX
    FROM START-INDEX
    BY 1
    UNTIL (WS-FILE-STREAM-IN AT WS-INDEX) = LT-
DELIMINATOR
END-PERFORM
MOVE (WS-FILE-STREAM FROM START-INDEX
TO WS-INDEX)TO WS-AGE
    PERFORM    1000-PARSE-CUSTOMER-STREAM
THRU 1OOO-PARSE-CUTOMER-STREAM-END.
ELSE
    PERFORM 2000-SAVE-CUSTOMER THRU
2000-SAVE-CUSTOMER-END.
END-IF.
1000-PARSE-CUSTOMER-STREAM-EXIT.
2000-SAVE-CUSTOMER.
*** CALL A SQL MODULE TO SAVE THIS INFORMATION IN
THE
*** RELATIONAL DATABASE
CALL "SAVE-CUSTOMER-IN-DATABASE" USING WS-
CUSTOMER
...
2000-SAVE-CUSTOMER-END.
```

Conversely, a stream could be created by a non-object system or another object system, populated with customer information, and sent to one's object-based system. Once in the object-based system, the CustomerObject could use a "streamOff: aSteam" method, instanciate a CustomerObject, and populate it with the appropriate attribute values.

Collaborations

Stream-based Communication. This is the parent pattern to the Self-Describing Stream pattern. In this pattern, information is transmitted using a simple stream and a shared, generic format. The Self-Describing Stream is a more specific implementation of Stream-Based Communication.

Bridge (from the Gamma book Design Patterns) describes a way to de-couple an abstraction from its implementation so that the two can vary independently. The Bridge pattern is often used to define collaborations between a business object and a format object while decoupling the business object from its specific stream format.

Abstract Factory (from the Gamma book Design Patterns) is a pattern for creating families of related classes. This could be used with the Bridge pattern to retrieve the format dynamically based on non-static information.

Alternatives

Fixed Format Stream—This pattern is a specific variation of Stream-Based communication where the messaging format is defined and stored on both the sending and receiving systems.

Downloadable Format Stream—This pattern is a specific implementation of Stream-Based communication where the messaging format is stored at a central location and is downloaded by the communicating parties when needed.

SAP - Exhibit 1105
SAP v. Valtrus et al. - IPR2025-00420
Page 244 of 281

Stream-based Communication

FIG. **111** illustrates a flowchart for a method **11100** for providing a stream-based communication system. A shared format is defined on interface code in operation **11102** for a sending system and a receiving system. A message to be sent from the sending system to the receiving system is translated based on the shared format in opration **11104**. The message is sent from the sending system and received by the receiving system in operations **11106** and **11108**. The message received by the receiving system is translated based on the shared format in operation **11110**.

As an option, information in the translated message received by the receiving system may be stored in a relational database. As another option, , the shared format may be based on an order of attributes in the message.

In one version, one of the systems may be an object-based system and one of the systems may be a non-object-based system. In another version, both of the systems may be object-based systems. In a third version, both of the systems may be non-object-based systems.

In order to successfully transmit a formatted message, both the sending and receiving systems must understand the format and structure of the transmitted information. Some communications mediums, however, do not inherently transmit the formatting information with the data. How can information be easily communicated between systems when the communication mechanism does not inherently convey data structure or other meta-data information?

For two systems to successfully communicate, they must understand the structure of the data they are passing. The sending system needs to convert standard programming constructs (objects, structure, strings etc.) into bytes of data that can be transmitted along a network. The receiving system needs to receive the bytes of data on a wire and reconstitute it back into objects, structure, strings etc.

Many communication mechanisms inherently provide this functionality to a software developer. CORBA and DCOM are two examples of this type of middleware. Using CORBA, one system can send a structure of information to another system. CORBA will convert the structure into a network appropriate format, transmit it across the network and reformat it on the receiving end.

Other types of middleware, however, do not provide this full range of functionality. Lower level communication protocols like TCP and UDP as well as higher-level protocols like HTTP and telnet do not provide support for sending data structures.

Additionally, popular message queuing software (IBM'S MQSeries and BEA MessageQ) and many custom communications mechanisms are lacking support for transmitting structured data across the network.

These communication protocols do not inherently convey meta-data information. 'Meta'-data is information about the data. Meta-data could describe the data structure (senders address in bytes 1–10, receivers address in bytes 11–20, data in 21–100, etc.), data types, etc.

When the highest-level common communication protocol between two systems cannot convey this meta-data information, an alternative communication mechanism is needed.

FIG. **112** illustrates how systems **11200**, **11202** of the present invention communicate over a communication mechanism **11204** that cannot inherently convey meta-data information.

How can they exchange structured data?

In object-based systems, issues with conveying meta-data are even more prevalent. Object-based system is often need to transfer object information across non-object communication mechanisms (e.g. DCE, . . .) or to non-object systems. Because neither non-object communication mechanisms nor the non-object systems understand the notion of objects, how can the structure of the objects be meaningfully conveyed?

FIG. **113** is an illustration of an object-based system **11300** communicating with a non-object system **11302** using a communication mechanism **11304** that cannot convey meta-data information.

How can they Exchange Structured Data?

Therefore, use Stream-based Communication to transmit information between systems. Stream the data between the two systems and use a generic format to relay the information and its associated meta-data between the systems.

A stream is simply a buffer that data is "written to" and "read from" in discrete quantities. The size of the buffer is predetermined and can be very small (i.e. one byte in length) or very large (i.e. a page). The buffer can't hold objects or structures, but just raw data. Buffers are quite dumb and don't understand anything about their raw data. Thus, it does not have meta data for the information in the buffer.

The "shared format" provides the meta-data information needed to interpret the raw data in the buffer. This shared format is like a secret decoder ring for systems sending and receiving messages. The sending system uses the decoder ring to convert objects, structures, etc. into raw data on a stream. The receiving system uses another decoder ring to reconstitute the raw data back into objects or structures. If objects aren't supported, the raw data is converted into a comparable format for use by the receiving system.

FIG. **114** depicts an example of Stream Based Communication with two disparate systems **11400**, **11402** communicating via stream-based communication **11404**.

In FIG. **114**, the object-based system **11400** uses a shared format (decoder ring) to convert an object into raw data. The raw data is then copied onto the stream. The stream then delivers the data to the Non-Object system **11402**. The Non-Object system **11402** reads the raw data and reconstitutes the data using its shared format.

In this example, the sending system is sending objects while the receiving system doesn't understand objects. Thus, the receiving system can only convert the raw data into a data equivalent of the object sent.

Benefits

Maintainability. When using this pattern, a shared, generic format is used to interpret the data. As a result, the two systems are de-coupled and less dependent upon each other. As long as the format remains unchanged, changes to the internal implementation of either system will not affect the other system. Maintenance of decoupled systems is easier.

Batch Compatible. Strings of data can be concatenated and transmitted as a group. This enables batch messaging (e.g. Request Batcher) and processing.

Enables Lightweight Persistence. Stream-based communication can be used to interface with a lightweight persistence mechanism. Objects, structures, etc. can be can be converted to raw data and streamed to a flat file for saving. At a future time, the file can be opened, the raw data can be streamed out of the file and reconstituted into full blown objects or structures.

The implementation of Stream Based Communication is very straightforward. Simply define interface code on the

SAP - Exhibit 1105
SAP v. Valtrus et al. - IPR2025-00420
Page 245 of 281

sending system that creates a stream and parses the data onto this stream using a format shared by the both the sending and receiving systems. On the receiving system, define interface code that reads the stream and, using the same shared, format, parses the data off of the stream and into a data structure compatible with the receiving system.

The specific implementation of the formats can be, and most likely will be, different from system to system but the actual format must be shared and should be generic between systems. Shared so that the information is accurately relayed and generic to keep the systems as de-coupled as possible by not exposing any implementation details of either system in the format. Further, this shared format can be implemented in a variety of places depending upon the specific requirements of the interface.

For object-based systems, make each object responsible for implementing streaming behaviors that use this shared format. Each object should use the format as a map to parse the attribute values onto a stream (streamOn). Conversely, each object class should use the format as a map to parse its attribute values off of a stream and put them into a newly instantiated instance of the object (streamOff).

In the example below, an object within an object-based system uses stream-based communication to stream its attribute values onto a stream. Then a communication mechanism transports the stream to a non-object system, and a non-object system reads the information off of the stream and inserts it into its relational database.

FIG. **115** is an illustration of a Customer object **11500** in an object-based system **11502** streaming itself into a stream **11504**, the stream **11504** being sent to a non-object system **11506**, this stream **11504** being read and the information is inserted into a relational database **11508**.

1. The CustomerObject with attributes name, sex, and age has a method "streamOn: aStream." It is invoked with an empty stream as the argument 'aStream'. The CustomerObject "streamOn:" method goes through each of the object's attributes and parses each values as a string onto the stream.

The fixed format contract here is embodied in the order that this method parses the attributes onto the stream. A pseudo-code example in Java is the following: Note-Assume that "asString( )" converts the receiver to a string and that "padWithSpaces( )" pads the string with spaces and makes the string the length specified.

```
/** Stream my attribute values on aStream **/
public void streamOn (OutputStream aStream)
{
aStream.write(this.getName( ).asString( ).padWithSpaces(10));
aStream.write(this.getSex( ).asString( ).padWithSpaces(7));
aStream.write(this.getAge( ).asString( ).padWithSpaces(3));
}
```

2. The stream is then put into a message communication mechanism like MQSeries or MessageQ and sent to the non-object system.
3. Once at the non-object system, interface code reads through the stream, parses the values off of the stream, converts them to the appropriate types if required, and puts them in a copybook with the appropriate structure. In this example, the fixed format contract is embodied in the structure and type of the WS-SHARED-FORMAT-CUSTOMER working-storage copybook. Refer to the pseudo-COBOL example below.

```
...
DATA DIVISION.
FD FILE-STREAM-IN
    RECORD CONTAINS 20 CHARACTERS
...
WORKING-STORAGE SECTION.
*** THIS COPYBOOK CONTAINS THE SHARED FORMAT OF
THE
*** CUSTOMER IN THE DATA STRUCTURE AND DATA TYPES
01  WS-SHARED-FORMAT-CUSTOMER
03  WS-SHARED-FORMAT-NAME     PIC X(10).
03  WS-SHARED-FORMAT-SEX      PIC X(7).
03  WS-SHARED-FORMAT-AGE      PIC 999.
*** THIS COPYBOOK IS THIS SYSTEMS VIEW OF A CUSTOMER
O1  WS-CUSTOMER
03  WS-NAME    PIC X(10).
03  WS-AGE     PIC 999.
03  WS-SEX     PIC X(10).
...
PROCEDURE DIVISION.
...
*** OPEN THE FILE STREAM AND PUT THE CONTENTS IN THE
*** WS-SHARED-FORMAT-CUSTOMER COPYBOOK.
OPEN FILE-STREAM-IN
READ FILE-STREAM-IN INTO WS-SHARED-FORMAT-
CUSTOMER
    AT-END CLOSE FILE-STREAM-IN
END-READ.
*** MOVE THE VALUES INTO FROM THE SHARED FORMAT
INTO
*** THE WS-CUSTOMER VARIABLES.
MOVE WS-SHARED-FORMAT-SEX TO WS-SEX.
MOVE WS-SHARED-FORMAT-AGE TO WS-AGE.
MOVE WS-SHARED-FORMAT-NAME    TO WS-NAME.
...
*** CALL A SQL MODULE TO SAVE THIS INFORMATION IN
THE
*** RELATIONAL DATABASE
CALL "SAVE-CUSTOMER-IN-DATABASE" USING WS-
CUSTOMER.
...
STOP-RUN.
```

Conversely, a stream could be created by a non-object system (or another obejec-base system for that matter) and sent to one's object-based system. In this case, CustomerObject could use a "streamOff: aStream" method and instanciate a new instance of a aCustomer-Object and populate it with the appropriate attribute values.

Again, there are several variations of this pattern depending upon what the specific requirements are. Some of these variations are further explained in the children patterns. Refer to Fixed Format Stream, Downloadable Format Stream, and Self-describing Form at Stream.

Collaborations

Fixed Format Stream—This child pattern is a specific variation of Stream-Based communication where the messaging format is defined and stored on both the sending and receiving systems.

Downloadable Format Stream—This child pattern is a specific variation of Stream-Based communication where the messaging format is stored at a central location and is downloaded by the communicating parties when needed.

Self-Describing Stream. This child pattern is a specific variation of Stream-Based communication where the messaging format is parameterised and stored on the stream. A message language is used to read and write the format of the message from the stream.

Structure Based Communication—This pattern uses a Fixed Format Stream to transmit data structure between systems. It is often used to obtain data from a Server for display in a Client UI.

SAP - Exhibit 1105
SAP v. Valtrus et al. - IPR2025-00420
Page 246 of 281

Bridge (from the Gamma book Design Patterns) describes a way to de-couple an abstraction from its implementation so that the two can vary independently. The Bridge pattern is often used to define collaborations between a business object and a format object while decoupling the business object from its specific stream format.

Abstract Factory (from the Gamma book Design Patterns) is a pattern for creating families of related classes. This could be used with the Bridge pattern to retrieve the format dynamically based on non-static information.

Structure-Based Communication

FIG. **116** illustrates a flowchart for a method **11600** for efficiently retrieving data. A total amount of data required for an application executed by a client is determined in operation **11602**. In a single call, the total amount of data from a server is requested over a network in operation **11604**. All of the data is bundled in operation **11606** into a data structure by the server in response to the single call. In operations **11608** and **11610**, the bundled data structure is sent to the client over the network and the data of the data structure is cached on the client. The cached data of the data structure is used as needed during execution of the application on the client in operation **11612**.

The data structure may be bundled on the server by a business object. In addition, the business object may be instantiated by an action of the client. Also, the network may be at least one of a local area network and a wide area network. As a further option, the request may be administered by a proxy component. Further, the data structure may contain no logic.

In a client server application, the client communicates with a server over a network. Depending upon the speed of the network and the number calls across the network, an application can experience performance problems. How can a client update a server while minimizing the network traffic and maintaining an acceptable level of system performance?

Acceptable system performance is an important attribute of every application.

When creating a client-server application, the performance of the network must be considered during the design and development of an application. The speed at which data can be transmitted across the Local Area Network or Wide Area Network, can make or break a client-server application.

In a typical three-tiered client-server application, the business objects are maintained away from the users (Client) on separate Server machines. Whenever a user needs the expertise of a business object, the user must send a request across the network to the Server machine.

FIG. **117** illustrates the manner in which a client **11700** requests information from server objects **11702** via a network **11704**.

Depending upon the size of the message and the speed of the network, this could take a long time. This is a reality of three-tiered client-server applications.

When a client-server application is introduced to the world of distributed objects, the network can become an even larger bottleneck. In a pure distributed object approach, the client is passed an object reference to a business object on a server machine. The client then accesses the specific business object over the network as if it resided on its local machine. Using this "pure" distributed object approach, the application's calling pattern begins to look like the schematic of FIG. **118**.

FIG. **118** illustrates the method of the present invention in which a client **11800** requests attributes from a server object **11802** via a network **11804**.

This is an excellent programming model that frees the developer to access local and remote objects in the same fashion. Unfortunately, it makes it easier for the application developer to forget about the physical realities of the network. A network call is always slower than a call within a single machine. Ignoring this reality may result in an unacceptably slow application.

On a very fast LAN where the number of network calls is small, this calling pattern may be acceptable. On a slower LAN, any WAN or when the number of network calls is large, this pattern will yield unacceptable network performance. Something must be done to maintain an acceptable level of system response for the users.

FIG. **119** illustrates the transmitting of all data in a Data Structure **11900** from a client **11902** to a server **11904** and visa-versa. As shown, to maximize the performance on the client, it is best to bundle all the necessary data into a single data structure that can be transmitted as a structure across the network.

The Client would first determine the sum total of everything it will need from the business object on the Server machine. The Client makes a request for all of this data from the business object. The business object bundles all the data into a data structure and returns it to the client. The Client will cache this data (using the Caching Proxy pattern) on its local client machine and use it as needed.

Benefits

Better System Performance—This pattern will improve the performance of an application by reducing the network traffic between the client and the server. The client makes one network request to retrieve all of the data from the server. Regardless of the number of displayable attributes the application will only make one network send.

Without this pattern, the client could make a network send for every attribute retrieval. If the user wants to retrieve a customer's name, address and phone number, that could result in three network requests (one for each attribute). Without this pattern, the network traffic can become prohibitive and the performance would suffer.

Structure Based Communication assumes a client needs information from an object that exists on a server. Thus, this pattern assumes the existence of an "interesting" object on the server machine.

Even though the "finding" and "instantiating" of a server object isn't part of this pattern, it does establish context and sets the stage for the pattern. As a result, a message trace diagram for finding and instantiating a particular object instance is shown below. This will set the stage for the implementation of the Structure Based Communication pattern.

FIG. **120** illustrates the method in which a client **12000** finds and instantiates a Customer Object from a customer component **12002**. The various steps shown in FIG. **120** will now be set forth.

Collaborations

1. The client instantiates a Proxy (Customer Component Proxy) to the Customer Component. The Client then asks the Proxy for Customer Jimbo Jones.
2. The Customer Component Proxy forwards the request across the network to the Customer Component.

3. The Customer Component requests the information for Jimbo Jones from the database.
4. The Database returns the data associated with customer Jimbo Jones.
5. The Customer Server Component instantiates a customer object using the Jimbo Jones data from the database.
6. The Customer Server Component returns a remote object reference to the "Jimbo Jones" object running on the Server.
7. The Client creates a proxy to the "Jimbo Jones" object using the remote object reference.

Now that a customer object (Jimbo Jones) exists on the server component, Structure Based Communication can be used to pass the needed data from the server to the client.

FIG. **121** ilustrates a Structure Based Communication that builds upon the method of FIG. **120** and depicts the flow of control during Structure Based Communication. The various steps shown in FIG. **121** will now be set forth.

Collaborations

8. The Client asks the Customer Proxy for the data associated with the Jimbo Jones object.
9. The Customer Proxy forwards the request across the network to the Customer Component
10. The Jimbo Jones object creates a data structure and populates it with its data.
11. The Data Structure is passed across the network to the Customer Proxy on the Client.
12. The Customer Proxy forwards the data structure containing Jimbo Jones' data to the Client component.

Participants

Client—The "client" for the transaction. This could be a User Interface that displays customer data for viewing by a Customer Service Representative.

Network—A LAN or WAN network that connects the Client with the Customer Component.

Customer Component—A server component that encapsulates the data for all of the customers in a system.

Customer Component Proxy—A proxy to the Customer Component. Any request it receives, it forwards across the network to the Customer Component.

Customer Proxy—A proxy to the Jimbo Jones Customer Object Any request it receives, it forwards across the network to the Jimbo Jones Customer Object.

ACustomerStructure—A data structure. It contains the data (but no methods) from the Jimbo Jones object.

Database—Any relational database.

Jimbo Jones Object—An object that represents the Jimbo Jones customer. This object contains Jimbo Jones' data and methods associated customer methods.

Sample Java Code

The following java example accompanies the previous message trace diagrams. The java code follows the same scenario as the message trace diagrams, but it has been simplified in some areas.

The following snippet of code defines the data structure used to pass customer information.

```
public CustomerStructure(
        String    firstName,
        String    lastName,
        short     streetNumber,
        String    street,
        String    city,
        String    state,
        String    zipCode)
```

The following block of code is the "main" routine for the Client. Even though all distributed calls are made through a proxy, the proxy code is not shown in this example. The two proxy classes encapsulate the details associated with distributed network calls.

```
// Client side code here.
Main( )
{
// Create a Proxy to the Customer Component.
CustomerComponentProxy aCustomerComponentProxy =
    new CustomerComponentProxy( );
// Get a Proxy to the Jimbo Jones Customer
// object.
CustomerProxy aCustomerProxy =
    aCustomerComponentProxy.getCustomer("Jimbo Jones");
// Get Customer data from the Customer Server
// Component(Call across the network)
CustomerStructure aCustomerStructure =
    aCustomerProxy.getCustomerAsStructure( );
// Use the Customer data received in the
// structure. For Example, display the data
// structure data (aCustomerStructure) in a UI.
        .
        .
        .
}
```

The following code is a sample Customer Server Component. The Customer Server Component is used to retrieve the data associated with customer Jimbo Jones from the database. It also instantiates a customer object using the data retrieved from the database.

```
// Customer Component Code here
public class CustomerComponent
{
    // Put the data associated with a Customer Object
    // into a data Structure. This data structure
    // will be sent across the network to a client.
    public Customer getCustomer(String aCustomerName)
    {
        // Find the Customer in the database
        .
        .
        .
        // Instantiate the Customer Object
        Customer aCustomer = new Customer(..
        .
        .
        // Return a "remote object reference" to the
        // Jimbo Jones Customer object.
        return (aCustomer);
    }
}
```

Finally, this is the code for the Jimbo Jones Customer object.

```
// Customer object code here
public class Customer
{
    // Put the data associated with a Customer Object
    // into a data Structure. This data structure
    // will be sent across the network to a client.
    public CustomerStructure getCustomerAsStructure()
    {
        CustomerStructure aCustomerStructure = new
CustomerStructure();
        aCustomerStructure.firstName = this.getFirstName();
        aCustomerStructure.lastName = this.getLastName();
        aCustomerStructure.streetNumber = this.getStreetNumber();
        aCustomerStructure.street = this.getStreet();
        aCustomerStructure.city = this.getCity();
        aCustomerStructure.state = this.getState();
        aCustomerStructure.zipCode = this.getZipCode();
        return aCustomerStructure;
    }
    // Getters and Setters for all attributes
    // (code not shown here)
        .
        .
        .
}
```

Note: The "Main" code example above obtains a Proxy to customer "Jimbo Jones" and then a Structure of data through the proxy. The code was written for ease of understanding, but causes two calls across the network. In reality, it is preferred to perform both functions using one network call. Limiting the number of network calls will improve system performance. The recommended code might look something like this:

```
// Create a Proxy to the Customer Component
    // (same as above)
    CustomerComponentProxy aCustomerComponentProxy = new
    CustomerComponentProxy();
    // Get a Proxy to the Jimbo Jones Customer
    // object
    //              AND
    // customer data at the same time.
    CustomerProxy aCustomerProxy;
    CustomerStructure aCustomerStructure =
        aCustomerComponentProxy.getCustomerData("Jimbo Jones",
    aCustomerProxy);
```

Collaborations

Proxy—This pattern is documented in the book "Design Patterns" by Gamma, Helm, Johnson and Vlissides. The proxy pattern is often used to communicate with server components in a distributed environment. The Proxy pattern is often used to retrieve data structures from a server component.

Cached Proxy—This pattern is documented in "The Proxy Design Pattern Revisited" section of the Pattern Languages of Programming Design **2** book. A Cached Proxy caches data locally on the client. Structure Based Communication uses and builds upon this pattern.

Globally Addressable Interface—This pattern often works in conjunction with Structure Based Communication. Oftentimes, a Globally Addressable Interface is used to obtain Structures of data for display on a Client.

Locally Addressable Interface—This pattern can also be used in conjunction with Structure Based Communication. After establishing a relationship with an LAI, a client may obtain data from the Server object using Structure Based Communication.

Alternatives

Distributed Objects—The "pure" distributed object approach is an alternative to Structure Based Communication. Using this pattern, individual objects are queried for each piece of information needed by a client.

PRESENTATION SERVICES (**1000**)

Presentation Services enable an application to manage the human-computer interface. This includes capturing user actions and generating resulting events, presenting data to the user, and assisting in the management of the dialog flow of processing. The Presentation Services forward on the user requests to business logic on some server. Typically, Presentation Services are only required by client workstations.

In addition to Presentation Services on the client, some business logic will usually reside on the client as well to aid the Presentation Services. Even on thin clients some sort of validation logic is usually included with the Presentation Services. A quick review of the Gartner Group's five styles of client/server computing help to illustrate this.

FIG. **122** shows the Gartner Group's Five Styles of Client/Server Computing **12200**.

The way that Presentation Services interact with client-side business logic is very important to the overall scalability and maintainability of the application. An application's business logic is expected to be highly reusable, even on the client. If business logic is coupled with the Presentation Services too tightly, it will be very difficult to separate and reuse the business logic if the Presentation Services ever need to be altered (not an uncommon occurrence).

The patterns in this section help to guide application architects on strong, proven techniques to safely integrate client-side business logic with an application's Presentation Services. The Activity pattern lays the groundwork for separating the Presentation Services and business logic on the client by assigning non-presentation logic to a type of object called an Activity. The View Configurer pattern helps to assign new views with their appropriate Activity. Finally, the User Interface Validator pattern describes how to implement customizable, extendable validation logic on a user interface.

Activity

FIG. **123** illustrates a flowchart for a method **12300** for providing an activity module. A server and a presentation interface of a client are interfaced to permit the receipt of requests for service from the presentation interface of the client in operations **12302** and **12304**. A portion of the requests are handled on the client in operation **12306**. In operations **12308** and **12310**, another portion of the requests are forwarded to the server for further handling purposes and changes are effected in the presentation interface.

A plurality of presentation interfaces may be interfaced. Optionally, a model may be interfaced for management purposes. With such an option, the model may further include a proxy. As another option, errors and exceptions may also be handled. As a third option, events intended to be received may be triggered by the presentation interface.

Many client/server applications maintain some amount of business logic on the client. How can an application represent and reuse "client-side" business logic across multiple, volatile user interfaces?

Imagine a typical client/server system design. In almost all cases, a typical system executes data access logic on the server and presentation logic on the client, business logic is

SAP - Exhibit 1105
SAP v. Valtrus et al. - IPR2025-00420
Page 249 of 281

split across both the client and server. The majority of this business logic is maintained on the,server. This logic is represented by various components and business objects that can communicate with each other to complete a variety of system use cases.

The client, on the other hand, is mostly responsible for supporting user interactions with the system. To be successful, the client must also execute some degree of business logic. While this can vary from implementation to implementation, some categories of logic are invariably located on the client. This includes simple data validations, representing data structures and relationships, error and exception handling, and communications with the server.

To complete a single use case, the client may need to interact with a number of server components. From the user's perspective, one unit of work is being performed but it may involve multiple, discrete interfaces and multiple server invocations. Some business logic is required to manage the complex flow to complete this unit of work. For example, suppose a use case for a network inventory management system is "Add Network Card". This may require the user to input information in three or four screens and client communication with more than one server component. Managing this flow is not the responsibility of the presentation logic but still needs to be executed on the client. The system may also require a number of interfaces to complete the same use case. Depending on the user category executing the use case, the interface may be a PC, a handheld device, or a telephone. Even on the same type of device, the interface may differ depending on the user category. Some users may want to access the application via a standard Windows interface while others may want to access it via a "web-centric" interface (internet browser). In all of these cases, the unit of work to be completed by the user is not changed and should be reused.

FIG. **124** illustrates multiple interfaces to an application **12400** including a handheld device **12402**, a desktop PC **12404**, and a telecommunications device **12406**.

Often these user interfaces will be changed over time to fit user's changing needs. While the tasks completed by the user may not change, the interface to complete those tasks will need to. Windows users will want to move to the Web. Web users will want to move to handheld devices. The presentation code should be able to be changed without causing a rewrite of the business logic on the client.

Therefore, bundle business logic executed on the client separate from the presentation logic. This new type of class is an Activity.

An Activity is responsible for:

managing client logical units of work

maintaining client representation of a business model

validation across multiple interfaces (complex business logic)

error and exception handling

communication with server and other services

creating other Activities

triggering events intended to be "caught" and acted on by the presentation logic

An Activity resides between the actual user interface and the business model and server components as shown in the Entity Relationship diagram below:

FIG. **125** illustrates an activity entity relationship diagram.

While any user interface maintains a reference to the Activity **12500** it provides an interface **12502** for, the Activity is unaware of what (if any) interfaces exist on it. This decoupling allows for a large amount of flexibility with the interfaces to an application. Multiple types of interfaces can exist on a single type of Activity. Code is reused and none is lost if presentation logic is replaced with something different. While a user interface can communicate directly with its associated activity, an activity should never directly communicate with any of its interfaces. This would set up a dependent relationship that would reduce the flexibility of the activity.

Instead, an activity can communicate to its interfaces through an event mechanism. Interfaces are set up as dependents of the activity and the activity sends events to all of the interfaces on it. Each interface can decide how to handle the event.

FIG. **126** illustrates a roles and responsibilities diagram.

Benefits

Maintainability. By separating the presentation and non-presentation logic, the client is easier to understand and maintain.

Reuse. The presentation layer may be replaced or reused without affecting the non-presentation logic.

FIG. **127** illustrates a typical implementation between a user interface and its activity. The diagram shows the various "layers" of interaction where the lightest shaded boxes are the presentation, the next darkest is the activity, and the darkest is the component (proxies **12700**).

A user request is captured and processed by the presentation object (NetworkInventoryUserInterface **12702**). In this case, the processing involves simple validation (format and "is null" checking).

The presentation object then copies its data into a structure representing some business entity (aNetworkItem **12704**) and passes it to the activity **12706**. The presentation object then triggers the activity to start its processing of the new network item.

The activity then performs possibly more complex validation and communicates with the server components to complete the use case.

Collaborations

Facade—The Activity acts as a Facade to all of the server components by coordinating a user interfaces interaction with them.

Separation of Concern—Dividing defined responsibilities into separate classes (presentation logic into UI classes and client-side business logic into activity classes).

Observer—An activity's interfaces are observers of that activity.

User Interface Validator

FIG. **128** illustrates a flowchart for a method **12800** for structuring validation rules to be applied to a user interface for maximum maintainability and extensibility. In operations **12802** and **12804**, a plurality of user interface widgets are provided along with a plurality of validation rules which govern use of the user interface widgets. A user is allowed in operation **12806** to select the validation rules to associate with the user interface widgets of a first user interface. The validation rules of the user interface widgets of the first user interface are automatically associated across a plurality of separate different user interfaces in operation **12808**.

The validation rules may be created at the time the first user interface is created. As another option, the validation rules may be implemented by a different class for each type of validation. As a further option, an indicator may be displayed upon one of the validation rules being violated.

SAP - Exhibit 1105
SAP v. Valtrus et al. - IPR2025-00420
Page 250 of 281

Additionally, each validation rule class may extend an abstract validation rule class that defines what types of widgets are supported. Also, a request for the validation rules may optionally be received from one of the user interfaces.

How can you structure validation rules to be applied to a user interface for maximum maintainability and extensibility.

Imagine a typical Windows or web-based client/server application. In most cases where a "windows" type of user interface is provided, an application supports some business rules by validating data entered by the user. A common example of this is checking the format of data in an entry field or ensuring that a required field is not left empty.

The business rules supported by user interface validation is usually somewhat limited. The scope of these rules is generally constrained to checking if a field is empty, checking the format of a field (date, time, numeric, currency, etc.), and checking if a field has alpha-characters, numeric-characters, or both. In addition, due to fact that many widgets provide constraints through their own form (list boxes, radio buttons), the types of widgets that require this type of validation checking is also somewhat limited (text fields, editable combo boxes, etc.).

FIG. **129** illustrates widgets with their validation requirements **12900**.

Because this type of validation will most likely be required across all of an application's user interfaces and the fact that the types of validation rules and widgets needed to validate are limited, this behavior is a strong candidate for a framework.

The framework would provide a common approach to validating user data across all of an application's user interfaces. Rules would be applied consistently throughout the application. While some common validation rules would be provided, the framework needs to allow their behavior to be modified (overridden) and make it easy for new rules to be added.

Finally, both immediate and batch validation should be provided by the framework.

Therefore, for each user interface in an application, encapsulate their validation logic in a User Interface Validator. FIG. **130** illustrates a user interface validator association diagram. A User Interface Validator **13000** associates various validation rules with the user interface widget they are to be applied to.

The associations are created at the time the user interface is created. The validations are triggered when deemed necessary by the user interface. Any validations that fail are displayed to the user including the type of validation that failed and the widget that it failed on.

The rules are implemented by a different class for each type of validation. Each of these validation rule classes must know how to check its rule for every type of widget that can be checked. As mentioned in the Context section of this pattern, this will most likely be limited to text entry type widgets. In addition, each validation rule class extends an abstract validation rule class that defines what types of widgets are supported. This is an implementation of the Visitor pattern.

FIG. **131** illustrates a validation rule class diagram.

Note that the check operations accept a Validate **13200** type of class. Each widget that can be validated with this framework must implement a validateRule method. This simple method accepts some ValidationRule **13202** as a parameter and simply turns around and calls the check method on the rule passing itself in as a parameter. This interaction is shown in FIG. **132**, which illustrates a rule validation interaction diagram.

The concrete implementation of the check method will be invoked. This method knows how to extract the data from the particular widget provided and verify the rule.

The User Interface Validator's job is to associate these rule instances with all of the widgets they pertain to. When the validate method is invoked on the Validator, all of the rules are sent to each of the appropriate widgets via the validateRule method.

New rules can be added by creating new classes that extend off of the abstract Validation Rule class. No changes need to be made to the widgets.

Benefits

Consistency. All user interface validation rule checking is done in the same way using the same rule logic.

Extensibility. New rules can be added without affecting any other part of the application.

Automation. Application of validation rules can be automated with a GUI based tool rather easily.

The associations between a widget and the rule to apply to it should be set up when the user interface is created. A user interface can implement a method that accepts a rule and widget and passes it on to the User Interface Validator as shown in the code example below:

```
ValidateTextField userNameField = new TextField("user name");
ValidateTextField passwordField = new TextField("password");
ValidateTextArea commentsArea = new TextArea("comments);
this.addValidation(userNameField, new NotEmptyValidationRule());
this.addValidation(passwordField, new NotEmptyValidationRule());
this.addValidation(passwordField, new NotNumericValidation
Rule());
this.addValidation(commentsArea, new MaxLengthValidationRule
(255));
```

The addValidation method on the user interface is shown below.

```
public void addValidation(ValidateWidget aWidget, ValidationRule
aRule)
{
    UIValidator myValidator = this.getValidator();
    MyValidator.addAvlidation(aWidget, aRule);
}
```

The add Validation method on the User Interface Validator is shown below.

```
public void addValidation(ValidateWidget aWidget, ValidationRule
aRule)
{
    Hashtable rulesAndWidgets = this.getRulesAndWidgets();
    if (rulesAndWidgets.containsKey(aRule))
    {
        aRule = aRule.clone();
    }
    rulesAndWidgets.put(aRule, aWidget);
}
```

In the above code, three widgets are created and then associated with various validation rules. The user name and password fields are required and cannot be left blank, the password may not contain any numbers, and the comments text area may not be longer than 255 characters.

Note that each of the widgets is created with a string that describes a name for the widget that the user would recognize. This name is used in the error list to help a user identify which widget failed validation.

SAP - Exhibit 1105
SAP v. Valtrus et al. - IPR2025-00420
Page 251 of 281

At some appropriate time, the user interface sends the validate message to the User Interface Validator. This method steps through each of the rules provided to it when the user interface initialized and passes them to their associated widget by the validateRule method. The code is shown below:

```
public Vector validate()
{
    Vector errors = new Vector();
    Hashtable rulesAndWidgets = this.getRulesAndWidgets();
    Enumeration rules = rulesAndWidgets.keys();
    while (rules.hasMoreElements())
    {
        ValidationRule aRule = (ValidationRule)rules.nextElement();
        ValidateWidget aWidget = (ValidateWidget)
            RulesAndWidgets.get(aRule);
        String anError = aWidget.validateRule(aRule);
        if(anError != null)
        {
            errors.addElement(anError);
        }
    }
    return errors;
}
```

Note that in the code example above, the validateRule message returns a String rather than a boolean. This string can be passed back to the user interface that invoked the validate and used to describe the errors that occurred to the user.

Collaborations

Visitor Each of the rules are implemented as a Visitor according to the GoF pattern of the same name.

View Configurer

FIG. **133** illustrates a flowchart for a method **13300** for assigning a view to an activity. Notification is received that a startup event of an activity has occurred in operation **13302**. A reference to a first instance of an object created by the startup event of the activity is also received in operation **13304**. In operation **13306**, a view to launch is determined in response to the receipt of the notification and the reference. The view is based on predetermined criteria. The view is associated with the activity and displayed in operations **13308** and **13310**.

The predetermined criteria may include user preferences, an experience level of a user, security profiles, and/or workflow settings. Also, the activity may be allowed to run without a corresponding view. The activity may also operate on a machine separate from a machine of an end user.

As an option, a request may be sent to be notified when a new instance of an object is created. As another option, a configuration file may be read for obtaining configuration information.

How do I associate a new view with the appropriate business activity underneath, in a configurable manner?

Consider a user interface that displays and collects data for an activity object underneath.

The ICM/MVC patterns provide for a layered architecture. Each layer talks to a layer below it, and no lower layer talks to an upper layer. For example, the view messages downward to the activity and the business objects, and the activity messages to the business objects. Layers talk down. No layer messages back upward.

Traditionally, activities launch their views directly. In the example illustrated below, the Search View tells the Search Activity to launch the Customer Maintenance Activity, which then opens up its own view. But this violates the ICM approach, because then the model is talking directly up to the view.

FIG. **134** illustrates a manner in which the maintain customer activity operation **13400** of the present invention launches its view **13402**.

It might be more appropriate to let the view layer, rather than the activity layer, make decisions about launching other views. The view layer already knows about usability preferences, positioning on the screen, etc. However, one wants the activities to control conversational flow for preconditions, postconditions, workflow, and any other additional business logic.

A view should not be able to launch a new, separate activity, because that involves business logic. Instead, one wants activities to launch other activities. When the activity layer controls conversational flow, one needs a mechanism to launch views on top of these activities, without violating ICM.

Therefore, a View Configurer **13500** will be created to manage the relationship between activities **13502**, **13504** and views. This would likely be a singleton.

FIG. **135** illustrates the view configurer **13500** launching the maintain customer view operation.

The View Configurer is a generic mechanism which allows launching of different views, based on certain criteria. It uses an observable relationship with activity factories to solve this problem. With the View Configurer, developers do not hard code the particular policies for the selection of a view. Moreover, this mechanism allows activities to run without a corresponding view.

Communication from the activity to the View Configurer will be conducted through broadcasting (as described in the Observer pattern). In this manner, the activity doesn't know about the existence of the View Configurer, it listens to activity broadcasts such as when the controller starts up. This configurer can use the Observable Factory to get a handle to the activity instance.

There are four main steps involved with the View Configurer observer/observable interface:

Prior to the flow depicted above, the View Configurer has registered with the Activity Factory saying "Tell me when a new instance is created". This is an example of an "Observable Factory," which can be thought of as a Factory which implements the subject/observable role of the Observer pattern. The Factory needs to be a singleton, so the View Configurer has visibility to it.

The Search View tells the Search Activity to launch the Maintain Customer Activity. The factory for the Maintain Customer Activity creates a new instance of the Maintain Customer Activity.

Because the View Configurer has pre-registered an interest in the startup of activities, it will receive a broadcast message. In this step, the View Configurer should receive a minimum of two parameters:

Notification of the startup event that has just occurred.

A reference to the new instance of the object that was just created.

The View Configurer then determines which view to launch. This can be based on a variety of criteria, such as user preferences, experience level, security profiles, or workflow settings. The configurer determines the correct view and attaches it to the activity underneath.

Benefits

Development. Depending on the distribution model in place, business processing can be executed and tested before the appropriate views have been implemented.

Automated testing. The View Configurer is particularly useful when you want to use scripts and avoid bringing up windows with automated testing. This is especially true for performance testing, where you might want to run **100** transactions, which might involve instantiating **100** instances of the same activity.

Running processes in batch mode. The View Configurer allows processes to run without a View, and makes it very simple to connect, disconnect, or reconnect related views.

Distribution Transparency. In a distributed environment, the process might live on a different machine from the end user's machine. In that case, it cannot launch the view directly, within its own executable. (Unless using a remote windowing system like X-windows, etc.) So the View Configurer allows application architects to transparently move process logic around, depending on the distribution model.

Collaborations

The Observer Pattern (Gang of Four Pattern) describes how to provide visibility to other entities via a one to many relationship. A singleton activity factory will create new activity instances, and broadcast the startup of the new activities to the View Configurer.

An interface for the creation of activities is used in conjunction with the Observer Pattern. In this way, the startup of new activity instances can be broadcast to the View Configurer. This is described in the Factory Pattern (Gang of Four Pattern).

Generally, there will only need to be one View Configurer per executable. The View Configurer would likely be a singleton, as described in the Singleton Pattern (Gang of Four Pattern).

ENVIRONMENT SERVICES (**1016**)

Environment Services provide miscellaneous application and system level services that do not deal directly with managing the user-interface, communicating with other programs, or accessing data. These services are divided into:

Operating System Services

Runtime Services

Version Management

Licensing Services

Error Handling/Logging Services

Properties

Task and Memory Management

Security

"Miscellaneous services" should-not be interpreted as "less important services." In fact, they are vitally important. Developers are more productive when they are not required to be concerned over logging and auditing, error handling and context issues. Obtaining the freedom to largely ignore these issues requires close attention to providing facilities which are well thought out and meld into the application structure.

Despite the pervasive demands of environmental considerations, many forms of documentation largely gloss over these issues. For example, many times when reading API documentation we find authors disclaim the fact that no error handling or "programming by contract" is shown within the code examples to improve readability. Yet, getting error handling right is key to stability in the execution environment. Programming by contract with the use of preconditions and post-conditions is perhaps the most aggressive style of programming known to date to assure correct programs. Assertion, Exception Hierarchies, Exception Response Table and Polymorphic Exception Handler tackle these problems vigorously by helping to define clearly how to solve some of these key kernel application architecture considerations. The Exception patterns provide a blueprint illustrating how architectural issues can be abstracted into a service level component so that the impact to the application code is minimal.

A demanding issue in distributed systems is gathering and using trusted information about the clients interacting with the systems. In earlier generations of systems the number of users was a fairly precise calculation—just count the number of workstations which could potentially connect to an application. Information about the users was also a fairly simple matter since they connected directly to the resources from which they were requesting services. Today, with clients offering web services within n-tiered architectures this is no longer easily predictable. In addition, requirements to support these less predictable numbers of users and to have a personal "one-to-one" relationship with them is key to many web strategies. The LoadBalancer and UserContext pattern offer some help in this area. The former addresses strategies for ensuring maximal leverage of the system resources and services and the latter helps in addressing the issue of maintaining state and context about the user. These facilities are mandatory when security, auditing and logging are considered essential properties of the environment.

Assertion

FIG. **136** illustrates a flowchart for a method **13600** for testing successfulness of an operation having pre-conditions and post-conditions that must be satisfied for the operation to be successful. A first assertion is raised in operation **13602** asserting a pre-condition that must evaluate to true if the operation is successful. The operation is then executed in operation **13604**. A second assertion is raised in operation **13606** asserting a post-condition that must evaluate to true if the operation is successful. An error message is outputted upon failure of at least one of the assertions in operation **13608**.

Optionally, an error handler may be provided for detecting a failure of the assertion of one of the conditions and shutting down a system running the operation upon the detection of the failure. As another option, an assertion may be raised at the beginning and end of every operation of a plurality of operations. Also, a check may be performed prior to raising the assertions for determining the propriety of raising the assertions.

Each assertion may be raised with descriptions for helping to identify where the assertion failed. Also, each assertion may be raised with parameters for helping to identify why the assertion failed. In one embodiment, two types of assertion classes may be provided. In such an embodiment, one of the assertion classes may implement assertion-checking logic and the other assertion class may implement only null operations, with one of the assertion classes being selected to be raised.

Every operation has a set of preconditions and postconditions that must be met for the operation to be considered successful. If these expectations are not met, the system state is in error. How can operations check for these errors so that the handling of these critical errors are consistent across the application?

Methods typically obtain and return a value, set an attribute based on a passed in parameter, or modify the state of the application based on some complex business rule or ruleset. While there is always some expected result of the invocation of an operation, there are also other, less expected possibilities. The provided parameters may not be within the expected range, thereby causing an error. A communications failure could cause the operation to fail to complete or, worse yet, return incorrect data or leave the system in an inconsistent state.

Any complete design determines that some formal specification is required to ensure operations complete correctly. This specification is most often in the form or pre-and post-conditions. These conditions define a set of logical expressions that must hold true for the operation to begin and end as expected. These conditions are usually defined during the Design Phase of development. An example is shown in the Operation Diagram below:

FIG. **137** illustrates an operation diagram depicting an example of pre-conditions **13700** and post-conditions **13702**.

The conditions, in short, define the contract for the method. All of these pre-conditions must hold true before an operation's execution and all of the post-conditions must hold true after an operation's execution. Only then is the operation said to be successful. If any of these conditions fail, a critical error has occurred. The system must assume it is in an inconsistent state and cannot continue processing.

It is expected that the system programmers will check for pre- and post-conditions systematically in the operations they are coding. This seemingly simple requirement becomes non-trivial when some issues are considered:

How can multiple developers implement these checks in a consistent manner?

Some condition checks may be expensive to complete (database and remote component queries). How can these be turned on and off to meet performance expectations? Problem with deferred evaluation; see below.

How can the exceptions raised when a condition check fails be handled in a consistent manner throughout the system?

Therefore, a type of object should be developed to represent a check against an operation's conditions. This generic class of objects is known as an Assertion. Application developers should then raise Assertions throughout the system to check the correctness of the code and system state.

An Assertion accepts conditions that must always evaluate to true. If any of these conditions ever fail, a critical error has occurred and the system should shut down. Preconditions and post-conditions are good examples of the type of conditions that should be asserted during an operation's execution.

The Assertion class is passed a condition that, if evaluated to be false, raises the appropriate errors and shuts the system down. The purpose of this pattern is to formally recognize the pre- and post-conditions of a method in the actual code rather than through developer comments. By implementing an asserts method on a common superclass, the interaction with the Assertion class can be hidden from the functional developer. An example of the use of assertions is shown below:

```
public Customer createCustomer(int newCustomerNumber)
{
    Customer newCustomer = null;    // declare the new customer
    this.assert(newIdentifier>0);//pre-condition, a customer's
                    // identifier must be greater than
                    // zero
    // code to create the customer
    this.assert(newCustomer != null);    // post-condition, the customer
was
                    // created
    return newCustomer;
}
```

Assertions can be raised with descriptions and parameters. A description can help to identify where the Assertion failed and a parameter list can help to identify why the Assertion failed.

Assertions should be raised at the beginning and end of every operation. Prior to raising the Assertion, a check should be made to see if it appropriate to raise one (if assertions are enabled, if performance sensitive assertions are enabled). This can be accomplished by querying the Assertion class for its state before checking the assertion:

```
if(!Assertion.isPerformanceSensitive())
{
    // assert!
}
```

All operations will have both pre- and post-conditions. Even in cases where an operation defines an input parameter as something as broad as an integer, it is doubtful that all integers are acceptable to the operation. In this case, an Assertion should be raised to check if the parameter is in the appropriate range.

A "top-level" error handler should be defined to catch all AssertionExceptions and handle them in a clean and consistent manner. This should include reporting the assertion failure and shutting down the system following an orderly procedure.

It is important to note the difference between assertions and standard error-handling. Assertions are condition checks that can be turned on and off during runtime whereas standard error handling is always enabled. This is because assertions must always be true. The burden is on the testing process to catch all failed assertions. Thus, a failed assertion should simply never happen in deployed code. However, exceptions can happen, and therefore cannot simply be turned off.

Benefits

Ease of Error Identification. Many error are caused by invoking an operation with improper data (parameters). By formalizing these conditions, it is very obvious is an error was caused by bad data or bad code.

Correctness. Properly placed assertions assure that the system is in a correct state and responses can be trusted. Assertion checking complements, but does not replace, a comprehensive testing program. The responsibility remains with the designer to identify the correct conditions to assert.

Consistency. All checks will be made and handled in a similar fashion.

Control. The enabling and disabling features of the Assertion allows an operations controller to determine when

SAP - Exhibit 1105
SAP v. Valtrus et al. - IPR2025-00420
Page 254 of 281

and what checks should be made at runtime rather then development time.

Flexibility. All handling and clean-up of incorrect assertions is located in one place making changes to this logic much easier to implement.

Readability. Polices concerning how assertions are actually thrown and handled is not in the functional code.

Documentation. The code actually documents the design assumptions. This can also be used by documentation generators which read through the code.

The Assertion class can be defined as shown in the following specification:

Class Assertion
  void raise(boolean condition) throws AssertionException
  void raise(boolean condition, String description) throws
AssertionException
  void raise(boolean condition, Vector parameters) throws
AssertionException
  void raise(boolean condition, Vector parameters, String description)
throws AssertionException
  boolean isEnabled( )
  boolean isPerformanceSensitiveEnabled( )
Class AssertionException extends Exception

One possibility on how to handle the enabling and disabling of assertion checking would be to have two possible types of Assertion class. One which implements the actual assertion-checking logic and another which only implements no-ops. The Assertion instance is then obtains through an AssertionFactory which can be set as to which of the two types to distribute. These settings are determined at runtime.

It should also be noted that in Java, the exception that is thrown should be a generic run-time exception that doesn't need to be caught by the method or mentioned in the method's throw clause.

Collaborations

Factory

Distributed Garbage Collection

FIG. **138** illustrates a flowchart for a method **13800** for detecting an orphaned server context. A collection of outstanding server objects is maintained and a list of contexts is created for each of the outstanding server objects in operations **13802** and **13804**. A compilation of clients who are interested in each of the outstanding server objects are added to the list in operation **13806**. Recorded on the list in operation **13808** is a duration of time since the clients invoked a method accessing each of the contexts of the outstanding server objects. The list is examined at predetermined intervals for determining whether a predetermined amount of time has passed since each of the objects' has been accessed in operation **13810**. Contexts that have not been accessed in the predetermined amount of time are selected in operation **13812** and information is sent to the clients identifying the contexts that have not been accessed in the predetermined amount of time in operation **13814**.

After waiting a preselected amount of time for receiving a response from one of the clients, the context may optionally be deleted if a response from one of the clients is not received within the predetermined amount of time. Also, a response may optionally be received from one of the clients requesting that one of the contexts be maintained. In such a situation, upon receipt of the response, a time the context was last updated may be updated to a current time.

As a further option, a queuing delay may be accommodated for a response from the clients. Also, each of the clients may maintain a collection of all objects the client is interested in. The clients then may send requests to keep alive any objects the clients are currently interested in.

A client requests a server process but due to abnormal circumstances fails to clean up. How is the orphaned process detected and removed?

In the design of a stateful server, the LUW Context pattern facilitates the server process constructing domain objects at the request of the clients and maintaining these objects within a given context. Domain objects are entered into a registry with their appropriate context which the server maintains and updates when a request is received to create or delete an object. Each time a context is accessed then a notification is broadcast to the registry, regardless of a state change. With a simple context management, each time a context is referenced by a client a reference counter is incremented and similarly decrements when the reference is destroyed. Once the reference count returns to 0 then the context can be removed from the registry.

If the context is not explicitly deleted by the client then it will remain in the registry as the server has no way of detecting that the context is orphaned.

Even if the client application is rigorously designed to ensure all redundant contexts are deleted, an abnormal client event may result in its termination leaving an orphaned server context.

FIG. **139** illustrates a Client **1 13900** that has instantiated A **13902** and C **13904**, deletes C but fails to delete A.

The server still has a reference counter greater than 1 even though the client is no longer interested.

Therefore, Distributed Garbage Collection should be implemented to ensure that orphaned server contexts are deleted on the server. In the registry for the Garbage Collection the server maintains a collection of outstanding server objects and foreach object a list of its contexts, the clients currently interested and the duration since a method was invoked upon a given context by a client. Periodically this list is examined to establish if any of the objects have not been accessed for some configurable time and are candidates for reaping. So, for example, a value of minutes could serve as a default poll event or keep alive interval. If a candidate for a orphaned server process is identified then the clients are sent a message, requesting if they are still interested in the context. This might be performed by publishing an "is anyone interested" message to the registered clients to establish if anyone is interested in the object in its assigned context or by asking the clients explicitly depending on the nature of the architecture.

The client side also maintains a collection of all of the objects that it is interested in. When it is queried, it instructs the server to keep alive any objects it has an interest in for which a query has been received.

FIG. **140** illustrates a GarbageCollector **14000** requesting for interest in context A **14002**. No responses are received from any clients so the server assumes it is orphaned and deletes it.

If the period configured for a client to respond expires then the context is deleted. This accounts not only for an abnormal termination of the client but for failure of the client

SAP - Exhibit 1105
SAP v. Valtrus et al. - IPR2025-00420
Page 255 of 281

application to clean up. However, if a request is received from a client to maintain a context then the time the context was last accessed is updated to the current time and it remains in the Garbage Collection registry.

FIG. **141** illustrates a GarbageCollector **14100** requesting for interest in context B **14102**. Client **2** registers interest so the reaper updates the access time stamp and maintains B.

Benefits

Cleanup on the Server. Reduces the amount of redundant resources on the server to a minimum. This is especially important if a stateful component is held in a transaction by a client and the architecture prevents additional clients from accessing it, e.g. with BEA's M3.

Performance. Ensures that only the required contexts are maintained on the server, minimizing the work that the server is required to do, especially during the cleanup process at the end of a LUW.

Centralization. The collector has a central view over all of the contexts that are currently accessed by all of the clients within a given context. This simplifies the persistence of a context at the end of processing.

In order to prevent potential race conditions the client must be given sufficient time to respond to the keep alive message from the server before the context is deleted. Typically the client has a separate listener for upward messages originating at the server, so queuing is not an issue at the client end. However, a server is more likely to queue on the receiving end, especially in a system with high message rates.

Unless there is a dedicated listener on the server it must be configured to accommodate for any queuing delay on receipt of the client response.

Collaborates With

Context Pattern Language describes the architecture that is required before the Distributed Garbage Collection is required.

Variation Of

Java Shared Namespaces with distributed garbage collection.

Objectstore PSE WeakArrays.

Exception Hierarchies

FIG. **142** illustrates a flowchart for a method **14200** for creating a common interface for exception handling. Naming conventions of exceptions are determined in operation **14202**. A prefix and/or a suffix is added to each exception interface name in operation **14204** for indicating that the exception interface is an exception. In operations **14206** and **14208**, where an exception error occurred is indicated and a determination is made as to what caused the exception error. Context is provided as to what was happening when the exception error occurred in operation **14210**. Streaming of the exception is allowed to a common interface in operation **14212**. An error message is outputted indicating that an exception error has occurred in operation **14214**.

As an option, a layer and/or domain may be added from which each exception originates to each of the names of the exception interfaces. As another option, the exceptions may be partitioned into classes based on the way exceptions are handled, exceptions associated with different layers of a system, domains, and/or the source of the exceptions. As a further option, a class may be created which represents a source of the exception and holds an original copy of the exception for avoiding creation of duplicate exceptions. Also, arbitrary exceptions may each optionally support a clone method which creates a copy of the arbitrary exception.

Developing exception handling logic without classifying and organizing exceptions makes the handling logic cumbersome and fragile to change. How should exceptions be structured?

The traditional way of conveying errors is by passing error codes from callee to caller. This approach is adequate in some cases, but in general, it is less powerful and more error prone than an exception based approach. In the traditional approach, only minimal information can be passed, such as a failure to locate a configuration file (information on which file has to be provided by some other means). It is also very easy, and common, to ignore the return code. Projects which faithfully test every return code end up mixing a high percentage of error logic with the primary logic. This increases the complexity, and the development, review, and maintenance effort.

Some computer languages (Java, C++) support an error reporting mechanism based on exceptions. In these languages an exception can be a class type and hold arbitrary information, such as the name of the configuration file that was missing. Also, exceptions cannot be as easily ignored as return codes. If the callee raises an exception and the caller doesn't handle it, the caller's caller is checked to see if it handles the exception. This continues until the exception is handled or the program terminates. Designed properly, the error handling logic will be somewhat separated from the primary logic and will be less dense than the traditional approach.

The exception class designer is free to create any interface for the class, and each exception class can have its own unique interface. The exception handling logic **14300** will know which exception **14302** was raised (via runtime support) and can make use of the interface particular to the given exception. You can think of the exception handling logic being a set of "chunks" of logic where each chunk handles a specific type of exception. With this in mind, you can see how having many different exception types will cause the exception handling logic to grow. As a new exception type is added to the system, a new "chunk" might have to be added to the handling logic. This is not good. The code is not flexible to change and is in several places. Note FIG. **143**.

Suppose you have all these chunks of handling logic and discover that the logic is pretty much the same. For example, assume your architecture is layered and you want to treat all exceptions from the persistence layer the same, such as logging the error and notifying the user. Also assume that the persistence layer can raise any one of fifty exceptions, and more are expected to be added in the future. This is fifty chunks of code that must be present in the exception handling logic, and again, this logic may be in several places. Wouldn't it be nice to write one chunk of handling logic and be done with it?

Let's take another scenario. Suppose you want to prevent any raised exception from bringing down your system, as least not without a fight. In some cases the error will be unrecoverable and there is not much you can do but release resources (locks, communication channels, . . .) and terminate. What caused the problem is going to be on the tops of the minds of the production support people, and yours when you get their call (always in the middle of the night). You could write the exception handling logic chunks for each exception type—remembering that each exception has its own interface and will require separate logic to handle each interface—for each exception, but now you have to handle all the exceptions in the system. Wouldn't it be nice to write one chunk of handling logic and be done with it?

Therefore, to simplify the error handling logic and be able to treat groups of exceptions the same, a few techniques should be used to organize and define the exception interfaces.

The first step is to create an exception interface that all other interfaces will use or extend. It is not possible to provide one here as it greatly depends on the requirements at hand. But here are some guidelines:

- Determine the exception naming conventions. Use either a prefix or suffix to indicate that the interface is an exception. Also consider naming exceptions with the layer or domain they originate from. For example you may have an exception, CaAddressExcp, which is owned by the Customer Acquisition domain.
- Provide a means to determine where the error occurred (file, line, client or server, layer, . . .) so that it can be investigated.
- Provide a means to determine what happened (could not open file: XYZ).
- Provide context as to what was happening (Saving account information).
- Provide a way to stream the exception or stringify it.
- Consider separate production messages versus debug messages.
- Don't try to indicate severity. This is determined by the context of the caller, not the callee.

The intent is to be able to handle any arbitrary exception the same by having a common interface. Take time and get this right, to avoid updating several other exceptions later.

Now that this base exception interface is available, any handling logic can treat all exceptions alike; only one chunk of logic needs to be written. Specific exceptions can still be handled on a case by case basis as required. You can extend this concept to further partition the exceptions by creating a tree of exception types. By handling any exceptions at particular point in the tree, you effectively handle all exception types below that point. The trick is in creating a useful tree. Here are some guidelines:

Determine where handlers will be put and how they will respond to each exception. If you find that many are handled in the same way there may be a natural grouping that can be leveraged.

Consider the stability of your grouping. Is the group cohesive or is regrouping likely?

If parts of your system are layered, consider a branch that consolidates each layer. This enables a handler to deal with all exceptions emanating from a given layer.

Consider grouping by domains (Legal, Finance).

Consider grouping by subsystem

Consider common problems such as parameter validation, pre- and post-conditions

Consider the source (client or server).

FIG. **144** illustrates that groupings are not always exclusive. It is possible to group some exceptions **14400**, **14402**, **14404** by layer and then domains within that layer.

Benefits

- Simplicity. Simplifies handling logic by being able to write a handler that deals with the base exception type.
- Consistency. Consistent approach to error handling.
- Maintainability. Minimizes coding changes by reducing the multiple number error handling chunks.
- Manageability. Provides Conceptual Framework

The solution section covered many of the considerations in creating the exception tree so this section only provides some additional details to consider.

Wrapping and delegation can be used to simplify in certain situations. Consider a distributed application and the need or desire to handle server and client exceptions differently, or to know the source of the error. One way to avoid creating duplicate exceptions (one per source) is to create a class which represents the source and holds the original exception. For example AaServerExcp can hold a pointer to the base class AaExcp. The handling logic can catch AaServerExcp exceptions and then access the held exception. An alternative is to put a code in the base class with indicates source but then all logic needs to know to set this value and all handling logic needs to test for it.

To hold onto an arbitrary exception you need a way of creating a copy of it, but you may not know the actual type of the exception. In C++ the exception will be destroyed when you leave the handling logic, so you need the ability to create a copy to hold onto. A common technique is it have all exceptions support a "clone" method which creates a copy of themselves.

Consider how to stream an exception so it can be sent from server to client.

Exception Response Table

FIG. **145** illustrates a flowchart for a method **14500** for recording exception handling requirements for maintaining a consistent error handling approach. An exception response table is provided in which an exception is recorded in operations **14502** and **14504**. The context of the exception is entered in the exception response table in operation **14506** and a response for the exception is listed in the exception response table in operation **14508**. The response is subsequently outputted upon the exception occurring in the context in operation **14510**.

A typical response and a last resort response may be listed in the exception response table. The typical response may also be outputted upon the exception occurring in the context. The last resort response may be outputted upon the exception occurring out of the context. Additionally, abbreviations may be used to reduce an output size of the exception response table. Further, the exception response table may also include an exception category field for permitting organizing multiple exceptions by source. Optionally, an optimization may be determined that can be made based on similar entries in the exception response table. Further, the optimization made may also include classifying the exceptions for organizational purposes.

The response to an exception may vary per exception type and the context in which it is thrown, such as being thrown on the client or server, and the context in which it is handled. How do you record the exception handling requirements?

During exception handling design there are several aspects to capture to achieve a consistent approach:

- The set of exceptions to be handled
- The set of responses to these exceptions
- The context in which the exception is handled; e.g. client or server, batch or GUI

The set of exceptions to handle and their organization structure varies by project. Typically exceptions are organized into hierarchies to facilitate handling. The response to an exception may vary by exception type, the context in which it was thrown, and the context in which is handled. Here are some examples of error handling decisions of a hypothetical project:

- "All exceptions thrown on the server, and not handled by the server logic, will be propagated to the client."

"The current transaction is aborted if a server exception is not recoverable"

"All Server exceptions derived from Excp will be logged if not handled by the server code. The last resort handler will ensure this."

"GUI clients will display the error information in a splitter window"

"Batch clients will send error information to Operations"

These few examples demonstrate how context (Batch, GUI, Client, Server, last resort) can affect the handling of exceptions, and that even in a given context, the exception type may play a role in the handling. In a real system there may be several other context and exception-type specific requirements.

There are two common exception handling contexts that should be present in most systems. One is referred to as the Typical Response and the other is referred to as the Last Resort Response. The Typical Response is the error handling code intentionally added to handle exceptions. For example, car.start( ) is likely to fail due to being out of gas. The Typical Response may be to fill the tank and retry. The Last Resort Response is what to do when an exception is not handled (the Typical Response could not handle the error, such as a hole in the gas tank). Last Resort Response is a way of capturing what should be done when application code fails to handle an error. Recovery is usually not possible at this point but the handler may be coded to log the error and notify Operations of the problem. Without this response, systems may crash unnecessarily, or without indicating what happened.

All these permutations of exception types, contexts, and responses need to be managed in order to maintain a consistent error handling approach.

Therefore, use an Exception Response Table to capture the exceptions in the system, and the appropriate responses by context. What is important to capture is the exception, context, response, information; documenting the error handling requirements.

The following table lists exceptions by category and type, with the typical and last resort response. Other contexts and responses are listed within these columns. The exception category field is optional but can help to organize exceptions by their source (application, architecture, . . .) or hierarchy. This table can become quite packed with response information so a nomenclature may need to be developed to condense the information. The implementation section provides an example of this; Other ways of formatting this information are possible.

Exception	Typical Response	Last Resort Response
Exception Category Exception-Name Description ... Exception Category Exception-Name Description		

Benefits

Requirements Traceability. Exceptions requirements are captured and managed through implementation.

Hierarchy Design. Analysis may show optimizations that can be made such as handling a subtree of exceptions with the same code, as the response is the same to any exception in the subtree.

Interface Design. Discovery of interface requirements on the exception classes to support a particular response is another benefit.

Handler design. Assists in exception handling design by identifying common responses that can be leveraged by the handlers.

The table below shows an example of an Exception Response Table for a fictitious client/server system. This is followed by the nomenclature section which is customized per project.

Name	Typical Response	Last Resort Response
Architecture Framework Exceptions		
AaAssertionExcp Assertion failure	C: N/A S: N/A	C: L, Popup(severe), Shutdown S: L, N, P(AaServerAaExcp), Shutdown
AaExcp Base class for exceptions	C: N/A S: N/A	C: N/A S: N/A
Application Exceptions		
CaBalanceExcp Account out of balance	C: Popup(warn) S: P(AaServerAaExcp)	C: L, Popup(warn) S: L, N, P(AaServerAaExcp)

Nomenclature

Note: Abbreviations were used so that the table could be printed. The nomenclature section is only meant to serve as an example.

Context:
C = Client
S = Server
Response:
N/A = not applicable; don't handle
L = log error
L(diagnostic) = log errors for diagnostic purposes only
N = notify operations
Optional = application, context dependent. Not required to be caught
P = pass exception to client
P(<exception>) = pass given exception type to client, will be different from type caught
Popup(warn) = display warning message
Popup(severe) = display severe warning message
Popup(retry) = display retry message
Shutdown = release resources and shutdown gracefully.

Exception Hierarchy discusses how to organize exceptions.

Last Resort Exception Handling describes where handlers should be placed to prevent a program from terminating without warning.

Polymorphic Exception Handler describes how to design and code exception handlers that reduce the impact of changes and the overall size of the error handling logic.

Polymorphic Exception Handler

FIG. **146** illustrates a flowchart for a method **14600** for minimizing the amount of changes that need to be made to exception handling logic when new exceptions are added. Exceptions are organized into hierarchies in a polymorphic

SAP - Exhibit 1105
SAP v. Valtrus et al. - IPR2025-00420
Page 258 of 281

exception handler in operation **14602**. A root of one of the hierarchies in which an exception occurs is caught in operation **14604**. The exception is instructed to rethrow itself in operation **14606**. The rethrown exception is caught and identified in operations **14608** and **14610**. A type of the rethrown exception is determined in operation **14612** and a message is outputted indicating the type of the rethrown exception in operation **14614**.

Single exception interfaces may be used as the roots of the hierarchies. Also, the polymorphic exception handler may handle each unique root. Further, an added exception may be organized into a hierarchy and handled by the polymorphic exception handler. As an option, handling behavior may be encapsulated in the polymorphic exception handler. As additional option, catch blocks may also be created to catch the rethrown exception.

Large systems can be quite complex and require error management integrating disparate components and/or libraries (i.e. DBMS APIs, data structures library, middleware, etc) How can exception handling logic be written so that little or no changes are required when new exceptions are added to the system?

A software system using exceptions as the error handling approach may have to respond to a variety of exceptions. Handling each exception type on a case by case basis is cumbersome and expensive, both in terms of initial development and subsequent maintenance. In languages such as Java and C++, the mechanism to handle exceptions is to use try-catch blocks which look like this:

```
try
{
    // perform some work here
}
catch (ExceptionTypeA& excp)
{
    // Exception A thrown. Handling logic here
}
catch (ExceptionTypeB& excp)
{
    // Exception B thrown. Handling logic here
}
catch (...)
{
    // Don't know what was thrown, but still need to handle it.
}
```

This example shows only two explicit exception types being handled but a system typically has several potential exceptions. If the development of the exception types is poorly designed the try-catch blocks can become quite large as they attempt to handle each exception. Imagine trying to handle, say, fifty more exception types, in several places, in the code. The error handling code expansion is exponential! FIG. **147** depicts a program **14700** (i.e., the exception handler of the present invention) with a few try-catch blocks **14702**. As more exceptions are added these blocks expand to handle each new exception.

Another problem with exception handling logic is that it can be quite involved, such as logging the information to a persistent store, notifying Operations support, rolling back a transaction, etc. the example only showed one commented line to represent the code. Again, imagine each catch block requiring several lines of code. This logic may be repeated in each catch block.

Taken together, varying exception types and potentially repeating and complex logic in the catch blocks, the development and maintenance efforts regarding error handling are going to be much more expensive than they need to be.

Therefore, structure the exceptions into hierarchies, create an exception handler object that performs the catch block logic, and minimize the number of catch blocks required to support a given try-block.

Exception Hierarchies organizes exceptions into hierarchies and facilitates the design of exception handlers. Handlers can then be designed to handle the roots of hierarchies. This is much simpler than handling each exception type on a case by case basis. In custom development where the project has control of all code, a single exception interface can be used as the root. The more likely situation is some custom development and using third party libraries which may also use exceptions. In these cases, the exception handler will handle each unique root.

Using an exception handler, versus custom logic per catch block, reduces the maintenance and development effort as the code is easier to read, there is less of it, and any changes that need to be made can be made in one place.

The following code snippet shows the form of the try-catch blocks using the polymorphic exception handler. It may seem equivalent to the prior catch-block example but it is not. The first distinction is the type of exceptions handled. In this case, the roots of the exception hierarchies are caught, not the individual exception types. For this example there are only two exception hierarchies in the system, so only these roots are handled. What this means is that as new exceptions are added to the hierarchies, this code does not change, and remember, this code is in several places in the system.

The second difference with this code is the encapsulation of the handling behavior in the exception handler. The handle method can perform arbitrarily complex logic behind the scenes, and if this needs to change, is changed in one place. For example, if the current handling logic logs a message to disk and now needs to be extended to notify Operations personnel, this can be centralized in on place. The code as written does not need to change.

```
try
{
        // perform some work here
}
catch (ExceptionRoot& excp)
{
    ExcpHdlr hdlr;
        hdlr.handle(excp);
}
catch (ThirdPartyRoot& excp)
{
    ExcpHdlr hdlr;
        hdlr.handle(excp);
}
catch (...)
{
    ExcpHdlr hdlr;
        hdlr.handle();
}
```

FIG. **148** depicts the same program **14800** (the polymorphic exception handler) with smaller catch blocks **14802**. A handler **14802** has been added which consolidates the common code and the number of catch blocks has been reduced overall by making the handler responsible for handling each exception. The downside is that now the handler is subject to frequent change as exceptions are added to the system. The maintenance effort outweighs this disadvantage.

The examples have shown a single exception handler being used. In practice it is more likely that multiple will be used. For example, the exception handler on a server may have different requirements or constraints than a client, or one client may be GUI based and display pop-up error messages, where another client is a batch program that needs to send notification messages to Operations. This can be handled by creating multiple handlers or using the Strategy pattern to customize the behavior.

Benefits

Simplicity. Reduces development and maintenance effort required for exception handling

Maintainability. Reduces impact of changes

Robustness. Centralizes/Encapsulates handling logic

Flexibility. Multiple handlers can be used

The exception base class declares a method, rethrow, which is used by the handler to determine the real type of the exception. Another approach is to use double dispatch which may be show in a future version. Below is an example of this interface only showing the essential detail.

```
//-----------------------------------------
//- Base Class of Exceptions
//-----------------------------------------
class Excp
{
public:
    //- Rethrow the exception. Throw *this;
    virtual void rethrow() const = 0;
};
//-----------------------------------------
//- Example Derived Class of Exceptions
//-----------------------------------------
class Derived : public Excp
{
public:
    virtual void rethrow() const { throw *this; }
};
//-----------------------------------------
//- Example Derived Class of Exceptions
//-----------------------------------------
class SubDerived : public Derived
{
public:
    virtual void rethrow() const { throw *this; }
};
```

When the exception handler is passed the exception from the catch-block all it knows it that it has a root exception type. For some projects this may be sufficient if the exception interface is rich enough and all exceptions are treated the same. In other cases, exceptions may require specialized treatment. With the rethrow mechanism in place, the handler can create a try-catch block and have the exception rethrow itself. The catch blocks are then used to catch the specific exception type.

```
//-----------------------------------------
//- Exception Handler
//-----------------------------------------
class ExceptionHandler
{
public:
    ExceptionHandler();
    //- Handle the root exception
    void handle(const Excp& );
    //- Handle a third party root
    void handle(const ThirdPartyExcp& );
};
```

-continued

```
//-----------------------------------------
//- Handle the exception
//-----------------------------------------
void ExceptionHandler::handle(const Excp& e)
{
    //- Rethrow the exception to get the specific type
    //- Note that catches are in the order of most specific to
    //- most general.
try
{
    e.rethrow();
}
catch(SubDerived& excp)
{
    // Handle SubDerived
}
catch(Derived& excp)
{
    // Handle Derived
}
catch(...)
{
    // Handle e parameter here since nothing matched it.
    }
}
ExceptionHandler::handle(const ThirdPartyExcp& e)
{
    // Handle based on ThirdPartyExcp interface
    // Can't rethrow because ThirdPartyExcp doesn't support this.
    // Could use RTTI if needed.
}
```

Load Balancer

FIG. **149** illustrates a flowchart for a method **14900** for distributing incoming requests amongst server components for optimizing usage of resources. Incoming requests are received and stored in operations **14902** and **14904**. An availability of server components is determined and a listing of available server components is compiled in operations **14906** and **14908**. A determination is made as to which server component on the listing of available server components is most appropriate to receive a particular request in operation **14910**. Each particular request is sent to the selected server component determined to be most appropriate to receive the particular request in operation **14912**.

Optionally, the determination of which server component is the most appropriate may be performed by allocating the requests on a round-robin basis whereby requests are assigned to consecutive server components by traversing along the listing of available server components. As another option, the determination of which server component is the most appropriate may also include calculating an amount of utilization that each available server component is currently experiencing.

The amount of utilization of each available server components may be calculated based on current CPU utilization, kernel scheduling run-queue length, current network traffic at a node to the server component, and/or a number of requests currently being serviced. Also, a request may be rerouted to a different available server component upon a crash of the selected server component. Additionally, the server components may be saved in a persistent store, wherein a check is made to determine whether a connection to a server component needs to be reestablished.

In order to support scalability in a high volume distributed component environment, resources tend to be replicated. How can incoming requests be distributed amongst the available server components in order to optimize the usage of system resources?

SAP - Exhibit 1105
SAP v. Valtrus et al. - IPR2025-00420
Page 260 of 281

In a distributed system, server components provide functions and data that can be accessed by client components. Many identical copies of a server component can be running on different platforms in the system in order to support large volumes of client requests.

In order to make use of the system's scarce resources, some way of routing an incoming request to the best server component available is required. In general, all requests take a similar length of time to service.

FIG. **150** illustrates server components **15000** receiving service requests **15002**.

Therefore, use Load Balancer to select the best server component out of an available pool for the client to use.

FIG. **151** illustrates a load balancer **15100** mediating the requests of FIG. **150**.

Incoming client requests are routed by the Load Balancer to the best available server component.

A number of possible strategies exist for deciding which server component is the most appropriate at a given point in time.

Round Robin—Allocate the received requests on a round-robin basis, whereby a list of the available server components is created and, as requests are received, they are allocated by traversing down the list. When the end of the list is reached, the next request is allocated to the server component at the beginning of the list.

Utilization Based—Allocate the received requests based on the utilization that each server component is currently experiencing. The definition of utilization can be tailored to meet specific requirements or deployment strategies. It may be based on a combination of current CPU utilization, kernel scheduling run-queue length, current network traffic at that node, number of requests currently being serviced, or any other factors particular to the environment.

Benefits

Performance. Based on the selection strategy employed, the client is connected to the server component that is best able to serve it.

Scalability. As the number of users and requests increase, processing can be distributed across the available resources.

Robustness. In the event of the server crashing, the client can then ask the Load Balancer to provide another server component for it to use. This can be extended still further by federating Load Balancers and their associated server component pools.

The following is the IDL that was used to define the Load Balancer:

```
interface LoadBalancer
{
    Object getService ()
        raises (ArchitectureException );
    void register ( in Object aServerComponent )
        raises ( ArchitectureException );
};
```

Collaborations

Round Robin Load Balancing

Utilization Based Load Balancing

User Context

FIG. **152** illustrates a flowchart for a method **15200** for maintaining a security profile throughout nested service invocations on distributed components. In operation **15202**, interconnections are provided between distributed components each having nested service invocations. A user is identified in operation **15204**. The user is associated with roles in operation **15206**. In operation **15208**, a user context instance is created upon successful identification of the user. The user context instance also includes information about the user including the roles. A request is received from the user to invoke a service on a component in operation **15210**. The component invokes an additional service of another component. The user context is queried for the information about the user in operation **15212**. The user information is compared with an access control list for verifying that the user has access to the component in operation **15214**. The user information is also compared with an access control list for verifying that the user has access to the additional service of the other component in operation **15216**.

Optionally, all user interactions may be logged as well. As another option, a user interface may be modified to provide access to actions that can be performed by the user based on an identity of the user and the roles associated with the user. The user context instance may also be passed along as a parameter of service invocations. Additionally, the service invoked may associate any objects created, updated , or deleted with the user context instance. As a further option, the user context instance may also encapsulate security certificates of the user.

For security and auditing purposes, user information must be maintained throughout a service's implementation across multiple, distributed platforms. How can this original security profile be maintained throughout nested service invocations on distributed components?

All mission-critical systems require some form of security and auditing capabilities. These capabilities restrict who can use the system and what they can and cannot do and, in the case of a security breach or dispute, resolve who did what and when.

To meet these capabilities, users must be identified, associated with roles and granted authorization before any operation proceeds. In addition, all user interactions and results of those interactions may be logged. On a user interface, access to certain panels and controls are granted according to a user's role.

In a distributed, component-based system, these complex requirements become even more difficult to implement. Typically, a client (or user) invokes some service on a component. That component may invoke any number of additional services on any number of additional components to complete its designated task. These successive service invocations are a result of the initial client request so the security profile that allowed the initial request must also allow all successive requests.

FIG. **153** illustrates a component interaction diagram showing an interaction between a number of components in a financial system. A user initiates an addStock( ) service on the Portfolio component **15300**. To perform the addStock( ) service, the Portfolio must use the getStockPrice( ) and the deductFromAccount( ) services on the Market and Finance components **15302**, **15304**, respectively. This implies that a user who can access the addStock( ) service must also have permissions to access the getStockPrice( ) and the deductFromAccount( ) services. This may need to be checked by each of the distributed components within the context of one logical service. In addition, auditing what has been done, or perhaps requested to be done, adds another common requirement that must be accounted for. A compo-

SAP - Exhibit 1105
SAP v. Valtrus et al. - IPR2025-00420
Page 261 of 281

nent servicing multiple clients must associate client requests with corresponding services invoked on business objects. This information must be persisted as each change is committed.

Therefore, represent information about a user in a shared User Context object. This object maintains a user's unique identification that can be subsequently checked against a resource's access control list (ACL). A User Context instance is created upon a user's successful, validated identification to the system (usually through some "login" mechanism). After that, the system user interface can modify itself to provide only the actions that can be performed by that particular user acting in a particular role. Controls may query the User Context and modify their own visual state as needed (enable/disable, hide/show).

The User Context can also be passed along as a parameter of service invocations. All public, stateless services on a component should provide for a User Context to be passed along as a parameter. The service being invoked can then associate any Business Objects created, updated, or deleted as a result of the service invocation with the User Context.

One example of this would be a User Manager **15400** associating a User Context instance **15402** with the Business Objects **15404** they are affecting. FIG. **154** illustrates a user manger/user context relationship diagram.

These associations can be used for auditing purposes. When a change to a Business Object is committed, a log entry can be created tying the change with the user that triggered it.

Benefits

Common User Representation. One single representation of a user and their access rights can be shared across all areas of the system.

Extensible Security. Because there is one source for the User Context various policies or strategies could be used to identity and authenticate the User within a context. For example, it could encapsulate the User's certificates that allow more advanced security strategies to determine authorization.

```
Class UserContext
  UserContext(Identifier identifier)
  Identifier getIdentifier( )
  String getName( )
  void setName(String newName)
  void addRight(String accessArea, AccessLevel level)
  void removeRight(String accessArea, AccessLevel level)
  Vector getRights(String accessArea)
  boolean canCreateIn(String accessArea)
  boolean canReadIn(String accessArea)
  boolean canUpdateIn(String accessArea)
  boolean canDeleteIn(String accessArea)
Class AccessLevel
  static AccessLevel create( )
  static AccessLevel read( )
  static AccessLevel update( )
  static AccessLevel delete( )
  boolean=(AccessLevel anAccessLevel)
```

It is expected that the User Context will be passed from component to component. In this case the User Context will have to be defined using some sort of interface language definition (IDL).

Collaborations

Permission

Policy

SecurityManager

Logging

Alternatives

MTS & EJB offer an environment that does not require the passing of the context with every operation. A container as a set<context type> that provides a handle within the component for the methods to access the cached context.

INFORMATION SERVICES PATTERNS

Reliable information access mechanisms in a multi-user environment are a crucial, technical issue for almost all systems that a user builds.

Most business information systems manage data which must be saved in non-volatile storage (e.g., disk). The data must live, or "persist," between invocations of any particular application or program. Persistence is the capability to permanently store this data in its original or a modified state, until the information system purposely deletes it. Relational databases, object databases, or even flat files are all examples of persistent data stores.

This section discusses issues and approaches for developing an object-oriented persistence architecture.

A key issue frequently encountered in the development of object-oriented systems is the mapping of objects in memory to data structures in persistent storage. When the persistent storage is an object-oriented database, this mapping is quite straightforward, being largely taken care of by the database management system.

In the more common situation where the persistent storage is a relational database, there is a fundamental translation problem or a so-called "impedance mismatch". The physical, logical, and even philosophical differences between a relational and object data storage approach are significant. Mapping between the two is hard. The architecture must, in this case, include mechanisms to deal with this impedance mismatch.

The impedance mismatch is due to the following contrasting features of objects/classes and tables:

Identity: Objects have unique identity, regardless of their attributes. Tables rely on the notion of primary key to distinguish rows. While a relational DBMS guarantees uniqueness of rows with respect to primary keys for data stored in the database, the same is not true for data in memory.

Inheritance: This is a meaningful and important notion for classes; it is not meaningful for tables in traditional RDBMSs.

Navigation: The natural way to access and perform functions on objects is navigational, i.e., it entails following references from objects to other related objects. By contrast, relational databases naturally support associative access, i.e., queries on row attributes and the use of table joins.

The patterns in this section focus on problems and solutions associated with using a relational DBMS with an object-oriented persistence architecture.

A key objective of a comprehensive object-to-relational persistence architecture is shielding the application business logic and developers from the relational structure. The benefits are a simplified environment for business developers, reduced distraction with technical issues, and increased focus on the business object model and functional logic. However, in order to reap these benefits, a significant investment in architecture development is typically required.

The scope of a persistence architecture can range across the following levels of transparency and automation:

Heavyweight, fully-automated, including the mapping of the object model to the database schema and generation of all the database access code. Variants of this architecture type may allow the customization of database access code (e.g., for optimization purposes).

Lightweight mechanism which provides generic persistence capabilities to business objects but delegates all database access to separately developed data access routines. In this case, the data access routines are not part of the persistence architecture per se.

Minimal persistence approach in which each business object is directly responsible for database access

Of course, there is a tradeoff between transparency, automation, and flexibility on the one hand, and architecture complexity and development cost on the other.

The patterns in this section solve several of the fundamental problems encountered in the development of an object-to-relational persistence architecture, including the mapping of classes to tables (Data Handler, Individual Persistence), identity management (Object Identifiers as Object), caching(Object Identity Cache), allocation of responsibilities (Data Handler, Piecemeal Retrieval, Persistent State Separate from Persistent Object), and data access optimization (Multi-Object Fetch) and the mapping of basic SQL types to object attributes (Attribute Converter).

In addition to providing persistence capabilities, reliable information access mechanisms in a multi-user environment must support transaction semantics. As the real-life implementation of all of the patterns in this section requires integration with transaction management frameworks, the Persistence patterns should be considered and used in conjunction with the patterns in the Transactions section.

Attribute Converter

FIG. **155** illustrates a flowchart for a method **15500** for translating an object attribute to and from a database value. A database is provided and a conversion process is determined for converting an object attribute to and from a database value in operations **15502** and **15504**. The conversion process is encapsulated in an attribute converter. A first object attribute is directed to the attribute converter for conversion to a first database value in operation **15506**. A second database value is directed to the attribute converter for conversion to a second object attribute in operation **15508**.

A different attribute converter may be created for each type of conversion of object attributes to and from database values. In addition, the attribute converters may also implement a common interface. Further, all attributes of the same type may be directed to a particular attribute converter. Optionally, a second attribute converter may be substituted for the attribute converter for altering the conversion of the attribute. As an another option, the attribute converter may be altered for relieving a performance bottleneck.

Object attributes must go through some translation before they are written to and after they are read from some persistent stores. How can you isolate the translation algorithm from the persistent object and the persistence mechanism?

When interacting with a relational data store, the attribute value doesn't always map directly to a database type. Other times, an attribute value maps to more than one database type.

For example, in an Object based system, an attribute with a Boolean value is often converted from a Boolean object to a "T" or "F" string before it is saved in the database. In another example, a phone number attribute might be composed of an area code (**847**), an exchange (**714**) and an extension (**2731**). These three field might be saved in three separate database columns or combined into one before they are saved in the database.

FIG. **156** illustrates that an attribute **15600** can't be saved directly into the persistent store **15602**.

An impedance mismatch exists between the attribute and the data store and a conversion must take place.

The logic to perform this conversion can vary from one attribute to another. Based upon the attribute type, a different conversion must take place. In addition, special situations can arise where the same type of attribute will be stored differently in different situations. It is desirable to reuse this logic; however, the solution must be flexible enough that the developer is not locked into one single translation for an attribute type.

Therefore, use an Attribute Converter to translate database values to object attributes and vice versa.

FIG. **157** illustrates the use of an Attribute Converter **15700** to save an attribute **15702** into a database **15704**.

The knowledge of how to translate an attribute value to and from a persistent store is encapsulated in a separate Attribute Converter object.

The attribute's value should not be obtained directly from the attribute prior to saving it in the database. Nor should an attribute be instantiated directly from the raw value obtained from the persistent store. Values should be obtained or attributes created exclusively via an Attribute Converter.

It is recommended that a different Converter be created for each type of conversion required. This keeps the Converter's knowledge very specialized. As a result, the combination of Converters required to persist an entire object to a persistent store is very flexible due to the modularity of the Attribute Converter objects.

Benefits

Reuse. For some types of attributes, the conversion process can be rather involved. If this knowledge is encapsulated in an Attribute Converter, it can be reused for converting other attributes of the same type.

Flexibility. If the conversion for a specific attribute needs to be altered, simply substituting a different Attribute Converter will alter the behavior and not disrupt the rest of the application.

Maintainability. Altering a single Attribute Converter can affect several attributes in the system. For instance, if the conversion of one specific type of attribute is identified as a performance bottleneck, altering the corresponding Converter can benefit a large part of the system.

Ideally, all Attribute Converters should implement a common abstract class or interface. This allows the architecture to treat all Converters equally. The architecture need not know the specific translator class it is using.

The interface may look something like this.

```
public interface AttributeConverter
{
    public String translateValueForDataStore(Object anAttribute);
    public Object translateValueFromDataStore(String aColumn,
java.sql.ResultSet aResultSet);
}
```

The first behavior, translateValueForDataStore, takes an attribute and translates it into a String that can be used in an

SQL statement. The second behavior, translateValueFromDataStore, takes a column name and JDBC result set as arguments. It then answers the attribute translated from the given result set. Each implementation of AttributeConverter must then implement both behaviors in their own specific way.

```
public class BooleanTranslator implements AttributeConverter
{
    public String translateValueForDataStore(Object anAttribute)
    {
        String value = null;
        if(anAttribute != null)
        {
            if(((Boolean)anAttribute).booleanValue( ))
            {
                value = "'T'";
            }
            else
            {
                value = "'F'";
            }
        }
        else
        {
            value = "NULL";
        }
        return value;
    }
    public Object translateValueFromDataStore(String aColumn,
java.sql.ResultSet aResultSet)
    {
        Boolean result = null;
        String value = null;
        value = aResultSet.getString(aColumn);
        if(value.equalsIgnoreCase("T"))
        {
            result = new Boolean(true);
        }
        else if(value.equalsIgnoreCase("F"))
        {
            result = new Boolean(false);
        }
        return result;
    }
}
```

The Boolean Converter above knows how to translate a Boolean object to and from a character representation in the relational database.

Collaborations

Normalized Mapping—The Mapper contains all information required to store an object in a relational data store. Attribute Converter can be utilized by Normalized Mapping to store the knowledge needed to properly translate attribute state values to and from the persistent store.

Denormalized Mapping—Denormalized Mapper is another pattern for mapping objects to a relational database. The Attribute Converter pattern could be used by Denormalized Mapper to provide conversion between attribute values and database values.

Alternatives

Case Statements—Case statements aren't really a pattern, but they are an alternative to Attribute Converter. A case statement could be implemented in the super class to handle the translation of the data.

Data Handler

FIG. **158** illustrates a flowchart for a method **15800** for controlling data. A data retrieval mechanism is provided in operation **15802** for retrieving data from a database. The data retrieval mechanism also writes data to the database. In operation **15804**, 'the data retrieval mechanism is encapsulated in a data handler. A request from a domain object is received for a retrieval of a portion of the data in the database in operation **15806**. The data retrieval mechanism is utilized in operation **15808** to retrieve the portion of the data from the database. The portion of the data is passed through the data handler to the domain object in operation **15810**.

The data retrieval mechanism may be capable of being used by a plurality of domain objects. Also, the data retrieval mechanism may capable of being used by only one of a plurality of domain objects. Dependencies on the data retrieval mechanism within the data handler may also be managed via code generation.

The data handler may physically partitioned into a component separate from the domain object. Optionally, the domain object may write attributes to a data stream. In such a case, the data handler may define an order in which the attributes are written to the data stream. Also, a row class may define the attributes in the same order as the attributes appear on the database. Further, the data handler may iterate over the attributes and may save them to the database.

Business Objects in memory generally store and retreive their data members from some type of persistent store. When using Individual Persistence, how can we ensure that the retrieval mechanism used by the domain object is independent of the business logic?

Individual Persistence assigns responsibility for data access at the level of individual domain objects. Each domain object or class can retrieve, update, insert, and delete its data from a persistent store independently of other objects or classes. This promotes encapsulation and reuse across business transactions.

FIG. **159** illustrates the data retrieval mechanism calls being placed directly within the domain object **15900** (in this example SQL is inserted into the Account business object).

When persistence is at the class level, it is typical to code the actual SQL, serialization, or CICS call directly in the class itself. In the example shown above, an "Account" object can contain the SQL needed to retrieve and save its state to the database.

This approach can reduce the flexibility of the domain object, in that, changes to the access logic or the backend database must result in changes to the business object or class. How can we ensure that the business logic is independent of the data retrieval mechanism for such a class?

FIG. **160** shows the interrelationship between a database **16000**, a persist **16002**, and an account **16004**.

Business objects delegate their data retrieval mechanism to an appropriate handler. This Data Handler can be either be generic or specific to each type of domain object used. To minimize the impact of changes, dependencies on the database schema or data retrieval mechanism within the handler could be managed via code generation. In this manner, the physical data access is separated from pure business logic.

FIG. **161** illustrates that the database retrieval mechanism is separated from the business object **16100** by encapsulating the logic within a data handler **16102**.

Benefits

Loose Coupling of Data Access. The business object is independent of the database access logic and the backend database. As a result, the method by which the domain object accesses the persistent store can be changed without impacting existing source code.

Distribution. Data handlers can be physically partitioned into a separate component from the business logic. For

example, the data handler could be on a data server component near the DB, while the business logic is in an application component.

Multiple Data Handlers. Different strategies can be implemented based upon specific requirements. For example, on the client we can use serialization to communicate with the server; whereas the server can use standard DB access to communicate with DB.

Support for Testing. Similarly, during testing, hard-coded data handlers can be created to return dummy data. These can then be replaced at run-time or later in testing without impacting the code.

The following information focuses on the implementation of the Data Handler pattern (TiMapper) and the separation of the business domain objects from the data retrieval mechanism used on the project.

FIG. **162** illustrates the TiPersistenceStream **16200** and TiMapper **16202**.

TiPersistenceStream and TiMapper

Within the Rapid Batch Persist Service, objects save and load themselves by writing to or reading from a Persistence Stream. This is undertaken via the base class (TiPersist) with specialized streaming code created via the Creation Code Generator. As a result domain objects are only "aware" of how to stream themselves, and not how the data storage mechanism works.

The first attribute an object writes to the stream is its CLASS_ID. The stream then expects the other attributes to be put to the stream in the order defined by the mapper class (the data handler). This relationship between data handler and domain object is controlled via a row class, which defines the attributes in the same order as they appear on the DB (this class is also created via the Code Generator).

When the end of the stream is reached, the mapper class iterates over the list of attributes within the row and saves them via embedded Pro*C. When loading the reverse happens, in that, the mapper loads the information from Oracle and then populates a row based upon the CLASS_ID pulled from the database.

TiMapper

A Mapper for a class contains the columns(s) and table that the class will be written to, the type of the data and the order that they will be read from/written to the stream. It also reads and writes rows of data from the database. It generates a where clause from the primary key information (PID). A database runtime context is obtained from the Transaction Service (using the current implicit transaction context). It also contains the code to query for sequence ids, for classes that use optimistic locking.

The mapper contains the data retrieval mechanism that interacts with the Oracle database instance. As a result, if a different technology is used to interact with Oracle (e.g. stored procedures, embedded Pro*C, Method/**3** Pro*C) only the mapper class needs to change.

TiRow

A row contains the data to be written to a database row from a stream or to be written to a stream from a database row. It knows the column names on the table and knows the order in which they are read from streams.

TiMapperManager

The mapper manager is responsible for creating mappers for a given CLASS_ID. Each type of mapper registers with the mapper factory when the shared object is loaded into memory (using the dynamic registration factory pattern). The mapper factory is a singleton read only object.

The Factory pattern can be used to create the appropriate Data Handler for a specific business object. This pattern enables the data access method to be changed at runtime (e.g. batch mode, online mode or Request Batcher).

The Stream-Based Communication pattern can be used to stream the business object's data to the handler. The stream can then be either forwarded to a Request Batcher or can parsed and sent to database.

Individual Persistence

FIG. **163** illustrates a flowchart for a method **16300** for organizing data access among a plurality of business entities. Data about a user is retrieved and packaged into a cross-functional data object in operation **16302** and **16304**. A request for data is retrieved from one of a plurality of business objects in operation **16306**. In operation **16308**, the business object are directed to the data object such that the business object retrieves the entire data object. The business object also selects the data from the data object.

Both locking and integrity may use a uniform mechanism. The business object may retrieve account, customer, and bill-related data from the data object. Also, the business objects may be able to update themselves independently of each other.

Optionally, new business objects may take advantage of existing data access routines. Also, each business object may use a uniform access architecture.

Create a data access architecture that supports reusable, independent business objects in the context of atomic, functionally-specific transactions.

A business unit of work, or business transaction, typically acts on multiple business entities. But for each individual entity, the transaction might only display and update a few data attributes.

For example, accepting bill payment over the phone might use the account number, customer name, amount due, date due, and credit card number. The transaction could therefore avoid accessing many attributes of the account, customer, or monthly bill entities. This might naturally lead to a data architecture which only fetches required attributes, based on the transaction's needs.

Indeed, a traditional client/server program retrieves data in a piecemeal fashion. In this case, the example program would typically allocate a single record to fetch and store the required data items. Then, an "accept bill payment" data access module would fill this structure. This couples data access to processing function.

FIG. **164** illustrates retrieving data piecemeal.

This traditional style of data access may seem flexible. After all, developers can grab whatever data they need for a particular business transaction.

But such access is very unstructured. Different pieces of a cohesive account entity, for example, scatter across multiple windows. Some windows will use the account's effective date; others will not.

This also introduces redundancy. Retrieving the date requires determining the correct database, table, column, and type declaration. Yet another developer who needs this date, for a different data set, duplicates the effort. This does not encapsulate changes, thereby raising costs for testing, maintenance, and extension.

Moreover, each transaction must hand-craft its own retrieval procedure. Creating the thousandth new business transaction would require creating a thousandth new access module. Yet all data items would already have been retrieved by other modules. This style of data access is not reusable.

Finally, business entities are typically less likely to change than the transactions, or processes, which act on those

entities. For example, an enterprise might re-engineer its billing function. Regardless of the resulting process, the account, customer, and monthly bill objects would likely remain. This suggests that transaction-based data access is brittle.

Therefore, data access should be organized around business entities, rather than transactions. Individual Persistence packages data into cross-functional objects, rather than transaction-specific data structures. Each individual business object, instead of the window or application, manages the retrieval of its data items.

A business object provides public methods for accessing, comparing, displaying, and setting that data. Developers can therefore no longer plunder the persistent store, selecting data items at will. Instead, they must access their data through encapsulated, self-requesting business objects.

With this architecture, the example billing function retrieves account, customer, monthly bill, and bill payment objects.

FIG. **165** illustrates the manner in which the present invention retrieves whole objects **16500**.

For reuse, business objects should be able to request and update themselves independently of each other. Separating the data access for customer and account objects supports reusing them in isolation. Objects should therefore avoid explicitly requesting other linked objects, unless a formal containment relationship exists.

Finally, separation of concern allows business objects to remain blissfully unaware of the transactions which use them. A business object will not know which data items or services it may need to provide to its clients. Thus, the object must bring back all its data.

Benefits

Reuse. New transactions can take advantage of existing data access routines. For example, introducing a new business transaction, like perform credit check, would use existing customer and account objects. Yet, these domain model objects would already know how to update themselves. Therefore, the new application would build no new data access code.

Maintainability and extensibility. This approach supports "fix in one place." Any changes to particular data elements only need to be changed, tested, and recompiled in one access module, that of the owning business object.

Uniformity. Both optimistic locking and referential integrity (RI) are typically defined at the business object level. For example, separate account and customer objects typically have their own locking attributes. In addition, an RI rule usually relates one entity to another, such as "all accounts must have a customer." Organizing data access around business entities simplifies locking and integrity. Both can use a uniform mechanism, implying that the architecture can hide technical details. This avoids the hard-coding typical of the transaction-based approach.

Flexibility. Whole object retrieval is extensible. It allows a transaction to ask an object for any data. This supports maintenance and extension. A developer can easily change the particular data items a transaction uses. But whole retrieval still guarantees that those items will already have been retrieved. For example, a future release of the accept bill payment window could also display the social security number. Yet the data access routines would need no modification.

Each typical business class will support the standard CRUD flag capabilities, of:

Create

Retrieve

Update

Delete

This access architecture is uniform across business entities. The architecture can therefore standardize much of the processing. For instance, the architecture can handle, across business objects: dirty checks, CRUD flag management, optimistic locking, referential integrity, security checking, commit scope, request formatting, object streaming/unstreaming, message compression, and error handling. Moreover, business entities should support these capabilities through a consistent, polymorphic interface.

For example, all business objects could respond to the saveData message, to persist any changes saveData could first check, privately, if the object was even modified.

Then, using private CRUD flags, it could determine whether a save translates into an insert, update, or delete. Finally, saveData could stream out the business object's attributes, based on a defined layout. Then, a transaction persists its business objects by simply iterating over the collection, sending each member saveData.

The architecture should also encapsulate the data access protocols or products. For example, whether the business objects use a relational or object DBMS should be transparent to calling programs. This minimizes the impact of changes to the storage technology.

Individual Persistence naturally leads to multiple, small-grained request messages per transaction. Request Batcher solves performance problems with multiple network messages.

Data Handler encapsulates data access code from business objects. This protects business logic from changes in data access protocols and products.

Request Sorter handles referential integrity and deadlock avoidance in a uniform manner.

Multi-Object Fetch

FIG. **166** illustrates a flowchart for a method **16600** for retrieving multiple business objects across a network in one access operation. In operation **16602**, a business object and a plurality of remaining objects are provided on a persistent store. Upon receiving a request for the business object in operation **16604**, it is established which of the remaining objects are related to the business object in operation **16606**. The related objects and the business object are retrieved from the persistent store in one operation and it is determined how the retrieved related objects relate to the business object and each other (see operations **16608** and **16610**. A graph of relationships of the business and related objects is instantiated in memory in operation **16612**.

An object navigation pattern of accessing the business object and then accessing relationships with the related objects may be used to retrieve the related objects. The relationships between the business and related objects for instantiating the graph of relationships may also be determined from a source object, a set of target objects, and the name of the relationship. Additionally, the establishment of which of the remaining objects are related to the business object and the determination of how the retrieved related objects relate to the business object and each other may be pre-processed before retrieving the selected related objects and the business object from the persistent store.

As an option, a portion of the objects may also be retrieved from a cache. Also, as another option, a batch

SAP - Exhibit 1105
SAP v. Valtrus et al. - IPR2025-00420
Page 266 of 281

request may be sent to the persistent store for retrieving the remainder of the objects.

It is not unusual to retrieve multiple business objects within a unit of work. How can the persistent objects involved in a unit of work be efficiently retrieved?

A given business object maintains associations to several other business objects. A given unit of work needs to access a subset of the defined associations. In order to perform the unit of work, the business object and the required subset of associated objects must be retrieved from persistent store.

In the course of performing a unit of work, a set of related objects needs to be accessed. Typically, one starts from a "root" object and "navigates" relationships to access related objects. This process can be repeated on the related objects.

An entire graph of related objects can be accessed in this manner. A natural way to retrieve these objects is through lazy instantiation, i.e., objects are retrieved from persistent store as each relationship is traversed. This retrieval pattern typically requires multiple database/network accesses and can have serious performance implications, especially over a WAN.

Therefore, a mechanism is needed to perform the retrieval from persistent store in one access operation. This mechanism includes:

- Support for a declarative multi-object fetch statement which defines what is going to be fetched. This multi-object fetch statement does two things. It declares what is going to be fetched. It also declares how the objects that are being fetched relate to each other.
- Retrieval of the persistent data corresponding to the multi-object fetch statement.
- Instantiation of the graph of related objects in memory.

Benefits

- Performance. Performance is increased by making a single trip across the network and a single database access to fetch several instances of objects that participate in a transaction. The savings can be especially noticeable over a high-latency WAN.
- Continuity. Within the application code, the object navigation pattern of accessing an object and then accessing relationships from that object can still be used to access objects.
- Complexity. Requires more elaborate architecture than lazy instantiation.

FIG. **167** illustrates an example of an implementation of the Multi-Fetch Object **16700**.

FIG. **168** illustrates the Fetching of a Household object **16800** along with the other related objects using the multi object fetch results.

FIG. **169** is an interaction diagram showing when the multi object fetch is not used.

Note that if there is a large household, and each hobbyist in the household has lots of hobbies and interests, several trips to the server will be made to fulfill this query. There needs to be a multi-object fetch specification that keeps enough detail to know what needs to be fetched and how the fetched object will relate to each other. Here is a structure that will capture that information.

```
struct MultiObjectFetchSpec
{
   AxysClassId classId;
   char **associationName;
};
```

```
const HOUSEHOLD_CLASSID = 1;
const HOBBYIST_CLASSID = 2;
const HOBBY_CLASSID = 3;
const INTEREST_CLASSID = 4;
const NUMBER_OF_HOUSEHOLD_RELATIONSHIPS = 1;
const NUMBER_OF_INDIVIDAUAL_RELATIONSHIPS = 2;
static char *EmptyRelationship[1] = { 0 };
static char *
HouseholdRelationships[NUMBER_OF_HOUSEHOLD_
RELATIONSHIPS + 1] = {"Hobbyists", 0};
static char *
HobbyistRelationships[NUMBER_OF_INDIVIDAUAL_
RELATIONSHIPS + 1] = {"Hobbies", "Interests", 0};
static MultiObjectFetchSpec HobbyInterestMofSpec[5] =
{
        {HOUSEHOLD_CLASSID, HouseholdRelationships },
        {HOBBYIST_CLASSID, HobbyistRelationships },
        {HOBBY_CLASSID, EmptyRelationship },
        {INTEREST_CLASSID, EmptyRelationship },
        { 0,0 }
};
```

There is a class MultiObjectFetch that performs the fetch and associates all of the related objects that have been fetched so that when these related objects are accessed there is no further access to the database. The MultiObjectFetch class uses the MultiObjectFetchSpec to determine how the objects fetched relate to each other. This implementation assumes that the persistent framework being used can fill in the relationship given a source object, a set of target objects and the name of the relationship.

```
class MultiObjectFetch
{
    public:
    MultiObjectFetch *MultiObjectFetch(
        MultiObjectFetchSpec *mofSpec);
    PersistentObject *fetch();
    RWOrdered *fetchRows();
};
```

There is an assumption that the Household, Hobbyist, Hobby and Interest business objects inherits from a common base class, PersistentObject. If the restriction on the household is to bring back one Household, fetch( ) would used. If the restriction on the Household will bring more than one Household, fetchRows( ) would be used. The fetch and the fetchRows brings back the Household objects(s) and the related Hobbyists, Hobbies and Interests.

Static Approach (using Stored Procedures)

A stored procedure would be written that would retrieve the Household object(s), Hobbyist objects related to the Household objects, Hobby objects related to the Hobbyist objects and the Interest objects also related to the Hobbyist objects. It is important that the stored procedure fetch the objects in this order since the multi object fetch spec declared that this is the expected order. These fetched objects would then be passed to the MultiObjectFetch object which would fill in all the relationships of the returned object using the fetch specification.

Dynamic Approach (Dependent Requests/Pending Actions)

The multi-object fetch can be pre-processed before sending the request to the database. Any objects that can be fetched from the cache will be fetched from the cache. The remaining requests that cannot be satisfied from the cache will then be sent as a batch request to the database. This requires complex processing to determine if the cache can be used. This dynamic processing assumes that dynamic SQL

SAP - Exhibit 1105
SAP v. Valtrus et al. - IPR2025-00420
Page 267 of 281

will be used since it is not known at design time what objects will be found in the cache and what objects still need to be fetched from the database.

Dependent Request—used in dynamic approach

Request Batcher—used in dynamic approach

Batching Update—similar to bathing of fetches but used to batch updates

Relationship Stereotype—The setAssociation method is called to fill in the relationships

Object Identifiers as Objects

FIG. **170** illustrates a flowchart for a method **17000** for implementing an association of business objects without retrieving the business objects from a database on which the business objects are stored. In operations **17002** and **17004**, a business object is provided and an instance of an associated object is stored on a database. An association of the business object with the instance of the associated object is determined in operation **17006**. In operation **17008**, an object identifier is generated containing information including the determination association which is necessary to retrieve the instance of the associated object from the database. The object identifier is loaded when the business object starts in operation **17010**. A location of the instance of the associated object on the database is determined in operation **17012** from the object identifier and the instance of the associated object is retrieved from the database in operation **17014**.

The object identifier may be used to provide a unique identity that is required for implementing caching and identity management. Also, the object identifier may include a unique row identifier generated by the database, an identifier generated by a utility, and/or a unique string generated from one or more attributes. As an option, different types of business objects may be provided. In such a case, a different class of object identifier may be generated for each type of business object. As another option, the determination of a location of the instance and the retrieval of the instance of the associated object may also include the taking the object identifier as an argument and returning the instance of the associated object.

Most useful business objects have a relationship, or association, with other business objects. How should this association be implemented without having to read the associated object's state from the database?

Most useful business objects have a relationship, or association, with another business object instance. Traditionally, this relationship is expressed as a pointer or reference to the object. However, pointers (and references) are memory constructs valid only so long as the object state exists in memory. When storing the object to a persistent storage medium (such as a relational database) these associations need to be expressed in some other way. Likewise, when the object is restored from persistent storage, the associations need to be reinitialized since it is unlikely that the associated object will reside in the same memory location. If the associated object is stored in the persistent medium, this usually involves restoring it as well. This can become a long and expensive process if the association graph corresponding to the restored object is large or complex. It is particularly undesirable if the associations are not even traversed by the application.

Therefore, implement the associations using object identifiers that contain the necessary information to retrieve the object if it is needed. These objects can then be loaded when the object is restored, eliminating the need to restore the entire associated object In addition, since these object identifiers uniquely identify an object instance; they can be used/passed in place of memory pointers. When the object is needed, simply restore the instance using the object identifier.

Benefits

Performance. Objects are retrieved only when they are needed.

Caching and identity management. Object Identifiers can be used to provide the unique id needed to implement caching and identity management.

The object identifier (or OID) must contain enough information to uniquely identify the instance. This identifier could be a unique row id generated by a database, a UUID generated by a utility or a unique string generated from one or more attributes. It is generally desirable to have a different class of OID for each type of object, thereby creating a more type-safe environment. It should also be noted that OID's should have value semantics.

In addition, a mechanism must be available to retrieve objects given their OID. This can be as simple as a static (or class) method such as getById that takes the OID as an argument and returns the instance of the object. A more sophisticated approach would be to implement this and other persistence related methods in a generalized utility class. Below is a simple example that illustrates the relationship between two classes using object identifiers. Please note that this example is an extreme simplification. A useful implementation of this pattern would exist as part of a persistence framework that would include many additional methods and abstractions.

```
class FooId
public:
    // accessors, constructors and destructors
private
    long _id;
};
class Foo
{
public:
    // accessors, constructors and destructors
    static Foo* getById(FooId& id);
};
class Bar
{
public:
    Foo* getFoo()
{
    return Foo::getById(FooId);
            // The caller now owns the instance of Foo. Use of an
            // auto_ptr here is highly recommended.
    }
private:
    FooId _fooId;
};
```

Collaborations

Identity Manager—Uses Object Identifiers as unique key's for storing persistent state objects.

Persistent State Separate from Persistent Object—Uses Object Identifiers embedded in persistent state objects to eliminate coupling.

Object Identity Cache

FIG. **171** illustrates a flowchart for a method **17100** for mapping of retrieved data is into objects. An object is retrieved from a data store and cached in operations **17102**

SAP - Exhibit 1105
SAP v. Valtrus et al. - IPR2025-00420
Page 268 of 281

and **17104**. A unique object identifier is assigned to the object in operation **17106**. The object identifier is mapped to a representation of the object in a dictionary in operation **17108**. When a request for a reference to the object is received in operation **17110**, the object identifier of the object is retrieved from the dictionary in operation **17112**. The requested reference is associated with the representation of the object stored in the dictionary in operation **17114**.

In one embodiment, a data store may be accessed if the object identifier is not found in the dictionary to retrieve the object so that the process may be repeated with the retrieved object. In another embodiment, a query may be performed to retrieve multiple objects. A check may be performed to verify that each object is already cached so that objects not already cached may be cached.

Also, if an object in the cache has changed since read, an error may be raised if the object retrieved has changed since read and the object retrieved may be ignored if the object retrieved has not changed since read. If an object in the cache has not changed since read, the object in the cache may also be replaced with the object retrieved if the object retrieved has changed since read and the object retrieved may be ignored if the object retrieved has not changed since read. Further, if a query is guaranteed to return at most a single object, the cache may be used prior to going to the data store by sequentially applying the function to each object in the cache and implementing a predicate function which answers whether or not a given object satisfies the query.

Within a client context (e.g., a logical unit of work), the same object may be referenced more than once. How can object identity be preserved and redundant accesses to persistent store be avoided?

(Although this pattern is not specific to relational databases, we will, for the sake of concreteness and clarity, describe the pattern in terms of an object-to-relational mapping.)

Objects can be stored in and retrieved from a relational database. A retrieval strategy that simply translates relational data into objects in memory will almost certainly result in the instantiation of multiple copies of the same object. Furthermore, such a strategy is inefficient as the same data may be repeatedly and unnecessarily read from the database. This violates object identity and contributes to performance problems.

Suppose the class Account has an association with the class Customer. Suppose the instance of Account ABC is associated with the instance of Customer **123**. The following demonstrates multiple requests to Customer **123**.

Example:

customer**123**=getCustomer(**123**)

accountABC=getAccount(ABC)

secondRefererenceToCustomer**123**=accountABC.get Customer( )

Note that customer**123** and secondRefererenceToCustomer**123** are the same customer. In this scenario, the desired behavior is to have the data store accessed once for customer**123**. Also there should only be one instance of customer**123** in memory customer**123** and secondReferenceToCustomer**123** should reference this instance of the customer**123**.

Therefore, the mapping of retrieved data into objects should be mediated by an Identity Cache. FIG. **172** illustrates an Object Identity Cache.

Logically, an Identity Cache is a dictionary which, for each cached object, maps a unique object identifier to a representation of the object. Each object must be assigned an object identifier (OID) which uniquely identifies the object over the life of the system.

The mediation mechanism works as follows: When a reference to an object is requested, the identity cache is consulted. If the object's OID is found in the cache, the requested reference is associated with the representation of the object stored in the cache. If the OID is not found in the cache, the data store is accessed. The object representation that is retrieved from the data store is inserted into the cache and the requested reference is associated with it.

Benefits

Performance. Performance is improved for frequently accessed objects since they are only fetched from the database once.

Identity Preserved. Object identity is preserved since objects are cached based on the objects OID.

A dictionary can be used to implement the Object Identity Cache. The following points should be considering when implementing an Object Identify Cache.

A query could be performed that returns multiple objects. Each object that is retrieved must be checked to see if it is already in the cache. If it is not in the cache it must be inserted. The following shows what should be done when an object is retrieved that is already in the cache to get is correctly inserted into the cache.

	Object retrieved has changed since read	Object retrieved has not changed since read
Object in cache has changed since read	Raise an error. At commit transaction there will be an optimistic lock failure so it is better to raise it now.	Ignore the retrieved object, the object in cache is newer and the changes should not be lost.
Object in cache has not changed since read	Replace the object in cache with the object retrieved since the retrieved object is newer.	Ignore the retrieved object, it is already in the cache.

If the object is not in the cache when the object is retrieved, a simple insertion into the cache can be done.

If a query is guaranteed to return at most a single object, the cache may be used prior to going to the database by:

implementing (for a given class) a predicate function which answers whether or not a given object satisfies the query

sequentially applying the function to each object in the cache.

A multiple row query must go to the database unless there is a mechanism to indicate that the class is completely cached. This is applicable to static reference data.

Life Time of cached objects can affect Cache size. If life time of the cache is associated with a transaction there will be no problem. If the life time of the cache is associated with a longer lived entity such as a thread or a process, removal of objects from the cache must be actively managed. Two commonly used strategies are reference counting and LRU purging.

Collaborates With

Object Relational Query pattern describes a mechanism for storing objects in a relation database.

Object Identity

Persistent State Separatefrom Persistent Object

Used by

Context Management

Each Context (e.g., transaction, thread, etc.) owns an Identity cache which holds all of the objects in that context.

SAP - Exhibit 1105
SAP v. Valtrus et al. - IPR2025-00420
Page 269 of 281

Persistent State Separate from Persistant Object

FIG. **173** illustrates a flowchart for a method **17300** for separating logic and data access concerns during development of a persistent object for insulating development of business logic from development of data access routine. A persistent object being developed is accessed and a state of the persistent object is detached into a separate state class in operations **17302** and **17304**. The state class serves as a contract between a logic development team and a data access development team. Logic development is limited by the logic development team to developing business logic in operation **17306**. In operation **17308**, data access development is restricted by the data access development team to providing data creation, retrieval, updating, and deletion capabilities.

The business logic team may develop persistent objects that manipulate the state of the persistent object in accordance with business requirements. In one embodiment, the state may be implemented as a structure of values of basic data types. In another embodiment, the state class may contain member variables of lower data types including basic data types, extended basic data types with value semantics, other state classes, and/or vectors of the basic data types, the extended basic data types with value semantics and other state classes.

Optionally, the state class may support data structures of arbitrary shapes. Supporting classes may manipulate the state in a polymorphic fashion. As another option, the state may be further implemented as a class that contains key-value attribute pairs. The state class may also contain a keyed data structure containing attribute names and attribute values. Additionally, the state can also be asked to write data to a stream.

When designing and implementing persistent objects, how do we effectively insulate the development of business logic from the development of database access routine?

Given the use of a relational database as the persistent store, the scope of a persistence architecture can range across the following levels of transparency and automation:

- Heavyweight, fully-automated persistence architecture. Including the mapping of the object model to the database schema and generation of all the database access code
- Variants of the above scheme allowing the customization of database access code
- Lightweight mechanism which provides generic persistence capabilities to business objects but delegates all database access to separately developed data access routines. In this case, the data access routines are not part of the persistence architecture per se.
- Minimal persistence approach in which each business object is directly responsible for database access

From a persistence perspective, no matter which of these approaches is used, development of the system and architecture presents two distinct challenges to the development team. The first challenge is to accurately represent the business logic as a collection of business objects that include interfaces for performing the correct set of functionality. The second challenge is to be able to create, retrieve, update and delete (CRUD) records that represent the state of these business objects from the database in an efficient fashion.

Data access and business logic are significantly different tasks both in their goals and development approaches. Consequently, except when a fully automated persistence architecture is used, it is often the case that two separate teams are responsible for the development of business logic and data access. If both teams work directly with the business objects, serious contention may result. Problems encountered in practice include:

- Changes to business logic that impacts the development (e.g., requires recompilation) of database access code even when there is no change to the attributes of business objects. (Note: recompilation can be a problem even if a fully automated persistence framework is used.)
- Changes to the database schema can impact the development of business objects
- The data access team may unduly influence the design of the business objects, leading to a data-centric model and design
- The two teams' development schedules need to be in sync; slippage on one team can adversely impact the other team's progress

Therefore, detach the state of the business object into a separate state class that can be agreed upon and completed prior to the start of significant development by either the data access team or the business object team. This class should be nothing more than the raw data (preferably basic data types) used to represent the state of the object. The business object contains all business logic and a reference/pointer to an instance of the respective state class. This allows the development of business logic and data access to proceed in parallel with the state class serving as a contract between the two teams.

Using this approach, it is important to limit data access focus to providing CRUD operations (i.e., no business logic provided by stored procedures). The purpose of the data access portion of the application is to provide essential access to the data used by the business objets, and deliver this data in the most efficient way possible. Likewise, the purpose of the business object team is to develop business objects that manipulate the state of the object in accordance with the business requirements. Maintaining this separation insures there is no overlap between the development teams.

Benefits

- Development Time. Data access and business logic can be developed in parallel reducing overall development time.
- Separation of concern. Data access remains separate from business logic, improving the understandability of the design.
- Testability. Business objects can be more easily developed and tested based on data access stubs, thereby relieving the business object development teams from dependencies on the data access classes and the database libraries.
- Caching and identity management. Separating the persistent state from the persistent object can be leveraged to aid in managing multiple class instances that represent the same entity (see the Object Identity Cache pattern).
- Object distribution. Separating the persistent state from the persistent object can aid in passing state in a distributed system. In cases where in is necessary to pass an object as an argument to a distributed method, it is more desirable from a performance perspective to pass the object's state as opposed to a remote reference. A new instance can then be created from the state, manipulated locally and then returned to the caller.

The persistent state can be implemented in a variety of ways depending on the requirements of the system. They can roughly be described by the following

SAP - Exhibit 1105
SAP v. Valtrus et al. - IPR2025-00420
Page 270 of 281

Implement the State as a Structure of Values of Basic Data Types

Each business object class has an associated struct. This is the most straightforward approach, although also the least flexible. With this approach, the business logic may need to manipulate lower level data types contained in the state object.

Supporting classes which need to manipulate the state object (in order to retrieve data from the database or pass the value between processes) need to be knowledgeable about the struct data type to manipulate it. In addition, copying the struct may be non-trivial if it contains dynamically allocated memory.

```
struct State_Data
    {
        long      id;
        char      code[8];
        string value;
    };
```

Implement the State as a Class Containing Member Variables of Basic Data Types

1. Implementation using a "developer-friendly" state class: A state class can contain basic data types (except char*), extended basic data types with value semantics (e.g., currency, String), other state classes, and vectors (not arrays) of all of the above. This does not, however, imply that the class is a flexible (dictionary-like) data structure. These state classes could/should all inherit from a common abstract base class which defines (but does not implement, at least in C++) a serialization protocol (Java is a different story than C++ because everything is serializable almost by default).

```
class StateClass
public:
    virtual void read(Stream inStream) = 0;
    virtual void write(Stream outStream) = 0;
};
class StateData: public StateClass
{
public:
    virtual void read(Stream inStream);
            // implementation reads state data from stream.
    virtual void write(Stream outStream);
            // implementation writes state data to stream.
Private:
    long      id;
    char      code[8];
    string value;
};
```

2. Implementation using an enhanced kind of the class described in (A) but which also happens to be a flexible data structure (in the sense that the same class can, similar to a dictionary, support data structures of arbitrary shapes).

This approach provides more flexibility since some common behavior can be abstracted into a base class. In addition, supporting classes which need to manipulate the state object (in order to retrieve data from the database or pass the value between processes) can do so in a polymorphic fashion. Also, copy constructors and destructors can be used to handle dynamically allocated memory.

Implement the State as a Class that Contains Key-Value Attribute Pairs.

This is an alternative approach to the one listed above. Using this technique the state class would contain a keyed data structure (e.g. a dictionary) containing keys (the attribute name) and values (the attribute value). In cases where you want to copy the state or pass it to another process, the supporting code does not need to know the type of the state object it is working with. State objects can simply be asked to read or write their data to and from a stream or string. While this offers a more dynamic solution, it should be noted that with this solution additional logic would need to be included in the persistent object to insure the validity of the associated state class.

In all of these approaches, another important concept is the implementation of associations between objects. In general, the best approach is to store these as Soft References to the other objects as opposed to actual pointers. This illuminates the need to load a large graph of objects when only one is needed, as well as easing the question of whether to implement a deep or shallow copy.

Identity Manager: Manages a collection of persistent state objects for a given class.

Context Manager: Used in conjunction with Identity Manager to maintain separate collections of persistent state objects for separate application contexts.

Lazy Instantiation: Restores persistent state object for a given object instance on-demand.

Object Identity Cache: Caches persistent state objects referenced by persistent objects.

Piecemeal Retrieval

FIG. **174** illustrates a flowchart for a method **17400** for providing a warning upon retrieval of objects that are incomplete. An object is provided with at least one missing attribute in operation **17402**. Upon receipt of a request from an application for the object access to the attributes of the object is allowed by the application in operations **17404** and **17406**. A warning is provided upon an attempt to access the attribute of the object that is missing in operation **17408**.

The warning may include information on how to handle the missing attribute. An implicit transaction may also be called by the object upon the attempt to access the attribute of the object that is missing. In addition, an explicit transaction may be called by the object upon the attempt to access the attribute of the object that is missing. Also, the transaction may also include finding the attribute that is missing.

When legacy transactions don't allow object or entity based retrieval, how do we retrieve useful objects?

Object and component based projects designed and built from the ground up will likely have a well thought out component model and architecture where GUI widgets are linked or bound to domain objects. Data access (and retrieval) for these objects is organized around the business entity, rather than a transaction, and so data is packaged into cross-functional objects, rather than transaction-specific data structures. Each business object manages the retrieval of its data items.

These domain objects provide public methods for accessing, comparing, displaying, and mutating data. Therefore, developers will only access data through these encapsulated, self-requesting domain objects. (See Individual Persistence).

For example a billing application that accepts bill payment over the phone might use the account number, customer name, amount due, date due, and credit card number.

With an Entity-Based Data Access architecture, the account **17500**, customer **17502**, monthly bill **17504**, and bill payment objects **17506** will all be retrieved. FIG. **175** illustrates an Entity-Based Data Access System.

This architecture is preferred if conditions allow, however legacy programs usually retrieve data through transaction or message based systems. These transactions often have two notable characteristics. One, a business unit of work or business transaction often acts on multiple business entities, and two, for each individual entity, the transaction might only retrieve and update a few data attributes.

Using the same bill payment example, a legacy system might utilize a accept bill payment' transaction. This transaction would require only a small portion of the attributes for the account, customer, or monthly bill entities and so the data would be retrieved piecemeal only as required by the transaction.

FIG. **176** illustrates a Retrieving Data Piecemeal System.

In this case, the transaction would allocate a single record (an 'accountPaymentData' **17600** structure as shown in the Figure) to fetch and store the required data items. This structure would then be used to populate the business entities.

As a result, domain objects are left incomplete and therefore unsuitable for use by all services. This forces the developers of services to know and understand the use of transactions to retrieve objects.

Therefore, when legacy circumstances dictate, retrieve data piecemeal, on a transaction basis as necessary rather than as complete business entities and develop mechanisms to handle access and updates to missing attributes of piecemeal objects.

By default, objects should return a warning when a missing attribute is accessed with no other means to retrieve it. This warning would simply let calling applications know that an attribute is missing or unavailable. The calling application would then have to determine how to handle these missing attributes.

A preferred approach to handling missing attributes would be to use multiple overlapping transactions. While, each transaction might only populate a part of the object, the transactions taken together assemble complete objects.

This use of overlapping transactions could either be implicit or explicit to the objects. An implicit transaction would be called by the object when a missing attribute is accessed. This method of lazy retrieval may be preferable because it is transparent to the calling application.

An explicit transaction would be called by a task independent of the object. Note however, that explicit transactions would either require that the task holds the object or that an object identity mechanism is used to find the existing object rather than create a new one.

In some cases, new transactions can be created for the explicit purpose of retrieving full or partial business entities. This approach requires a thorough knowledge of the legacy system.

In other cases, the legacy code can be opened and the transactions modified to bring back additional data. Care should used when doing so as legacy code is often fragile and poorly documented.

Benefits

Legacy Reuse. The overwhelming benefit of piecemeal retrieval is that it enables reuse of legacy systems. Clients generally have a substantial investment in their existing systems and they will be hard pressed to convert them to component based systems built on objects. This pattern allows new systems to reuse existing business logic through their transactions.

Performance and Impedance. Objects based on transactions typically bring back only that data which is needed. The transaction is typically mapped directly to change that the user is trying to make. This improves performance by reducing the number of network calls and only bringing back data that is needed.

Individual Persistence organizes data access around business entities rather than transactions.

Object Streaming handles the conversion of data from the structures received from transactions into business objects.

TRANSACTION SERVICE PATTERNS (1012)

Transactions and LUWS

A transaction is a set of business events that, coupled together, accomplish a particular business function, such as turning on gas service. Because the events are logically related, their data changes are logically related as well. Taken together, these data changes create a new, consistent state for the business model.

While a transaction is in process, the state of the business model may not be consistent, so it is necessary to manage the entire transaction from its point of origin to its point of completion. Whether the transaction is successful or not, the point of completion will always result in a steady, consistent state for the business model. For successful transactions, data changes will be committed and the business model will reflect all new business data associated with the transaction. For failed transactions, data changes will be rolled back and the business model will appear as it did prior to the start of the transaction.

To help manage the transaction from point of origin to point of completion, each transaction is organized through a single Logical Unit of Work (LUW). This LUW manages the business model and any of its subsequent data changes. While both users and internal exceptions can determine the success of a transaction, the LUW handles the commit and rollback operations.

FIG. **177** illustrates a Commit and Rollback routine **17700**.

As transactions become more complex and require a greater scale of changes to the business model, the LUW trying to manage these changes becomes large and unwieldy. To simplify these transactions, the LUW is broken down into nested, more granular, logically related units of work called Secondary LUWs. Secondary LUWs are identical to LUWs except that their commit and rollback operations affect only the business model of their parent LUW and are not persisted to a data store. Consequently, a secondary LUW must manage its data changes differently than its parent.

FIG. **178** illustrates Nested Logical Units of Work.

One method for managing changes to the business model involves copying the model into the secondary LUW. Another often simpler approach is to store both old and new (or potential) values for all objects in the business model.

Transaction Patterns

In the process of managing its business model, a LUW will often have to send messages to all business objects within the LUW. Examples of such messages include saveDataChanges, retrieve, or isDirty. Rather than hardcoding a call to each object in the model, the pattern LUW Context suggests using a bag (or collection) to hold all objects referenced by the LUW. Then, a single message can be sent to the bag which will forward it to all objects within it.

Support for user multi-tasking can also present problems for LUWs. Through multi-tasking, multiple LUWs will be running concurrently. Problems occur if the business models of these concurrent LUWs overlap and the transactions attempt to write to the same business object. A call center representative trying to solve two customer problems during the course of one call is one example of this scenario. The Separate Models pattern helps solve this issue by assigning each LUW independent copies of their portion of the business model. This keeps one transaction from interfering with another.

There are also several patterns that address problems related to sending transactions across the network. These patterns typically focus on minimizing network messaging.

When an LUW is called to commit, the transaction will assemble the necessary objects from the business model to send their data changes to the information services layer. This group of objects will include all new and dirty objects as well as any objects marked for deletion. For each business object, the transaction will likely have a corresponding request. If each of these requests were then sent to information services independently, a large number of network messages would result. To solve this problem, the Request Batcher pattern batches all requests associated with a transaction together into one network message. On the other end of the network, Information services would unbatch the transaction requests and persist the data changes.

Another problem that may arise when multiple requests are sent for a given transaction is deadlock. Deadlock occurs when two requests are trying to lock the same pair of objects. Each request locks one object and waits to commit until it can lock the other. Therefore, each request will wait for the other to complete while neither is able to do so. The Request Sorter pattern works with Request Batcher to handle this problem by sorting requests as they are being unbatched by Information Services. A request is not allowed to proceed until any dependent requests are completed.

During retrieval, one request may depend on the response data for another request. For example, a business transaction that tries to retrieve a customer when given a customer ID will probably also want to retrieve the customer's address. However, the transaction won't have the address ID until the customer is retrieve. Thus multiple network messages are required when one request is dependent on another. The Dependent Request pattern solves this problem by allowing a batched request to indicate that it depends on another request.

As these patterns demonstrate, there is a high degree of correlation between Transaction Services and Information Services. Many of these transaction patterns require an understanding of Information Services patterns. Two such examples are Individual Persistence and Piecemeal Retrieval. It is recommended that these patterns be read and understood prior to using Request Batcher, Request Sorter and Dependent Request.

Dependent Request

FIG. **179** illustrates a flowchart for a method **17900** for allowing a batched request to indicate that it depends on the response to another request. A group of business objects necessary for a transaction are provided in operation **17902**. Logically-related requests received from the business objects are batched into a single network message in operation **17904**. One of the requests is a parent request. Received from the parent request in operation **17906** is a register that at least one of the requests is dependent upon the response data. The network message is sent across a network and the requests are unbundled from the network message in operations **17908** and **17910**. Upon receipt of a response to the parent request in operation **17912**, data is directed from the response to the parent request to the dependent request in operation **17914**. Received from the response to the parent request is a response to the dependent request based on the received data in operation **17916**.

The dependent request may not have a primary key. In such situation, the response to the parent request may include the primary key for allowing the dependent request to be responded to. As another option, the dependent request may also include a pointer indicating that the dependent request is dependent on the parent request. In this situation, the pointer may be passed to the parent request during the step of batching the requests into the network message.

Additionally, the pointer may be a configurable field of the dependent request. The requests may also be reused independently of each other. In event another aspect of the present invention, the dependent request may wait for the parent response at the server for minimizing network traffic.

During retrieval, one request may depend on the response data for another request. Nevertheless, ensure that a single network message contains both requests, while using Request Batcher.

A business transaction typically acts on multiple business entities. Consider an account maintenance window, which edits information from account, customer, credit profile, and home and work addresses. Given a unique account identifier, the business transaction can retrieve all five objects.

Once the account retrieves all of its data, it will know its unique customer identifier. The customer can then retrieve its own data, which includes the identifiers for the credit profile and both addresses. Finally, the profile and addresses can retrieve their data. In this case, profile and address retrieval depends on customer data; customer retrieval in turn depends on account data.

However, Individual Persistence requires that each object have its own, independent request module. That is, customers do not always need accounts to be retrieved. After a customer's unique identifier has been filled in—regardless of by whom—it retrieves its data independently.

In theory, this independence is not an issue. The account could first get its data. After the customer's identifier was filled, the customer could send its own request.

In practice, however, sending multiple messages, in series like this, degrades network performance. Request Batcher **18000** provides a solution which bundles up requests into one network message, thereby minimizing traffic. FIG. **180** illustrates a Batching Retrievals and Dependency.

This batching framework applies not only to update messages. For retrieval as well, one overall, batch request receives one batch response. Yet an individual batched request may depend on the response data from another batched request. The serial nature of the two requests must be preserved, even while the requests actually go in the same batch, in parallel.

Therefore, additional mechanisms should allow a batched request to indicate that it depends on another request. If a business object does not have its primary key—or other attributes guaranteeing a unique match—becomes a Dependent Request. Because a dependent cannot fetch itself, some other business object will inevitably have the necessary foreign key. The dependent request will therefore register its dependency on this other object.

This object, the parent request, can have multiple dependents. The aforementioned customer had credit profile and

address dependencies. This implies the need for an arbitrarily large collection of dependent requests. A dependency collection can even include requests for multiple instances of the same class. This was the case with two address requests depending on the same customer.

FIG. **181** illustrates the Dynamically Setting Dependency.

Dependencies should not be hard-coded. Any business object can register as dependent on any other object which can provide the necessary data.

In this manner, dependencies can be set at run-time. This could happen while building the model, or as requests register with Request Batcher.

Benefits

Performance. This solution supports request batching for retrieving interdependent models.

Reuse. Because dependencies are not hard-coded, business objects can be reused independently of each other.

Loose Coupling. When a request dynamically registers its dependency, it need not know anything about its parent request. The dependent effectively says, "I don't know who you are, but I know that your response data contains my identifier."

Dependent Request would be irrelevant were it not for the "transaction impedance mismatch." This mismatch means that transactions no longer map one-to-one to access modules. Individual Persistence is the approach which dictates that each business entity have its own independent access module.

Request Batcher solves the performance problems of sending multiple, small-grained request messages over a network. But the resultant single message must support dependencies, with Dependent Request or a similar mechanism.

LUW Context

FIG. **182** illustrates a flowchart for a method **18200** for sending a single message to all objects in a logical unit of work. A group of business objects necessary for a transaction are provided and managed in a logical unit of work in operations **18202** and **18204**. A receiver is created which communicates with the business objects in the logical unit of work in operation **18206**. Upon receiving a message for the business objects in the logical unit of work in operation **18208**, the message is directed to the receiver in operation **18210**. The receiver also forwards the message to each of the business objects in the logical unit of work.

Several groups of business objects necessary for a transaction may also be provided with each group of business objects being managed in a separate logical unit of work. Also, a separate receiver may communicate with each group of objects. As another option, a request batcher in communication with the receiver may also be provided for batching requests from the business objects for delivery. In such an embodiment, the request batcher intercepts the requests from the business objects and holds the requests until told to deliver the requests by an activity associated with the logical unit of work.

Optionally, the receiver may hide technical details including details of persistence and garbage collection from business developers. As a further option, the business objects may be distributed across a network. Also, the receiver may distribute the message to each of the business objects across the network. Additionally, the logical unit of work may optionally be modeled as an object in software.

Applications often need to send technical messages, like saveDataChanges( ) or release( ), to all business objects in an LUW. Do this in a consistent manner and hide technical details from business developers.

Consider an Account Payment window, which displays information about an Account, Customer, Monthly Bill, and Payment. This window occasionally needs to send generic, technical messages to all business objects within its LUW. This messaging has nothing to do with the window's application-specific behavior. In fact, the other windows in the system need to send the same, generic messages to their LUW business objects. Although the business objects receiving these messages differ from window to window, the messages remain the same.

In addition, this messaging typically has an arbitrary order. All that matters is that all business objects eventually receive the same message.

For example, the window might use Individual Persistence. Then, when the user decides to save the window, all business objects receive a message like saveDataChanges( ). The resultant pseudo-code would look like:

```
AccountPaymentActivity::saveDataChanges()
{
    // Propagate along the save message to all business
    // objects in my LUW.
    this.getAccount().saveDataChanges();
    this.getCustomer().saveDataChanges();
    this.getMonthlyBill().saveDataChanges();
    this.getBillPayment().saveDataChanges();
}
```

A retrieveData( ) message might also be required, if the object model pre-instantiates objects before retrieving them. Similarly, refresh( ) could be used: the business object, if dirty, replaces any changes with data originally from the data store.

Even without Individual Persistence, there are other common messages the window may want to send. For example, distributed objects typically need to be told when their memory can be reclaimed. COM+ uses the well-known method releaseRef( ), whereas some implementations of CORBA use release( ). Regardless, this is a common message that would also need to be sent to the business objects, similar to the saveDataChanges( ) propagation above.

Dirty Flag provides another example. Here, the window accumulates the results of dirty checking, as follows:

```
AccountPaymentActivity::isDirty () {
    // Return true if any single business object in the LUW is dirty,
    return
        (this.getAccount().isDirty() or
        this.getCustomer().isDirty() or
        this.getMonthlyBill().isDirty() or
        this.getBillPayment().isDirty());
}
```

This hard-coded approach, although straightforward, is both tedious and error-prone. It is tedious because business developers shouldn't have to deal with technical issues like dirty checking or distributed garbage collection. They should focus instead on business-specific processing.

Moreover, this is error-prone because it can be difficult to detect if the developer makes a mistake. For example, a new requirement could make the window display address information. In addition to re-painting the window, the developer would also need to modify their hand-coded methods. But

the developer might forget to update the isDirty( ) or release( ) methods. Such errors can be difficult to locate. (Readers who have debugged memory leaks will certainly agree.)

Instead, an architecture mechanism should encapsulate the propagation of these technical messages. When a message needs to be forwarded generically, to all objects in an LUW, the architecture should handle it. Such a capability would free business developers from worrying about these technical details. FIG. **183** illustrates a Hand-crafted Message Forwarding scheme.

Therefore, an architecture "bag" will represent the business objects in a particular LUW. This bag, or collection, will hold onto each business object. Then, when the bag receives a message like saveDataChanges( ) or release( ), it simply forwards the message to each member business object.

FIG. **184** illustrates a Generic Message Forwarding feature.

Each LUW must have its own bag. This enforces the Isolation property (of ACID) for LUWs. That is, one LUW should not affect another LUW. For example, if the Account Payment and Account Services activities have separate LUWs, they will correspondingly have their own bags. Then, calling saveDataChanges( ) on the Account Payment activity will forward saveDataChanges( ) to only those business objects owned by Account Payment.

The bag also helps ensure the Atomicity property (also of ACID) for LUWs. It provides a single, atomic interface into the multiple business objects of the LUW. By design, it ensures that all business objects receive the same architecture messages.

Thus, the scope of a bag is an LUW. In addition, a bag provides contextual information for the LUW—i.e., which business objects that LUW uses. The architecture bag therefore models the LUW Context, and will be named as such.

Benefits

Encapsulation. LUW Context hides technical details of persistence, garbage collection, etc., from business developers. Some of the Known Uses have managed to hide this framework, in entirety, from business logic.

Robustness. This approach guarantees that each business object in the LUW receives forwarded messages. There is no longer the chance of a developer forgetting to include a particular business object in a group message.

Application Maintainability. As the application requirements change, the set of business objects in an LUW can change without impacting the generic, LUW code. For example, a future version of the Account Payment window could also display Address information. This introduces a new business object into the LUW. Yet it would not require updating saveDataChanges( ), or release( ) methods, as it would have previously.

Performance. LUW Context can dramatically improve performance in a distributed environment. By nature, it batches up messages for a group. This can reduce network messaging.

For example, consider a search window which has instantiated 30 business objects. Releasing those objects, if the messages were sent independently, would require 30 network messages. However, with LUW Context, a single message can go from the client to the server. Then, within the server executable, the LUW Context forwards release( ) to the 30 member objects. This is far less costly than using the network for that messaging. Because of this message batching, some readers may confuse LUW Context with Request Batcher. It is true that both reduce the number of network messages. However, the former is concerned with supporting a family of generic, architecture messages, like isDirty( ) and refresh( ), on a single atomic object. The latter is concerned with grouping database requests into a physical package, for un-batching at the server. Although both have similar principles and characteristics, they solve different problems and are implemented differently.

Architecture Extensibility. LUW Context models the LUW as an actual object in the software. Any other architecture processing which executes on a per-LUW basis can also be coded there. (See the Related Patterns section for examples.)

This pattern seeks to hide the message propagation from business logic. In fact, messaging to an LUW Context can be hidden completely in an architecture superclass. Previously, saveDataChanges( ) would've been coded specifically in each concrete activity class. LUW Context allows it to be abstracted, as in:

```
AbstractLUWActivity::saveDataChanges() {
    // Propagate along the save message to all business
    // objects in my LUW. Subclasses don't even need to
    // know about this method.
        this.getLUWContext().saveDataChanges();
}
```

This assumes that business objects were put into the LUW Context in the first place. The context object can be passed in when instantiating an object, transparently by the persistence and streaming frameworks, etc.

An LUW Context can collaborate with a Request Batcher, if requests are batched for transmission to the data store. Rather than storing the batcher globally, each context and hence model can have its own manager. This allows multiple domain models, in multiple contexts, to send transactions simultaneously but independently. Then, whenever a business object requests an access or update, its request will be intercepted by the model's particular Request Batcher. The batcher then holds these requests until the activity—which owns the LUW—tells the batcher to send them.

The LUW Context holds onto all domain objects in a particular model. It can therefore collaborate with an Identity Registration Manager, to enforce object uniqueness within the particular context.

The Potential Variables pattern, which provides local undo, is discussed in the first version of the Object Solutions Handbook. If local LUWs use this approach, then LUW Context is a natural location to store the LUW phase variable.

In that pattern, every time a business object sets an attribute, the variable must be checked. The LUW Context, as intermediary, can provide a simple public interface which supports setting and querying the phase variable.

Request Batcher

FIG. **185** illustrates a flowchart for a method **18500** for batching logical requests for reducing network traffic. A group of business objects necessary for a transaction are provided and managed in a logical unit of work in operations **18502** and **18504**. In operations **18506** and **18508**, logically-related requests received from the logical unit of work are grouped into a single network message which is then stored. The message is sent in operation **18510** upon receiving an order to send the message.

SAP - Exhibit 1105
SAP v. Valtrus et al. - IPR2025-00420
Page 275 of 281

Optionally, update and retrieval transactions may be grouped into a single network message which is stored. The message may be sent upon receiving an order to send the message. As another option, the requests from the message may be unpackaged at a point across a network and data changes may be persisted. In a further optional embodiment, responses to the requests may be received and the responses may be bundled into a reply. In one embodiment, the requests in the message may be sorted. In such an embodiment, the requests in the messages may also be separated into submessages.

When domain objects request themselves, minimize the impact of network traffic.

Individual Persistence assigns responsibility for data access to individual business objects. Then, each business object can retrieve, update, insert, and delete its data from a persistent store independently of other objects. This promotes encapsulation and reuse across business transactions.

FIG. **186** illustrates the manner in which the present invention sends requests **18600** independently.

Thus, an LUW which uses multiple business objects will correspondingly have multiple requests. This might suggest that each independent request communicate independently with data access services. Then, each logical request would translate into its own physical, network message.

Every network message imposes a certain amount of overhead, independently of its contentsor length. This implies that multiple, small-grained messages have more overhead than a single, large-grained message. Many networked-constrained environments cannot tolerate this additional overhead. Such environments should minimize the impact of this network traffic.

Therefore, a high-performance transaction should batch its logical requests into a single network message. Moreover, a framework should handle this packaging, transparently to application logic.

FIG. **187** illsutrates a manner in which the present invention registers requests.

A Request Batcher **18700** object will group logically-related requests. All requests will register with this coordinating object, rather than sending themselves immediately and independently to their server or database. The batcher will then store these requests together, until told to send them as a unit. This batching applies equally well to update and retrieval transactions.

A corresponding Request "Unbatcher" **18702** on the server will unpackage the bundled network message. Finally, this Unbatcher will bundle the network response and send it back to the Request Batcher.

Benefits

Performance. Sending a single message of multiple requests, as opposed to multiple messages of single requests, improves communication performance.

Dynamic. Batching and sorting requests is transparent to the requests themselves. Requests do not know that a particular transaction contains them. This dynamic relationship allows any type of request to be part of any transaction at run-time.

Scaleability. In an asynchronous or multi-threaded environment, an application could use multiple Request Batchers. For example, each LUW could have its own batcher. A batcher needs to store state while building the batch, as requests register. Using multiple instances facilitates registration for, and sending of, multiple batches simultaneously. Users can then multi-task while other, time-consuming requests process in the background.

This simultaneity can also be supported with one, multi-threaded batcher. In this case, each request registers along with its unique transaction id.

Centralization. The batcher has visibility over all requests in the LUW. This provides a centralized point to sort these requests, thereby supporting referential integrity and deadlock avoidance.

Request Sorter

FIG. **188** illustrates a flowchart for a method **18800** for sorting requests that are being unbatched from a batched message. A group of business objects necessary for a transaction are provided in operation **18802**. Logically-related requests received from the business objects are grouped in operation **18804**. Sorting rules and/or sort weights are obtained in operation **18806** and, in operation **18808**, the requests in the essage are sorted and placed in a specific order determined from the sorting rules and/or the sort weights. The sorted requests are batched into a single message which is sent to a data server where the requests are unbundled from the message in the specific order (see operations **18810**, **18812**, and **18814**).

A request may also not be allowed to proceed until all dependent requests are completed. A plurality of transactions may each use the same sorting rules for preventing deadlocks. Optionally, the class represented by each request may be determined so that the sorting rules may be based on a class type. As another option, the sorting rules may include referential integrity rules which ensure that references between two relational tables are valid. In such a situation, a linear ordering of requests may also be created based on the referential integrity rules. The numbering of the position of the request in the linear ordering may also be the weight of that request so that requests with lower weights are processed before requests with higher weights.

In an update transaction, order requests for referential integrity and deadlock avoidance.

Referential Integrity

Referential Integrity (RI) ensures that references between two relational tables are valid. That is, foreign keys in one table must refer to existing primary keys in another table. For example, RI rules could require that all accounts have a customer. Then, values in account.cust_id would need matching values in customer.cust_id.

Mission-critical RDBMSs can enforce RI at run-time. Then, if a modified foreign key does not match an existing primary key, the database prevents the update.

Continuing the example, a transaction may insert a new customer and its new account. If the transaction inserts the account first, account.cust_id will refer to a non-existent customer. The RDBMS will raise an error, thereby failing the transaction. Instead, the account request should run after the customer request.

Deadlock Avoidance

Even without RI, request ordering remains an issue.

Imagine a transaction A orders customer before account. Conversely, a concurrent transaction B orders account before customer. A will request a lock on the customer table, while B will request a lock on the account table. A must wait for B to complete and release its account lock. Yet B cannot complete until A releases its customer lock.

Thus, both transactions will deadlock. Many transaction-processing systems would simply fail both transactions, after a time-out. Yet both transactions may otherwise have been valid.

SAP - Exhibit 1105
SAP v. Valtrus et al. - IPR2025-00420
Page 276 of 281

Traditional Approach

Traditionally, transactions have hard-coded deadlock avoidance and RI. Each transaction has called its own update module. Each hand-crafted module has ordered multiple SQL statements, according to these rules.

However, with Individual Persistence, a transaction no longer maps to a centralized module. Instead, independent requests register for the transaction in an ad-hoc manner. FIG. **189** illustrates an Ad Hoc Registration feature.

Moreover, an account has no hard-coded "knowledge" that it should persist after a customer. This independence provides flexibility any business object can request an update without concern for other business objects. A framework which constrains the request order must support this flexibility.

Therefore, an update transaction should sort its requests before sending them to the data server. The sorted result will conform to RI rules. Then, across update transactions, all customer requests can appear before all account requests. In addition, every transaction will use the same sort algorithm. That will prevent deadlocks.

Multiple requests, can no longer send themselves directly to the server, in an ad hoc fashion. Instead, they must register with a centralized object, which can sort them first. A centralized Request Sorter will order multiple requests before finally sending the transaction.

FIG. **190** illsutrates a manner in which the present invention sorts requests by weight.

The sorter **19000** will have visibility to sorting rules, or even weights, to determine this order. The rules can typically be based on the class type. Before sending the transaction, the sorter can ask each request which class it represents. In this manner, the sorter can re-order the requests appropriately.

Benefits

Separation of Concern. This sorting pattern hides the technical details and complexity of RI from business logic. Applications avoid hard-wiring customized RI rules for its transactions.

Maintainability. RI rules can easily be changed without impacting application code. Granted, this does not happen frequently in production.

Reusability. The generic Request Sorter uses universal sorting rules, or weights. These rules are global across business processes. Moreover, the rules are based on existing, reusable business objects. Therefore, new applications can reuse the sorter, as well.

Visibility. If RI enforcement is distributed across application logic, it can be difficult to get a complete picture of the referential rules. Request Sorter centralises those rules (i.e. weights) in one, visible place.

A complete, linear ordering of all domain classes can be created, based on the RI rules. Each class will have a unique position in the ordering. This position is the class' weight for the sorting algorithm. Requests for domain objects with lower weights will always appear before requests with higher weights.

For example, consider the ordering:

27. . . .

28. Customer

29. MeterRead

30. Account

31. Meter

32. MonthlyBill

33. . . .

This satisfies the RI rule mentioned earlier, because Customers have a lower weight (**28**) than Accounts (**30**). Thus, requests for customers will appear in any transaction before requests for accounts. As long as the order satisfies every RI rule, the request sorter can use such a linear ordering.

This sort ordering can be created programmatically. A sort generator can convert pairwise relationships into linearly-ordered weights. Then, the Request Sorter could use an algorithm like QuickSort to do the actual sorting. (Alternatively, object requests could be sorted as they are registered, a la Insertion Sort.)

A centralized store-and-forward site must hold requests, before they send themselves to the server. Otherwise they cannot be sorted as a group. Request Batcher provides a centralized place to attach a sorter.

Separate Models

FIG. **191** illustrates a flowchart for a method **19100** for assigning independent copies of business data to concurrent logical units of work for helping prevent the logical units of work from interfering with each other. In operation **19102**, multiple logical units of work operating concurrently are provided. Each of the logical units of work manipulate at least one common business object. In operation **19104**, a copy of the common business object is created for each of the logical units of work such that the copy of the business object for one logical unit of work becomes a separate instance from the copy of the business object for another logical unit of work. Each copy of the business object knows the context of that copy of the business object in relation to the associated logical unit of work. Upon receiving a request to make changes to a copy of the business object of one of the logical units of work in operation **19106**, that particular copy of the business object is changed while the other copies of the business object are not changed. It is then verified in operation **19108** that only one copy of the business object has been changed and the common business object is updated in operation **19110** based on the change to the copy of the business object.

A business object may optionally be passed as a parameter from one context to another. In such case, a context copy of the business object may be created which includes a duplicate of the original data and excludes a context variable. As another option, an exception may be thrown when an attempt is made to create a copy of a business object being altered by one logical unit of work for another logical unit of work.

The business object may also be sent to another context as at least one of a single focus of a window that is being created and a parameter in an explicit parameter-passing mechanism. Additionally, the copies of the business objects may be created from a same retrieved data stream. As a further option, receiving a request to make changes to a copy of the business object of one of the logical units of work and changing that copy of the business object may further include the broadcasting of the change to the other logical units of work.

Support multiple business LUWs within an MVC-based architecture. Manage these LUWs concurrently yet separately, thereby preserving the Isolation property of ACID.

Multi-tasking allows the user to complete several different business functions independently of each other. Those functions which are business LUWs must process concurrently yet separately. For example, a user could establish a new

customer account while separately verifying bill details for another customer.

Providing for these multiple LUWs demands mechanisms which ensure integrity. Specifically, as with any LUW, a primary LUW must isolate its own changes. It is an independent workspace which prevents its changes from affecting other LUW.

However, an MVC-based OO architecture does not naturally support this requirement. With MVC, the domain model stores all data changes. Windows are merely a view into this model, and they have little business data of their own. In addition, MVC model objects have no idea which views are using them. Instead, the model anonymously broadcasts its data changes, and all views on the model respond by updating themselves. This synchronizes windows with their business data. Thus, MVC allows multiple views to simultaneously display, and be refreshed by, a single copy of the model data.

FIG. **192** illustrates the MVC Implementation with Global Model.

Unfortunately, this benefit of MVC introduces a problem. A globally-shared domain model does not naturally separate concurrent LUWs. It puts a burden on business "activity" objects, which coordinate the high-level business processing across their domain models. Each activity has to either avoid overlap or know specifically how it affects the model.

Consider a telecommunications system, with two separate business LUWs for paying bills and adding new services, like call waiting. An end user might launch windows for these two LUWs simultaneously. This would allow the user to multi-task while conversing with the customer.

Both windows display Account and Customer information. In addition, the Account Services window actually modifies the Account object, whereas the Account Payment window does not. Making a payment only modifies the Bill Payment object. Both windows, using MVC, could share the same Account **101** instance.

It is not atypical for custom architectures to have generic mechanism for persistence and transactions. For example, the architecture could use a straightforward mechanism which automatically saves all business objects within an LUW. Then, when the user saves the Account Payment window, the changes to Account **101** would be accidentally saved as well. The user would then not be able to later cancel changes on the Account Services window. This violates the isolation of the two LUWs.

A similar problem might arise with a garbage collection framework, which explicitly destroys all instances once the LUW has completed. In this case, Account Payment would need to ensure it did not explicitly free the memory for Account, while Account Services was still using it.

Therefore, using a global, MVC model may preclude using other architecture mechanisms. To avoid the problems of overlapping saves or releasing memory prematurely, the windows could have additional code to ensure the LUWs remain separate. However, adding application-specific code in this manner, to handle a global technical requirement, is undesirable.

Instead, business LUWs should be able to modify domain data independently of each other, transparently in the architecture. In addition, each data change should unambiguously belong to a single, originating LUW.

Modern Object Transaction Monitors promise to provide this capability. These products will handle locking, tracking which LUW has made changes to which piece of data, etc. However, in the absence of an OTM, a custom architecture needs a different approach.

Therefore, separate business LUWs by giving each LUW separate copies of business data.

Rather than using a globally-shared model, each business LUW will own a private, scratchpad copy of its domain model. This satisfies the independence requirement. A business object in one model will automatically be a separate instance from a business object in another model, even if they share the same functional identity. For example, simultaneously opened payment and services windows would have separate copies of Account **101**.

Then, changes made to a particular instance will only be reflected in the LUW which created and points to that instance. This contrasts with a single, globally-shared model. The latter would simultaneously reflect changes across multiple LUWs.

FIG. **193** illustrates the Separate Models for Separate Business LUWs **19300**,**19302**.

The aforementioned telecommunications example had two separate business LUWs for the account payment and account services functions. Although both activities may be related by the same logical account, this pattern gives each a different context copy. Then, when the customer representative cancels the addition of call waiting, she can still save the payment details.

FIG. **194** illustrates the Canceling of one LUW **19400** Independently of Another LUW **19402**.

Thus, using Separate Models preserves the integrity of business LUWs. It allows each LUW to easily save or cancel independently.

This pattern is not intended to allow different LUWs to simultaneously change their different physical copies of the same logical entity. In fact, if both windows modified their Account **101** copy, one of the LUWs would fail. (Mechanisms like optimistic locking would detect the data integrity conflict.)

Precisely for this reason, a good UI design doesn't typically allow simultaneous but separate LUWs to update the same data. And this was not an issue in the example above. Updates to the Account object occur on the Account Services window but not the Account Payment window.

Benefits

- Isolation. Most fundamentally, this pattern solves the Isolation requirement of ACID. It ensures that each LUW has its own "working storage" copies of business data.
- Transparency. Separating models can be done in an architected fashion, as outlined in the implementation section. The separation of LUWs—which is a technical issue—can be hidden completely from business logic.
- Imagine instead that LUWs didn 't have their own copies. Then, each operation might need an additional argument: the LUW owning the data change. This would pollute application code with an extra "transaction ID" argument, as in setBalance(newBalance, transactionId). As previously mentioned, this is only required in the absence of an Object Transaction Monitor. An OTM can transparently manage the transaction Id with the thread, without including it as an explicit argument.
- Uniformity. Application developers don't need to know about which objects may or may not be used by other, concurrent LUWs.

The following implementation assumes that the LUW Context pattern is used to help separate the LUWs.

Each instance of a business object knows which LUW owns it. That is, each instance knows its context. By definition, context gives something a scope, a frame of reference, a relationship to other things. To provide this relationship, an actual LUW Context object will hold onto business objects which share a business LUW.

In addition, each business object can point to its context. In that manner, business objects know their LUW. This could be useful, for example, while building a domain model. Then, the parent object could propagate its context to a linked, child object.

Business objects owned by the same business LUW share the same LUW Context, whereas different LUWs have different contexts. Each context therefore contains its own "working-storage" copy of the model. This delineates an individual workspace, or scratchpad, for each LUW.

At a higher level, each activity object which represents a business LUW has its own context object. That context remains with the activity throughout its entire lifecycle. For initialization, creating a new LUW activity also creates a new context instance for that activity. This context will then be passed downwards, to all business objects, as part of navigation.

Eventually, when the activity closes, it releases its LUW Context. This correspondingly releases all business objects. They can then be garbage collected, because the only LUW using those objects just closed. A context's lifecycle therefore corresponds directly to its activity's lifecycle.

Preserving Context Boundaries

Every context has a scope which limits the business LUW. This context boundary cannot be violated with objects from other LUW Contexts. For, the same instance of a business object cannot live in two different contexts. Otherwise, changes made to that instance would affect two different LUWs.

It is often necessary, however, to pass a business object as a parameter from one context to another. For example, a user may open up a customer details window based on a selection from a search window. The selected customer becomes the focus of the new.window, but it was instantiated in the search context. It is the responsibility of the details window to take the passed-in customer and make a context copy of it. A context copy duplicates the original's data, excluding the context variable, which is re-set to the new context. The copied customer can then be safely used and modified within the details context.

A business object can be passed as parameter to another context as:

the single focus of a window that is being created

a parameter in an explicit parameter-passing mechanism

For example, when a business object is the focus of a new activity, the launching activity could instantiate the new activity as follows:

```
MeterMaintenanceActivity::prorateMeterRead( MeterRead aMeterRead)
{
// Creates a new activity to prorate <aMeterRead>. This will manually
adjust the
// read charges, based on corrections from the location, the office, etc.
// Pseudo-code below.
// Create the new activity instance by reflection, based on the class name,
// and give it a new context and <aMeterRead> as focus.
newProrateActivity = this.newActivity(
                this.prorateMeterReadClassName(),
                aMeterRead );
// Other initialisation here . . .
newProrateActivity.startup();
}
```

The newActivity( ) architecture method instantiates a new activity, instantiates a new context, and creates a context copy of aMeterRead that the new activity can use.

Sometimes an activity cannot get enough information simply by navigating from the focus. Non-focus information that must be passed as an additional parameter could be handled in the following manner:

```
MeterMaintenanceActivity::prorateMeterReadWithCorrection(
    MeterRead originalMeterRead, MeterRead correctedMeterRead)
{
    // Creates a new activity to prorate <originalMeterRead> based on
measurements
    // in <correctedMeterRead>. Pseudo-code below. (Duplicates some
code above
    // for clarification.)
    // Create the new activity instance by reflection, based on the class
name, and
    // give it a new context and <originalMeterRead> as focus.
    newProrateActivity = this.newActivity(
                this.prorateMeterReadClassName(),
                originalMeterRead);
    // Pass along the corrected read, as well. This will create a context
copy and
    // then use reflection to call the right public setter on the activity.
    newProrateActivity.receive( aCorrectedMeterRead,
"setCorrectedRead");
    // Other initialisation here . . .
    newProrateActivity.startup();
}
```

Here, the receive( ) framework method allows any business object to be passed across the context boundary. The receiving activity will automatically create a context copy and then call the specified setter method, with the copy as argument. The setter is application-specific, and it allows the activity to handle and store the context copy wherever it wants.

FIG. **195** illustrates the Context Copying Protects Context Boundaries.

A dirty object should not be safely copied into a new LUW context. Otherwise, the second LUW would begin using information that was half-completed in the first LUW. Again, this violates the isolation requirement. The second LUW could save its changes before the first LUW. This means the first LUW couldn't undo any changes it had made to the dirty object. Instead, to avoid this problem, an exception should be thrown when trying to copy dirty objects across contexts. This disallows users from beginning a new LUW based on half-entered data.

Thus, context copying allows LUW contexts to share parameter information while preserving context boundaries.

Persistence Caching

Although LUW contexts manipulate separate copies of business objects, they can often share the same retrieved data stream. For example, when a workstation retrieves data for Customer ABCD, the returned stream can be stored in a global cache. If another context wants to later instantiate its own copy of Customer ABCD, it can reuse the details stored in the stream cache. This improves performance, by avoiding a redundant request to the remote data store.

Context "Refresh"

Each LUW, while working on its data, is independent of the other LUWs. From that perspective, each LUW context manipulates data that, to its knowledge, is the most current information from the data store. One instance's changes remain invisible to another copy of the same business entity, during the course of normal processing.

However, when an LUW context successfully commits changes, it will have more current data than other contexts

which it intersects. This up-to-date data can be broadcast and shared with the other contexts. These contexts can then decide to transparently incorporate the changes or not.

This refresh mechanism can be complex to build, and it requires an understanding of locking issues. For example, does the window have any changed data which might conflict with the new data? This would make the changes which hadn't yet been committed invalid, and the user would need to be notified.

Although only a few embodiments of the present invention have been described in detail herein, it should be understood that the present invention may be embodied in many other specific forms without departing from the spirit or scope of the invention. Therefore, the present examples and embodiments are to be considered as illustrative and not restrictive, and the invention is not to be limited to the details given herein, but may be modified within the scope of the appended claims.

What is claimed is:

1. A method for efficiently retrieving data comprising the steps of:

(a) determining a total amount and type of data that may be used during an application executed by a client, wherein the total amount and type of data is determined in response to a first client request, and wherein the total amount and type of data that may be used during execution of the application is a superset of the data required to satisfy the first client request;

(b) requesting the total amount and type of data from a server over a network in a single call;

(c) bundling the requested data into a data structure by the server in response to the single call;

(d) sending the bundled data structure to the client over the network;

(e) caching the data of the data structure on the client; and

(f) using the cached data of the data structure during execution of the application on the client in response to a second client request.

2. A method as recited in claim **1**, wherein the data structure is bundled on the server by a business object.

3. A method as recited in claim **1**, wherein the network is at least one of a local area network and a wide area network.

4. A method as recited in claim **1**, wherein the request is administered by a proxy component.

5. A method as recited in claim **1**, wherein the data structure contains no logic.

6. A method as recited in claim **2**, wherein the business object is instantiated by an action of the client.

7. A computer program containing code segments, which when executed on a computer perform efficient data retrieval, the computer program being embodied on a computer readable medium, the computer program comprising:

(a) a code segment that determines a total amount and type of data that may be used during an application executed by a client, wherein the total amount and type of data is determined in response to a first client request, and wherein the total amount and type of data that may be used during execution of the application is a superset of the data required to satisfy the first client request;

(b) a code segment that requests the total amount and type of data from a server over a network in a single call;

(c) a code segment that bundles the requested the data into a data structure by the server in response to the single call;

(d) a code segment that sends the bundled data structure to the client over the network;

(e) a code segment that caches the data of the data structure on the client; and

(f) a code segment that uses the cached data of the data structure during execution of the application on the client in response to a second client request.

8. A computer program as recited in claim **7**, wherein the data structure is bundled on the server by a business object.

9. A computer program as recited in claim **7**, wherein the network is at least one of a local area network and a wide area network.

10. A computer program as recited in claim **7**, wherein the request is administered by a proxy component.

11. A computer program as recited in claim **7**, wherein the data structure contains no logic.

12. A computer program as recited in claim **8**, wherein the business object is instantiated by an action of the client.

13. A system for efficiently retrieving data comprising:

(a) logic that determines a total amount and type of data that may be used during an application executed by a client, wherein the total amount and type of data is determined in response to a first client request, and wherein the total amount and type of data that may be used during execution of the application is a superset of the data required to satisfy the first client request;

(b) logic that requests the total amount and type of data from a server over a network in a single call;

(c) logic that bundles the requested the data into a data structure by the server in response to the single call;

(d) logic that sends the bundled data structure to the client over the network;

(e) logic that caches the data of the data structure on the client; and

(f) logic that uses the cached data of the data structure during execution of the application on the client in response to a second client request.

14. A system as recited in claim **13**, wherein the data structure is bundled on the server by a business object.

15. A system as recited in claim **13**, wherein the network is at least one of a local area network and a wide area network.

16. A system as recited in claim **13**, wherein the request is administered by a proxy component.

17. A system as recited in claim **13**, wherein the data structure contains no logic.

18. A system as recited in claim **14**, wherein the business object is instantiated by an action of the client.

* * * * *

UNITED STATES PATENT AND TRADEMARK OFFICE

CERTIFICATE OF CORRECTION

PATENT NO. : 6,615,253 B1
DATED : September 2, 2003
INVENTOR(S) : Michel K. Bowman-Amuah

It is certified that error appears in the above-identified patent and that said Letters Patent is hereby corrected as shown below:

Title page,
Item [57], **ABSTRACT**, should read as follows:

--

A system and method are provided for efficiently retrieving data. A total amount and type of data that may be used during an application executed by a client is determined in response to a first client request. The total amount and type of data that may be used during execution of the application is a superset of the data required to satisfy the first client request. In a single call, the total amount and type of data from a server is requested over a network. All of the requested data is bundled into a data structure by the server in response to the single call. The bundled data structure is sent to the client over the network and the data of the data structure is cached on the client. The cached data of the data structure is used as needed during execution of the application on the client in response to subsequent client requests.

--

Column 2,
Lines 20-30, should be replaced with the following paragraph:

--

A system and method are provided for efficiently retrieving data. A total amount and type of data that may be used during an application executed by a client is determined in response to a first client request. The total amount and type of data that may be used during execution of the application is a superset of the data required to satisfy the first client request. In a single call, the total amount and type of data from a server is requested over a network. All of the requested data is bundled into a data structure by the server in response to the single call. The bundled data structure is sent to the client over the network and the data of the data structure is cached on the client. The cached data of the data structure is used as needed during execution of the application on the client in response to subsequent client requests.

Column 310,
Lines 3 and 37, delete the second occurrence of the word "the". --

Signed and Sealed this

Twentieth Day of January, 2004



JON W. DUDAS
Acting Director of the United States Patent and Trademark Office