

SOFTWARE ARCHITECTURE

FOUNDATIONS, THEORY, AND PRACTICE

Richard N. Taylor
Nenad Medvidovic
Eric M. Dashofy

**WILEY
STUDENT
EDITION**

RESTRICTED!
FOR SALE ONLY IN
INDIA, BANGLADESH, NEPAL,
PAKISTAN, SRI LANKA
& BHUTAN

WILEY

Pets 3019
Depun: medvidovic
Date: 5.29.20 Page 1 of
Kelli C. Nordin, CSR

SOFTWARE ARCHITECTURE

Foundations,
Theory, and Practice

Richard N. Taylor

University of California, Irvine

Nenad Medvidović

University of Southern California

Eric M. Dashofy

The Aerospace Corporation

WILEY

0103
pivovarov
15.05.2

To Lily May, Lyrica, and Richard
- *Richard N. Taylor*

To Pavle and Sasha
- *Nenad Medvidović*

To Mom and Keith
- *Eric M. Dashofy*

Software Architecture: Foundations, Theory, and Practice

Authorized reprint by Wiley India Pvt. Ltd., 4435-36/7, Ansari Road, Daryaganj,
New Delhi – 110002.

Copyright © 2005, 2006 by John Wiley & Sons, Inc. All rights reserved.

All rights reserved. AUTHORIZED REPRINT OF THE EDITION PUBLISHED
BY JOHN WILEY & SONS, U.K. No part of this book may be reproduced in any
form without the written permission of the publisher.

Limits of Liability/ Disclaimer of Warranty: The publisher and the author make
no representations or warranties with respect to the accuracy or completeness of
the contents of this work and specifically disclaim all warranties, including without
limitation warranties of fitness for a particular purpose. No warranty may be
created or extended by sales or promotional materials. The advice and strategies
contained herein may not be suitable for every situation. This work is sold with the
understanding that the publisher is not engaged in rendering legal, accounting,
or other professional services. If professional assistance is required, the services of
a competent professional person should be sought. Neither the publisher nor the
author shall be liable for damages arising herefrom. The fact that an organization
or website is referred to in this work as a citation and/or a potential source of
further information does not mean that the author or the publisher endorses the
information the organization or website may provide or recommendations it may
make. Further, readers should be aware that Internet websites listed in this work
may have changed or disappeared between when this work was written and when it
is read.

Wiley also publishes its books in a variety of electronic formats. Some content that
appears in print may not be available in electronic books. For more information
about Wiley products, visit our website at: www.wiley.com.

Reprint: 2014

Printed at: Printed in India by Arxel,

ISBN: 978-81-265-2802-8

because of the action of gravity. In the game version, the physics of the universe must be simulated. For example, upon each tick of a game clock the values of the sensors could be computed by an environment simulator, rather than read from genuine sensor devices. As you compare the various examples, please be aware that the various lunar lander games are not equivalent. The purpose of the examples is to help develop understanding of the various styles through reference to a common application context—not a common, precise, and complete requirements specification.

Some of the styles discussed have complex details associated with their use that have been omitted here. Our primary objective is to illustrate a wide range of diverse styles that represent the capture of extensive experience. The material presented here can serve as an introduction to the styles; detailed instructions for their use must be sought elsewhere.

4.3.4 Simple Styles

Traditional Language-Influenced Styles

The two styles considered here reflect the program styles that result from the traditional use of programming languages such as C, C++, Java, and Pascal. These languages, of course, can be used to implement architectures in very different styles; discussed here are architectures that primarily reflect the basic organizational and control-flow relationships between components provided by these imperative languages.

Main Program and Subroutines. The main program and subroutine style should be instantly familiar to anyone who has programmed in a language such as C, Fortran, Pascal, or Basic. The components are the main program and the various subroutines (aka. procedures); the connectors between the components are procedure calls. For purposes of this discussion we specifically exclude programs that use shared memory (such as global variables) as a means of communication from being a part of this style.

Style: Main Program and Subroutines (no shared memory)

Summary: Decomposition based upon separation of functional processing steps.

Components: Main program and subroutines.

Connectors: Function/procedure calls.

Data elements: Values passed in/out of subroutines.

Topology: Static organization of components is hierarchical; full structure is a directed graph.

Additional constraints imposed: None.

Qualities yielded: Modularity: Subroutines may be replaced with different implementations as long as interface semantics are unaffected.

Typical uses: Small programs; pedagogical purposes.

Cautions: Typically fails to scale to large applications; inadequate attention to data structures. Unpredictable effort required to accommodate new requirements.

Relations to programming languages or environments: Traditional imperative programming languages, such as BASIC, Pascal, or C.

A simple version of the lunar lander game program is shown in Figure 4-7. Its design is based on the functional decomposition of the processing into repeated user interface steps

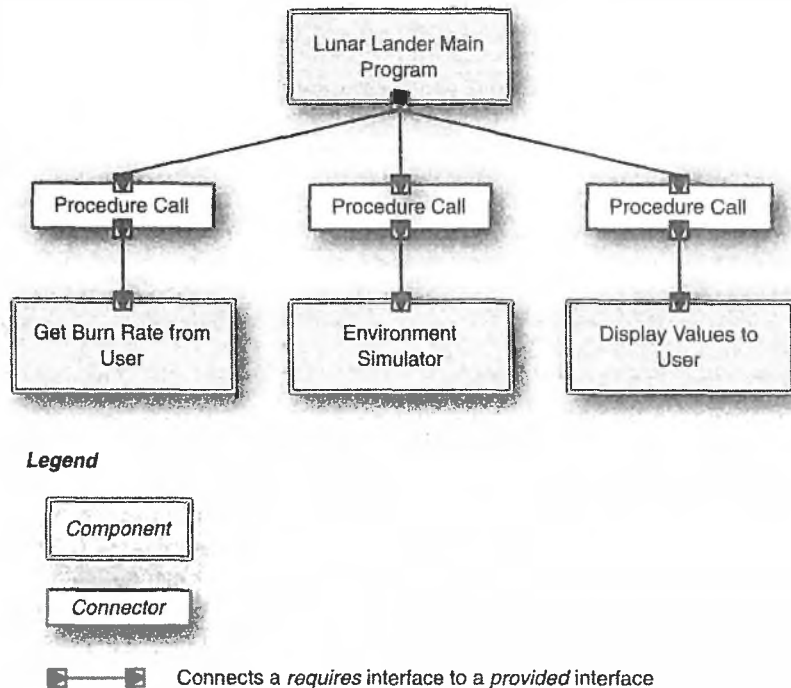


Figure 4-7.
Lunar Lander:
Main program
and subroutines.

(input and output) and all the processing that corresponds to simulating the environment and the actions of the spacecraft. Specifically, the main program displays greetings and instructions, then enters a loop wherein it calls the three subroutine components in turn. The first obtains the pilot's throttle-setting input. The second component primarily serves as the environment simulator, determining how much fuel is left and what the altitude and descent rate are. The only flight control law in this simple game is the translation of a pilot-specified throttle percentage into a volume/second fuel-flow setting (the burn rate) to control the descent engine. The third component displays the updated state. In this game there is no clock, so one cycle through the functional processing corresponds to one clock tick.

Object-Oriented. The object-oriented style should be similarly familiar. The only structuring provided is that of a world of objects whose lifetimes vary according to use. Comprehending a program structured in this manner requires understanding the numerous static and dynamic relationships among the objects.

Summary: State strongly encapsulated with functions that operate on that state as objects. Objects must be instantiated before the objects' methods can be called.
Components: Objects (aka. instance of a class).
Connector: Method invocation (procedure calls to manipulate state).

Style:
Object-
Oriented

Data elements: Arguments to methods.

Topology: Can vary arbitrarily; components may share data and interface functions through inheritance hierarchies.

Additional constraints imposed: Commonly: shared memory (to support use of pointers), single-threaded.

Qualities yielded: Integrity of data operations: data manipulated only by appropriate functions. Abstraction: implementation details hidden.

Typical uses: Applications where the designer wants a close correlation between entities in the physical world and entities in the program; pedagogy; applications involving complex, dynamic data structures.

Cautions: Use in distributed applications requires extensive middleware to provide access to remote objects. Relatively inefficient for high-performance applications with large, regular numeric data structures, such as in scientific computing. Lack of additional structuring principles can result in highly complex applications.

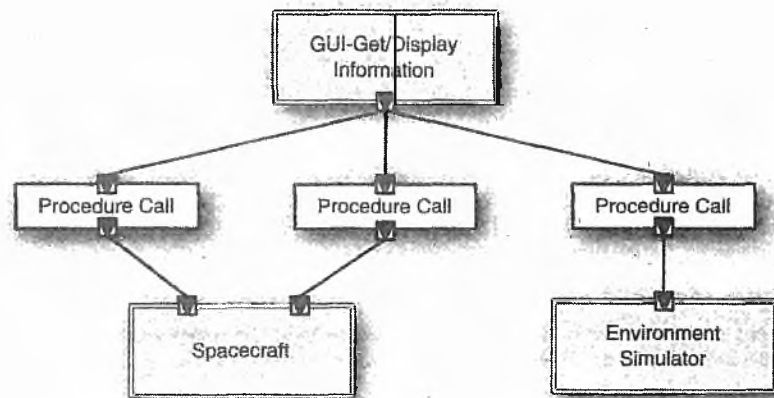
Relations to programming languages or environments: Java, C++.

The object-oriented design of the lunar lander game shown in Figure 4-8 has three encapsulations: the spacecraft, the user's interface, and the environment (essentially a physics model that allows calculation of the descent rate). Note a contrast with the main program and subroutine's (MPS) design: Interactions here with the user are handled by a single object, which performs both input and output functions. The functional decomposition of the MPS design resulted in those functions being performed by separate subroutines. Some details of the object-oriented design are apparent in Figure 4-9, which is a Unified Modeling Language characterization of the objects.

Layered

Layered styles are as simple and familiar as those above that reflect the traditional use of programming languages such as C or Java. The essence of a layered style is that an

Figure 4-8.
Lunar Lander in
the object-
oriented style.



architecture is separated into ordered layers, wherein a program within one layer may obtain services from a layer below it. The virtual machines style is familiar to anyone who has studied computer and operating systems architectures; the client-server layered style is ubiquitous in business applications. Both are discussed below.

Virtual Machines. In the virtual machines style, a *layer* offers a set of services (“a machine with a bunch of buttons and knobs”) that may be accessed by, at least, programs residing within the layer above it. The services that a layer offers can be termed the “provides interface” of the layer. The services may be implemented by various programs within the layer, but, when designing an architecture, from the perspective of the entities that use the layer’s services such distinctions are not apparent. In Figure 4-10, for instance, program A in layer 1 can access the services provided by layer 2; those services may in fact be implemented by programs B and C, but such details are not of concern to program A.

In a strict virtual machines style, programs at a given level may only access the services provided by the layer immediately below it. Thus, again referring to Figure 4-10, program A can only access services provided by layer 2, and not layer 3. A nonstrict virtual machines style would allow A to access the services of layer 3 as well.

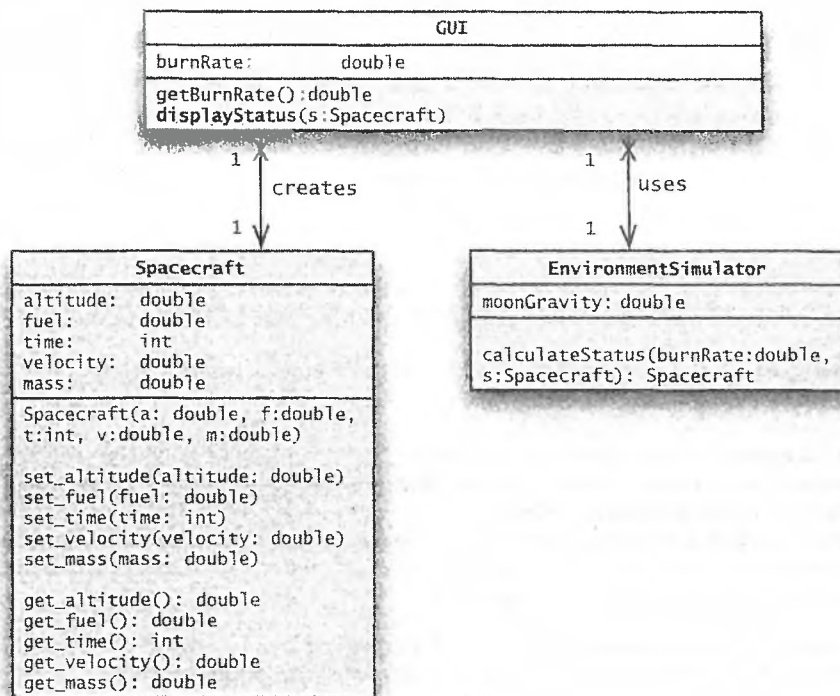
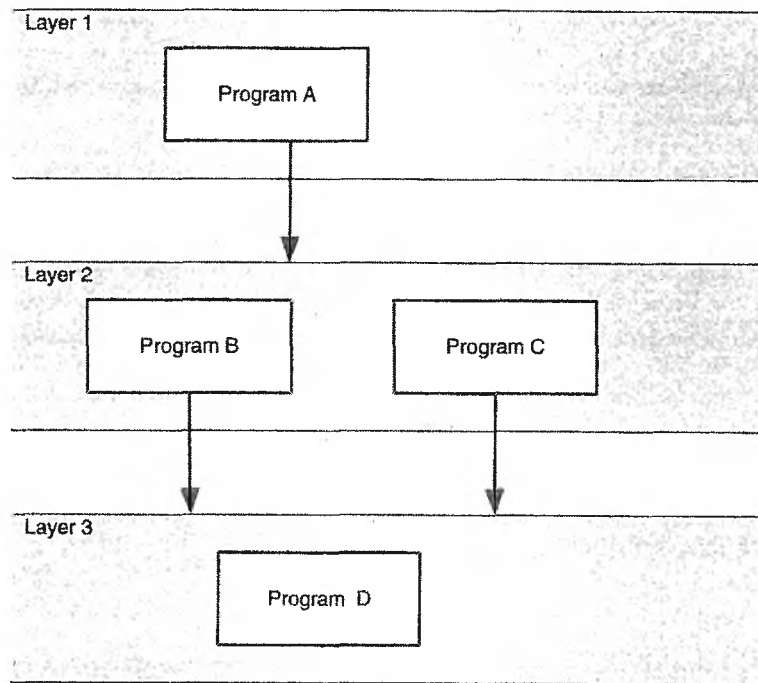


Figure 4-9.
JML
representation of
the classes used in
Figure 4-8.

Figure 4-10.
Notional virtual
machines
architecture.



Operating systems designs provide the most common use of the layered style. For instance, user applications would reside in layer 1, directory and file manipulation services on level 2, and disk drivers and volume management software at level 3. Networking protocols are also often implemented in a layered style.

Style: Virtual Machines

Summary: Consists of an ordered sequence of layers; each layer, or virtual machine, offers a set of services that may be accessed by programs (subcomponents) residing within the layer above it.

Components: Layers offering a set of services to other layers; typically comprising several programs (subcomponents).

Connectors: Typically procedure calls.

Data elements: Parameters passed between layers.

Topology: Linear, for strict virtual machines; a directed acyclic graph in looser interpretations.

Additional constraints imposed: None.

Qualities yielded: Clear dependence structure; software at upper levels immune to changes of implementation within lower levels as long as the service specifications are invariant. Software at lower levels fully independent of upper levels.

Typical uses: Operating system design; network protocol stacks.

Cautions: Strict virtual machines with many levels can be relatively inefficient.

A five-level version of the lunar lander game is shown in Figure 4-11. The top layer handles inputs received from the user through the keyboard; it calls the second layer to service those inputs. The second layer includes all the details of the lunar lander that pertain to the game logic and the environment simulator. It calls the third layer to begin the process of displaying the updated Lander state to the user. The third layer is a generic, two-dimensional game engine. Such a layer is capable of supporting any of a wide variety of games that only require two-dimensional graphics. The fourth layer is the operating system, which provides, among other things, platform-specific user interface (UI) support, such as window management. The bottom layer is provided by firmware or hardware, and

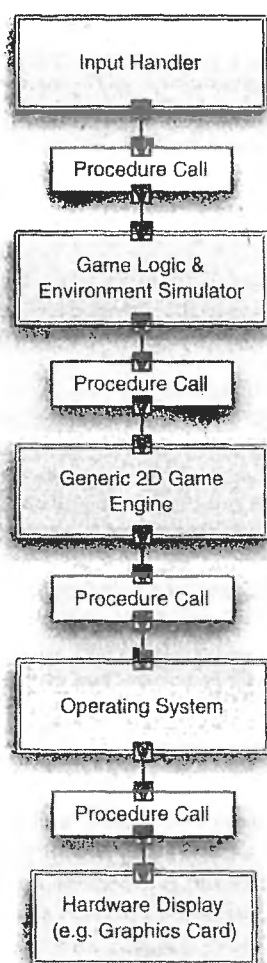


Figure 4-11.
*Lunar Lander in
the virtual
machines style.*

drives the physical user interface display. Each layer is independent of the layers above it, and only requires access to the services of the layer immediately below it. Note that the focus of this architecture is different from the preceding examples. The explicit inclusion here of a game engine, the operating system, and hardware is intended to help illustrate the layered style.

Client-Server. The client-server style can be simply understood as a two-layer virtual machine with network connections. That is, the server is the virtual machine below the clients, each of which accesses the virtual machine's interfaces via remote procedure calls or equivalent network access methods. Typically, there are multiple clients that access the same server; the clients are mutually independent. The obligation of the server is to provide the specific services requested by each client. Clients are sometimes referred to as "thin" or "thick," reflecting whether they include any significant processing beyond user interface functions.

Grossly simplified, a bank's automated teller machine (ATM) system is a client-server application. The clients are the ATMs and the server is the bank, which keeps track of all accounts and controls all of the transactions requested by users at the ATMs.

Style: Client-Server

Summary: Clients send service requests to the server, which performs the required functions and replies as needed with the requested information. Communication is initiated by the clients.

Components: Clients and server.

Connectors: Remote procedure call, network protocols.

Data elements: Parameters and return values as sent by the connectors.

Topology: Two-level, with multiple clients making requests to the server.

Additional constraints imposed: Client-to-client communication prohibited.

Qualities yielded: Centralization of computation and data at the server, with the information made available to remote clients. A single powerful server can service many clients.

Typical uses: Applications where centralization of data is required, or where processing and data storage benefit from a high-capacity machine, and where clients primarily perform simple user interface tasks, such as many business applications.

Cautions: When the network bandwidth is limited and there are a large number of client requests.

A multiplayer version of the lunar lander game is shown in Figure 4-12. In the system shown, three players simultaneously and independently play the game. All game state, game logic, and environment simulation is performed on the server. The clients perform the user interface functions. The connectors in this example are remote procedure calls, which are procedure call connectors combined with a distributor. A distributor identifies network interaction paths and subsequently routes communication along those paths. (The intricacies of connector types and their substructures will be explained in detail in Chapter 5.)

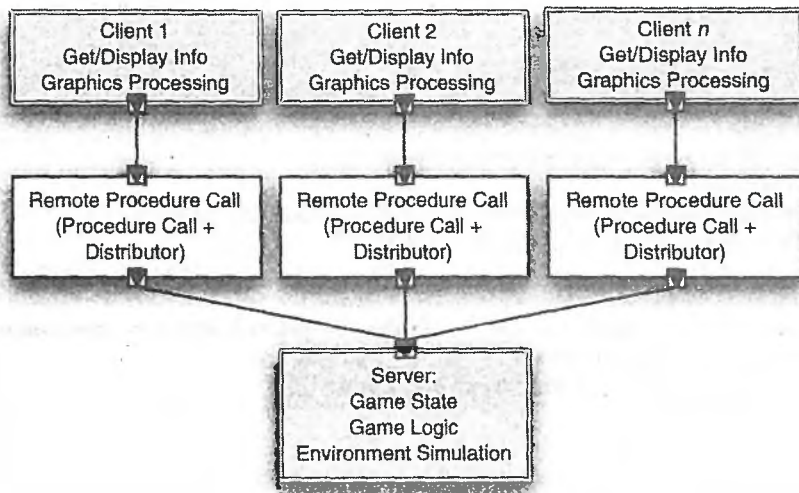


Figure 4-12.
A multiplayer
version of Lunar
Lander, in a
client-server
style.

Dataflow Styles

The two styles presented here can be characterized as dataflow styles. Their essence concerns the movement of data between independent processing elements.

Batch-sequential. The batch-sequential style is one of the oldest in the design of computer systems. It arose in the early days of data processing when the limitations of computing equipment required that large problems be subdivided into severable components that could communicate by the transfer of magnetic tapes.

A classic example of this style is shown in Figure 4-13. The application is one of updating a bank's record of all its accounts, based upon the overnight processing of the day's transactions (such as deposits or withdrawals from an account). The process starts with a master tape that has all of the bank's accounts on it, sorted by account number, and a daily tape that has the day's transactions on it.

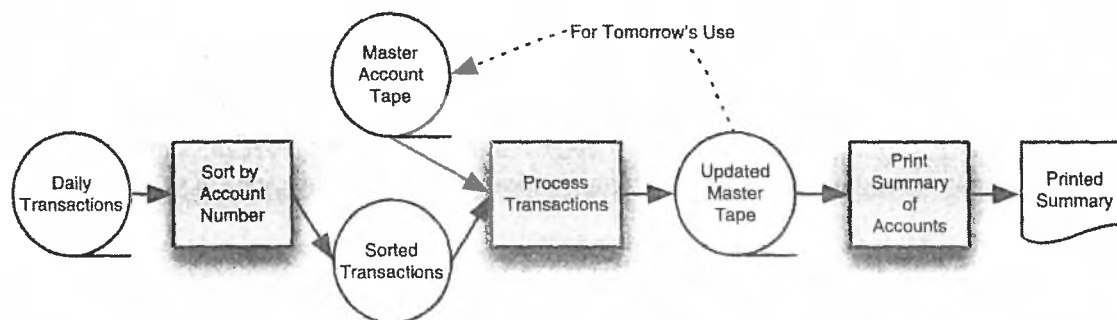


Figure 4-13. Financial records processed in batch-sequential style.

Style: Batch-Sequential.

Summary: Separate programs are executed in order; data is passed as an aggregate from one program to the next.

Components: Independent programs.

Connectors: The human hand carrying tapes between the programs, aka “sneaker-net.”⁵

Data elements: Explicit, aggregate elements passed from one component to the next upon completion of the producing program’s execution.

Topology: Linear.

Additional constraints imposed: One program runs at a time, to completion.

Qualities yielded: Severable execution; simplicity.

Typical uses: Transaction processing in financial systems.

Cautions: When interaction between the components is required; when concurrency between components is possible or required.

Relations to programming languages or environments: None.

Attempting to build a lunar lander game in the batch sequential style reveals how mismatched the style is to the application. The outline of such a (bizarre) design is presented in Figure 4-14. Here the game state is maintained on magnetic tapes.

Each functional processing step of the game is performed by a separate program. Upon performing the step’s function, the program produces an updated version of the game state, which is then handed off to the next program in the process. After the final step of displaying the game state is performed, the produced tape is carried back to the first program for another pass. This is obviously not a highly interactive, real-time game!

Pipe-and-Filter. Though radically different in detail from batch-sequential, the pipe-and-filter style introduced in Chapter 1 is also a dataflow style. Recall that the essence of the style is that streams of character data are passed from one filter program to another. The filters can operate concurrently, with no requirement for a producing component to finish before a component that consumes the producer’s output begins. Chapter 1 included a natural application of the pipe-and-filter style to the problem of producing a sorted list of selected file names.

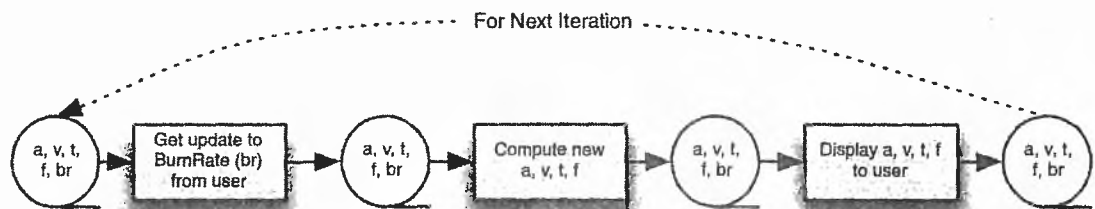


Figure 4-14. Lunar Lander: Batch-sequential.

⁵Savvy computer operators often wear athletic shoes, also known as “sneakers.”

Summary: Separate programs are executed, potentially concurrently; data is passed as a stream from one program to the next.

Components: Independent programs, known as filters.

Connectors: Explicit routers of data streams; service provided by operating system.

Data elements: Not explicit; must be (linear) data streams. In the typical Unix/Linux/DOS implementation the streams must be text.

Topology: Pipeline, though T fittings are possible.

Qualities yielded: Filters are mutually independent. Simple structure of incoming and outgoing data streams facilitates novel combinations of filters for new, composed applications.

Style:
Uniform
Pipe-and-
Filter.

Typical uses: Ubiquitous in operating system application programming.

Cautions: When complex data structures must be exchanged between filters; when interactivity between the programs is required.

Relations to programming languages or environments: Prevalent in Unix shells.

A design for the lunar lander game in the pipe-and-filter style appears in Figure 4-15. Get Burn Rate runs looping on its own, always polling the user for a value. When a value is generated it is sent off through the stream connector to the second filter. The second filter runs looping on its own as well, updating time as it chooses. It serves as the environment simulator and calculates the descent rate of the spacecraft. The third filter likewise loops, updating the display whenever new values become available on its input port. This design follows from a rather functional understanding of the game, similar to the main program and subroutines design. Functioning of a stream connector, and pipe-and-filter in general, is explored in much more depth in Chapter 5.

Shared State

The essence of shared state styles (sometimes colloquially referred to as "shared memory" styles) is that multiple components have access to the same data store, and communicate through that data store. This corresponds roughly to the ill-advised practice of using global data in C or Pascal programming. The difference is that with shared state styles the center of design attention is explicitly on these structured, shared repositories, and that consequently they are well-ordered and carefully managed. The use of global data in programming is usually only one of several means of communication between subprograms, and is done as an expedience.

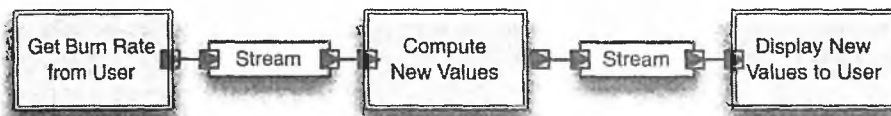


Figure 4-15.
*Lunar Lander in
pipe-and-filter
style.*

Blackboard. The blackboard style arose in artificial intelligence applications. The intuitive sense of the style is one of many diverse experts sitting around a blackboard, all attempting to cooperate in the solution of a large, complex problem. As any given expert recognizes some part of the problem on the blackboard for which he feels competent to solve, he grabs that subproblem, goes away and works on it, and when finished, returns and posts the solution on the blackboard. Posting that solution may enable another expert to identify a problem which he can solve, and so the process continues until the whole problem is solved. Thus the state of information on the blackboard determines the order of execution of the various "expert" programs.

Style:
Blackboard

Summary: Independent programs access and communicate exclusively through a global data repository, known as a blackboard.

Components: Independent programs, sometimes referred to as "knowledge sources," blackboard.

Connectors: Access to the blackboard may be by direct memory reference, or can be through a procedure call or a database query.

Data elements: Data stored in the blackboard.

Topology: Star topology, with the blackboard at the center.

Variants: In one version of the style, programs poll the blackboard to determine if any values of interest have changed; in another version, a blackboard manager notifies interested components of an update to the blackboard.

Qualities yielded: Complete solution strategies to complex problems do not have to be preplanned. Evolving views of the data/problem determine the strategies that are adopted.

Typical uses: Heuristic problem solving in artificial intelligence applications.

Cautions: When a well-structured solution strategy is available; when interactions between the independent programs require complex regulation; when representation of the data on the blackboard is subject to frequent change (requiring propagating changes to all the participating components).

Relations to programming languages or environments: Versions of the blackboard style that allow concurrency between the constituent programs require concurrency primitives for managing the shared blackboard.

Figure 4-16 shows the lunar lander program solved in a blackboard style. Here, a single connector regulates access from the various experts in manipulating the information on the blackboard. The blackboard maintains the game state; the experts perform the independent tasks of (1) updating the descent engine burn rate based upon input from the user, (2) displaying to the user the current state of the spacecraft and any other aspect of the game state, and (3) updating the game state based upon a time and physics model. That is, the third component determines how fast time passes and updates the game state in accordance with the passage of time and the physics model of the moon and the Lander.

Rule-Based/Expert System. A rule-based architecture is a highly specialized type of shared memory architecture. The shared memory is a so-called knowledge base, that is, a database that contains facts (statements of values of variables) and production rules which consist

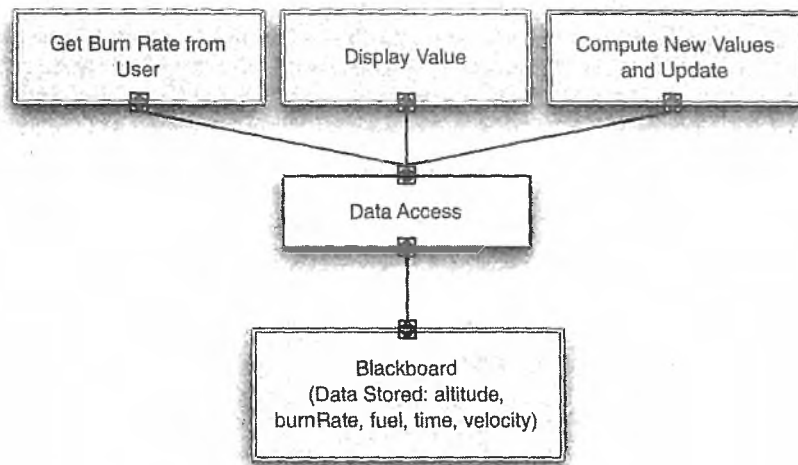


Figure 4-16.
Lunar Lander in
blackboard style.

of "if . . . then" clauses over the set of variables. A user interface component provides two modes: one for entering facts and production rules and another for entering queries, known as goals. Operating on the knowledge base, in response to user input, is an inference engine. Facts and production rules are added to the knowledge base; goals are compared against existing facts in the database. If an exact match is found, it returns *true* to the user interface. Otherwise, it evaluates the rules as necessary to determine the validity of the goal.

Summary: Inference engine parses user input and determines whether it is a fact/rule or a query. If it is a fact/rule, it adds this entry to the knowledge base. Otherwise, it queries the knowledge base for applicable rules and attempts to resolve the query.

Components: User interface, inference engine, knowledge base.

Connectors: Components are tightly interconnected, with direct procedure calls and/or shared data access.

Data Elements: Facts and queries.

Topology: Tightly coupled three-tier (direct connection of user interface, inference engine, and knowledge base).

Qualities yielded: Behavior of the application can be easily modified through dynamic addition or deletion of rules from the knowledge base. Small systems can be quickly prototyped. Thus useful for iteratively exploring problems whose general solution approach is unclear.

Typical uses: When the problem can be understood as matter of repeatedly resolving a set of predicates.

Cautions: When a large number of rules are involved, understanding the interactions between multiple rules affected by the same facts can become very difficult. Understanding the logical basis for a computed result can be as important as the result itself.

Relations to programming languages or environments: Prolog is a common language for building rule-based systems.

Style: Rule-Based/Expert System

Designing a solution for the lunar lander problem in the rule-based style requires thinking of the problem as one of maintaining a set of consistent facts about the spacecraft. This is fairly natural, as the physics model that determines the state of the spacecraft can be so understood. For instance, the passage of one second of time demands that the facts regarding the amount of fuel remaining, the velocity of the spacecraft, and the height of the spacecraft be brought up to date, consistently.

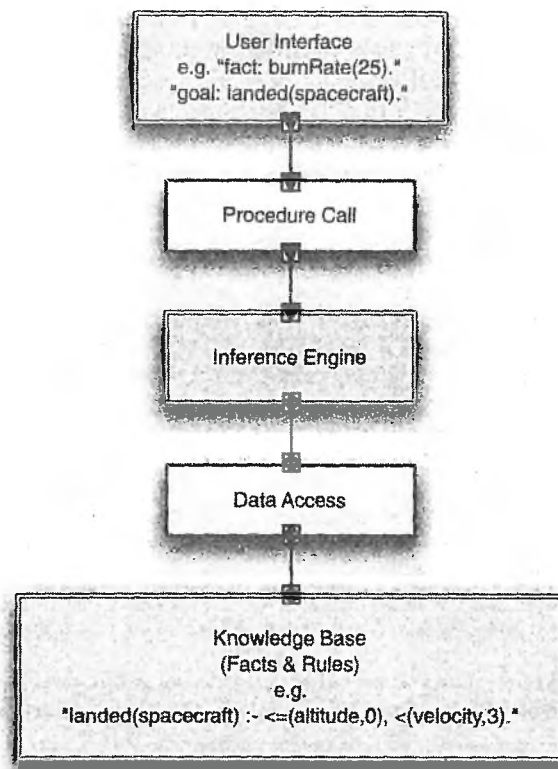
With regard to the user interaction model, in the example solution shown in Figure 4-17, the user enters the value of burn rate as a fact:

```
burnrate(25)
```

To see what the status of the spacecraft is, the user can switch to the goal mode and asks whether the spacecraft has landed safely:

```
landed(spacecraft)
```

Figure 4-17.
*Lunar Lander in
a rule-based
style.*



To handle this query, the inference engine queries the database. If the existing facts and other production rules satisfy the conditions "altitude ≤ 0" and "velocity < 3 ft/s," then the engine returns true to the user interface. Otherwise, it returns false.⁶

Interpreter

The distinctive characteristic of interpreter styles is dynamic, on-the-fly interpretation of commands. Commands are explicit statements, possibly created moments before they are executed, possibly encoded in human-readable and editable text. Commands are phrased in terms of predefined primitive commands. Interpretation proceeds by starting with an initial execution state, obtaining the first command to execute (possibly by reading some user input), executing the command over the current execution state, thereby modifying that state, then proceeding to execute the next command. Identification of the next command may be affected by the result of executing the previous command; indeed, in the general case the next command may be the product of the execution of the previous command.

The *basic interpreter* architectural style involves the execution of commands one at a time. Similarly, though typically at a larger granularity, the *mobile code* style involves the execution of one chunk of code at a time. The difference with mobile code is that the place where the commands are executed may vary over time.

Basic Interpreter. Execution of commands in the basic interpreter style is similar to the rule-based style described above, where the inference engine parses a command input and performs variable resolution based on the knowledge repository. In the case of the basic interpreter style, the command interpreter is more general (capable of more generic operations than performing inferences over a set of rules). The interpretation of a single command may involve numerous primitive operations. The knowledge repository is similarly more general since arbitrary data structures may be involved.

While perhaps sounding complicated, the interpreter style is quite familiar to what might be the world's largest cohort of programmers: the users of Microsoft Excel. Excel's formulas are in fact commands interpreted by the Excel execution engine—the interpreter. Similarly Excel's macros are interpreted by the Visual Basic interpreter. Many graphical editing programs are similarly interpreter based: a structure of commands describing the drawing is maintained by the drawing program. The drawing on the screen seen by the user is created by an internal interpreter running over the text, interpreting it, issuing drawing commands to the graphics engine.

⁶For a realistic model, the set of rules in the database will be extensive to perform the computation of a, v, t, f. A simple one-dimensional physics equation calculates altitude as follows:

$$\frac{1}{2} * (\text{accelerationThrust} - \text{accelerationGravity}) * t^2 + v * t + \text{altitudeOld}$$

However, a more realistic calculation of thrust is quite complicated (see notes at the end of the chapter for references).

**Style:
Interpreter**

Summary: Interpreter parses and executes input commands, updating the state maintained by the interpreter.

Components: Command interpreter, program/interpreter state, user interface.

Connectors: Typically the command interpreter, user interface, and state are very closely bound with direct procedure calls and shared state.

Data elements: Commands.

Topology: Tightly coupled three-tier; state can be separated from the interpreter.

Qualities yielded: Highly dynamic behavior possible, where the set of commands is dynamically modified. System architecture may remain constant while new capabilities are created based upon existing primitives.

Typical uses: Superb for end-user programmability; supports dynamically changing set of capabilities.

Cautions: When fast processing is needed (it takes longer to execute interpreted code than executable code); memory management may be an issue, especially when multiple interpreters are invoked simultaneously.

Relations to programming languages or environments: Lisp and Scheme are interpretive languages, and sometimes used when building other interpreters; Word/Excel macros.

In the version of the lunar lander game shown in Figure 4-18, for each command entered, the interpreter engine processes the code and updates the interpreter state as necessary. The commands in this case are specific directives regarding how the spacecraft should be manipulated—most importantly, how the descent engine should be throttled. The interpreter, thus, has been programmed specifically to interpret such commands. It may return values to the user, depending on the command. Thus, when the user enters “BurnRate(50),” the interpreter takes BurnRate as the command and the parameter as the amount of fuel to burn. It then calculates the necessary updates to the altitude, fuel level, and velocity. Time is simulated by having t incremented each occasion the user enters a BurnRate command. When the user enters the CheckStatus command, the user receives the current state of altitude, fuel, time, and velocity.

Mobile Code. As the name suggests, mobile code styles enable code to be transmitted to a remote host for interpretation. This may be due to a lack of local computing power, lack of resources, or due to large data sets remotely located. Mobile code may be classified as code on demand, remote evaluation, or mobile agent depending on where the code is being transmitted, who requested the transmission, and where the program state resides. Code on demand is when the initiator has resources and state but downloads code from another site to be executed locally. Remote evaluation is when the initiator has the code but lacks the resources (such as the interpreter) to execute the code. Thus, it transmits code to be processed at a remote host, such as in grid computing. Results are returned to the initiator. Mobile agent is when the initiator has the code and the state but some of resources are located elsewhere. Thus the initiator moves to a remote host with the code, the state, and some of the resources. Processing need not return to the initiator.

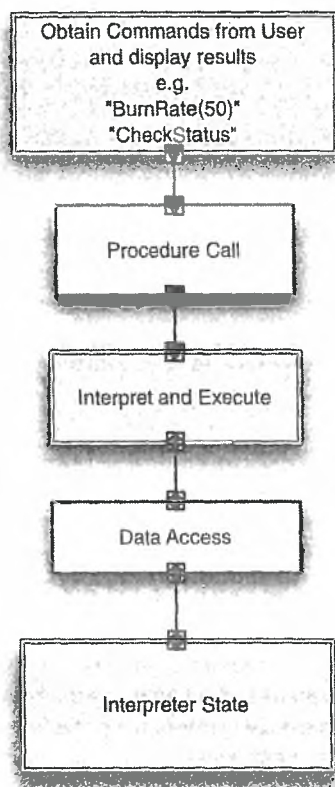


Figure 4-18.
Lunar Lander in
interpreter style.

In all of the mobile code styles, a data element (some representation of a program) is dynamically transformed into a data processing component (when those commands are interpreted or otherwise executed).

Summary: Code moves to be interpreted on another host; depending on the variant, state does also.

Components: Execution dock, which handles receipt and deployment of code and state; code compiler/interpreter.

Connectors: Network protocols and elements for packaging code and data for transmission.

Data elements: Representations of code as data; program state; data.

Topology: Network.

Variants: Code-on-demand, remote evaluation, and mobile agent.

Qualities yielded: Dynamic adaptability. Takes advantage of the aggregate computing power of available hosts; increased dependability through provision of migration to new hosts.

Typical uses: When processing large data sets in distributed locations, it becomes more efficient to have the code move to the location of these large data sets; when it is desirable to dynamically customize a local processing node through inclusion of external code.

Style:
Mobile Code

10.2.1 Basic Concepts

The overview of the key concepts that underlie software deployment, provided in this section, draws from a deployment technology characterization framework by Antonio Carzaniga, Alfonso Fuggetta, Richard Hall, Dennis Heimbigner, André van der Hoek, and Alexander Wolf (Carzaniga et al. 1998).

A software system is deployed on one or more hardware devices, referred to as *hosts* or *sites*. Each site provides a set of *resources* needed for hosting and executing the system or some of its subsystems. The resources include the different elements of the following.

- The hardware architecture (such as memory and CPU).
- The network architecture (such as available protocols and IP port numbers).
- The peripheral devices (such as hard disk and keyboard).
- The system software (such as operating system, device drivers, and middleware).
- Other application-level software (such as GUI builders and databases).
- The data resources (such as data files and Globally Unique Component Identifiers or GUIDs).

Resources can be either *exclusive* (such as IP port number, GUID) or *sharable* (such as CPU, data file).

A software system is composed from a specific set of components and connectors, with carefully prescribed interconnections and allowed interactions. Multiple versions of the components and connectors may exist, meaning that the system itself may have multiple versions as well. A *version* is defined to be a time-ordered revision, a platform-specific variant, or a functional variant.

Initial system deployment involves the transfer of system components or connectors from one or more *source* or *producer* hosts to one or more *destination* or *consumer* hosts. Subsequent deployment activity will typically involve introducing new functionality (that is, new components) to the system, or replacing existing components or connectors with different versions.

10.2.2 Deployment Activities

Architecture-driven software deployment comprises a process that must be carefully planned, modeled, analyzed, and finally effected or executed. We discuss these four activities in more detail below, and specifically focus on the relationship between each activity and software architecture.

Planning

It is critical that the deployment of a software system be carefully planned. Many important system properties, particularly in a distributed setting, will be affected by the system's deployment. For example, the system's latency in delivering a given service can be improved if the system is deployed such that the most frequent and voluminous interactions required for that service occur either locally or over reliable and capacious network links.

For any large, distributed system, many deployments—that is, mappings of software components and connectors onto hardware hosts—will be possible in principle. Some

product line, but this is not always the case. For example, an auto manufacturer, such as Ford, will build a number of cars on the same engineering platform but market them under different brand names, such as Lincoln, Mercury, and Mazda. Engineering product lines often share significant portions of their architecture, and the products within them may all conform to a broader DSSA. The product lines discussed in the remainder of this chapter are engineering product lines; however, Section 15.2.10 discusses strategies for unifying disparate products (perhaps those in a business product line) into an engineering product line.

15.2.6 Specifying the Architecture of a Product Line

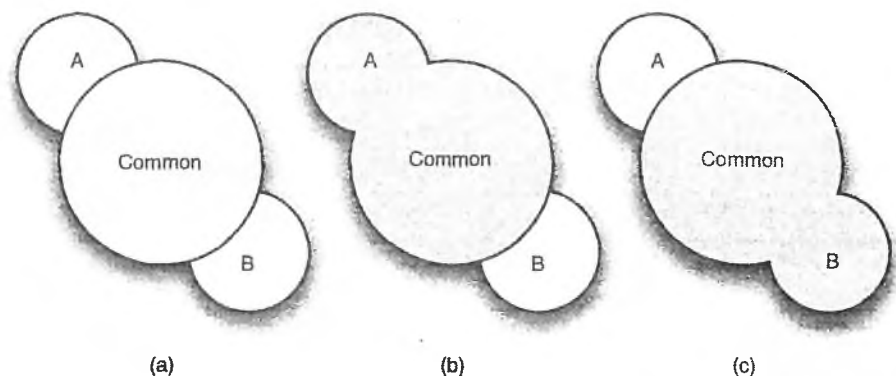
The architectures of product lines are somewhat different from reference architectures in a DSSA. In general, product-line architectures capture the complete architectures of multiple related products simultaneously. Conversely, reference architectures can be incomplete or partial. They may, for example, specify the architecture of a single product with loose guidance about how to adapt it to other contexts. They may also specify only invariant parts of an architecture and leave the rest up to solution developers.

Product-line architectures differ from reference architectures in two ways. The first difference is one of scope: Rather than attempting to describe the architectures of many (potentially diverse) solutions within a domain, product-line architectures focus on a specific, explicit set of related products, often developed by a single organization. The second difference is one of completeness: Product-line architectures generally capture multiple complete product architectures rather than leaving parts undefined.

Recall our characterization of architecture as the set of principal design decisions about a system. A product-line architecture can be similarly defined: A product line architecture captures simultaneously the principal design decisions of many related products. Some of these design decisions will be common among all the products, some will be common among a subset of the products, and some will be unique to individual products.

Figure 15-21 shows a product-line architecture with two products conceptualized as a Venn diagram. The large center circle represents the design decisions common to all products in the product line. The “A” shape represents the decisions that are unique to Product A. The “B” shape represents the decisions that are unique to product B. Figure 15-21(a) shows the three groups of design decisions. Figure 15-21(b) highlights Product

Figure 15-21.
A product-line
architecture
conceptualized as
a Venn diagram.



A: the union of the common design decisions and the “A” design decisions. Figure 15-21(c) shows Product B: the union of the common design decisions and the “B” design decisions.

To further discuss product-line architectures, we turn again to the Lunar Lander example. We have discussed different variants of the Lunar Lander game in earlier chapters, but each has been discussed in isolation. Product-line architectures give us the power to specify and discuss a family of Lunar Lander games in terms of explicit differences among the family members. The constructs that make this possible are *variants* and *versions*.

Variants

Extending an ordinary software architecture into a product line architecture can be accomplished through the addition of *variation points* to create *variant* architectures. Each variant architecture represents a different product—a member of the product line. A *variation point* is a set of design decisions that may or may not be included in the architecture of a given product. Variation points identify where the design decisions for a specific product architecture diverge from the design decisions for other products. Each variation point is accompanied by a condition that determines when the design decisions for that variation point are included in a product’s architecture. In the above example, the inclusion of the “A” design decisions can be seen as a variation point. The condition for this variation point is simple: The decisions are only included if the product being built is Product A.

By isolating and codifying variation points, a large number of product architectures can be described compactly by exploiting the different combinations of variations. Consider a hypothetical product line of Lunar Lander games that consists of three slightly different Lunar Lander products: Lunar Lander Lite, Lunar Lander Demo, and Lunar Lander Pro.

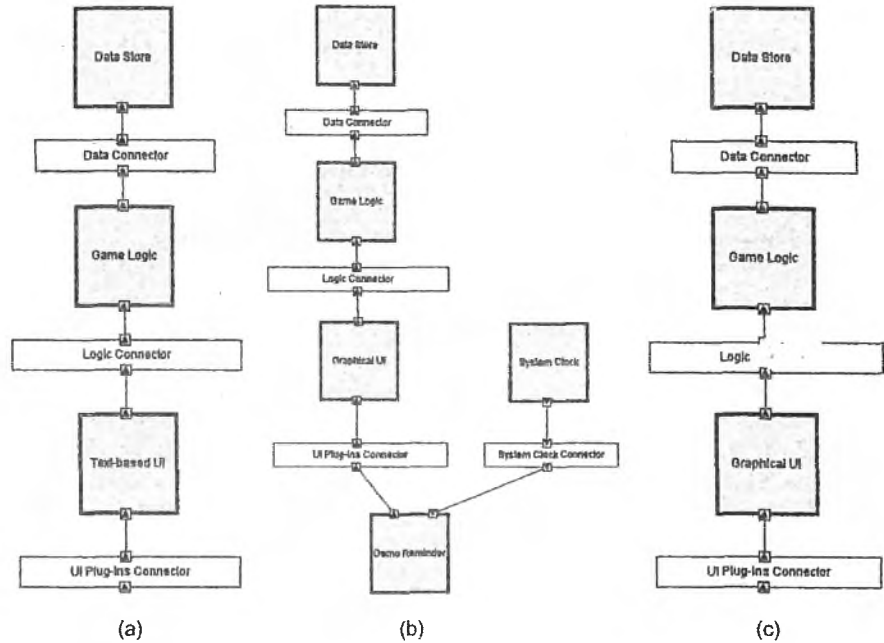
Lunar Lander Lite is intended as a freely distributed Lunar Lander game with a text-based user interface. It resembles the Lunar Lander games discussed in earlier chapters, such as those implemented in Chapter 9. Lunar Lander Pro is intended as a commercially sold Lunar Lander game with a fancier graphical user interface. Internally, the data store and game logic are identical to Lunar Lander Lite’s; the only difference is in the user interface component. Lunar Lander Demo is a freely distributed but time-limited version of Lunar Lander Pro. It has the same graphical user interface as Lunar Lander Pro, but it expires and is not playable thirty days after installation. Periodically, it reminds users that if they want to continue playing beyond the thirty-day limit, they are required to purchase or upgrade to Lunar Lander Pro.

The architectural structures of all three Lunar Lander products are shown in Figure 15-22.² Figure 15-22(a) shows Lunar Lander Lite, consisting of a data store and game logic component, along with a text-based UI. Figure 15-22(b) shows Lunar Lander Demo, with a graphical UI component in place of the text-based UI, and an additional demo reminder and system clock that plug into the user interface to remind the user to register. Figure 15-22(c) shows Lunar Lander Pro, identical to Lunar Lander Demo but missing the Demo Reminder and System Clock components.

Table 15-1 shows the elements—in this case components and connectors—included in each of the three product-line members. This kind of table is good for deciding whether

²The Lunar Lander example in this chapter will focus on structural design decisions; however, the techniques described apply, in general, to broader kinds of design decisions that go beyond just components and connectors.

Figure 15-22.
Architectures of
Lunar Lander:
(a) Lite, (b)
Demo, and (c)
Pro.



and how to create a product line. Here, it is obvious that there is significant commonality and overlap among the three products—good motivation for creating a product line.

Once the creation of a product line begins, it is a good idea to group low-level elements into variation points. Recall that each variation point represents a set of design decisions that may or may not be included in the architecture. This allows product-line architects to think about products in terms of features rather than low-level elements. For example, while it is conceivable to think of a product that contains a System Clock Connector but no System Clock, in practice this does not make much sense.

Table 15-1
Elements present
in each Lunar
Lander
product-line
member.

Components and Connectors	Product		
	Lite	Demo	Pro
Data Store	X	X	X
Data Store Connector	X	X	X
Game Logic	X	X	X
Game Logic Connector	X	X	X
Text-Based UI	X	X	
UI Plug-Ins Connector	X	X	X
Graphical UI		X	X
System Clock		X	
System Clock Connector		X	
Demo Reminder		X	

<i>Components and Connectors</i>	<i>Core Elements</i>	<i>Text UI</i>	<i>Graphical UI</i>	<i>Time Limited</i>
Data Store	X			
Data Store Connector	X			
Game Logic	X			
Game Logic Connector	X			
Text-Based UI		X		
UI Plug-Ins Connector	X			
Graphical UI			X	
System Clock				X
System Clock Connector				X
Demo Reminder				X

Table 15-2
Grouping
architectural
elements into
variation points.

A mapping of elements to variation points is shown in Table 15-2. Here, elements common to all products are grouped into Core Elements. Three other variation points are defined as well: the inclusion of a textual UI, a graphical UI, and whether or not the product is time-limited. The grouping of architectural elements or design decisions into features is often motivated by domain constraints, business goals, and the individual dependencies and exclusion relationships among the elements. Ultimately, these groupings must be chosen and defined by system architects—their creation is not an algorithmic process. Variation points will have dependencies and exclusion relationships as well: For example, all variation points depend on the inclusion of the Core Elements, and at least one of the two user interface variation points must be included in a product. These relationships and constraints must be carefully documented.

Once the variation points have been defined, individual products can be specified as combinations of the variation points. For business reasons, or reasons of variation point compatibility, not all combinations of variation points will be made into products.

Table 15-3 shows the creation of the three Lunar Lander products from the variation points defined in Table 15-2. Assuming that the core elements are always included, the three variation points altogether create 2^3 or eight potential products, of which three have been selected for construction. In addition to helping people understand the differences between the different product-line members, this kind of table is often useful in generating ideas for new products using the variation-point combinations *not* selected. For example, one could conceive of a Lunar Lander game that includes both a textual *and* a graphical user interface, perhaps giving the user multiple ways of interacting with the game and viewing its state. One could also imagine a time-limited version of the game with a textual UI, or a time-limited version with both user interfaces.

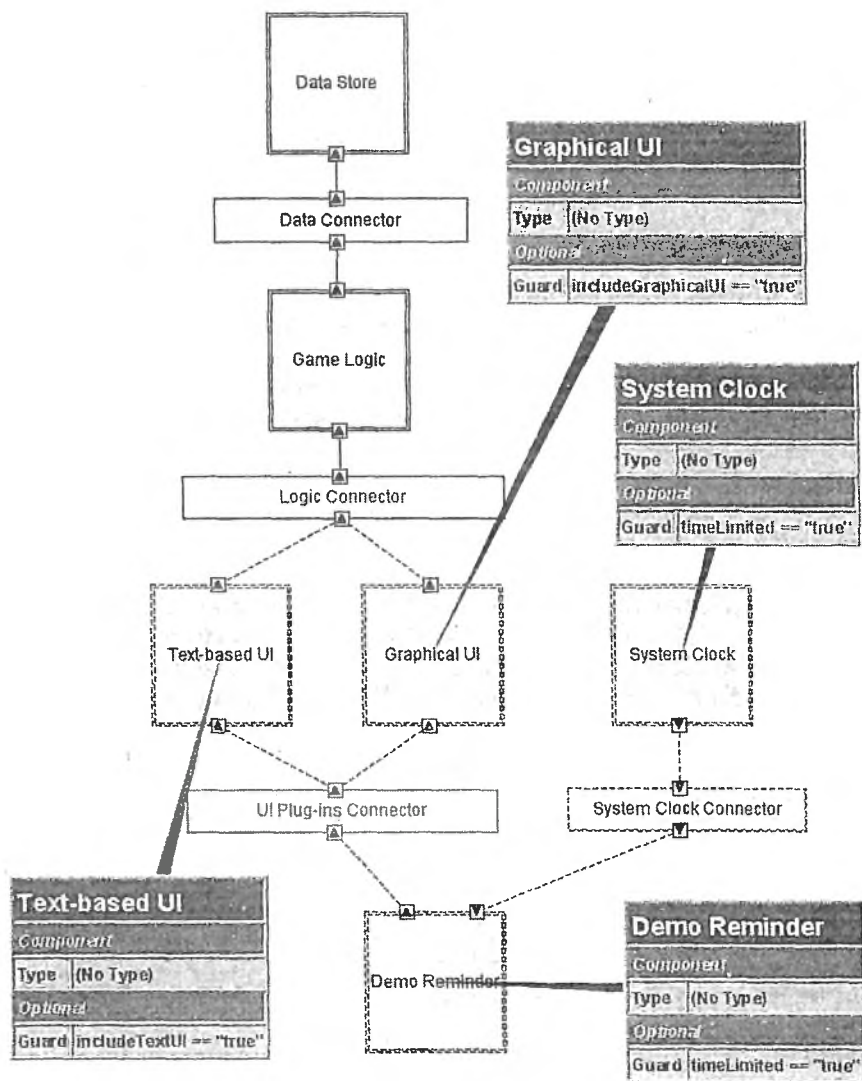
	<i>Lunar Lander Lite</i>	<i>Lunar Lander Demo</i>	<i>Lunar Lander Pro</i>
Core Elements	X	X	X
Text UI	X		
Graphical UI		X	X
Time Limited		X	

Table 15-3
Products as
combinations of
variation points.

By unifying common features and documenting these explicit variation points, we can combine these three individual architecture descriptions into a single product-line architecture. A few architecture description languages such as Koala (van Ommering et al. 2000) and xADL (Dashofy, van der Hoek, and Taylor 2005) allow you to do this directly.

The combined product line is shown in Figure 15-23, here in xADL's graphical visualization using xADL's product-line features (Dashofy and van der Hoek 2001) to express the points of variation. All elements from all products are shown together. Elements such as the *Data Store* and *Game Logic* components that are part of the core are represented normally. Elements that are included in one or more variation points are represented as

Figure 15-23.
Lunar Lander
product line
architecture.



optional. Each optional element is accompanied by a guard condition that indicates when it should be included in a product. In this product line, three variables—`includeTextUI`, `includeGraphicalUI`, and `timeLimited`—correspond to the variation points defined in Table 15-2. By setting the value of any of these variables to “true,” the elements corresponding to that variation point will be included in the architecture.

xADL’s use of a Boolean expression language to document these guard conditions gives it additional expressive power, including some ability to constrain relationships between features. For example, the *Text-based UI* and *Graphical UI* features could be made mutually exclusive by unifying the `includeTextUI` and `includeGraphicalUI` variables into a single variable, say ‘`uiType`,’ whose value would be either “graphical” or “textual,” but not both.

Selection

The process of reducing a product line into a smaller product line or a single product is known as *selection*. Selection involves evaluating the conditions associated with each variation point and either including or excluding the associated design decisions.

Consider the Lunar Lander product line. By fixing the `includeTextUI` variable to “false,” we have selected only products that do not include a textual UI. This reduces the total number of potential products from 8 to 4. By also fixing “`includeGraphicalUI`” to “true,” we can select only those products that include a graphical UI. Now, the total number of potential products is two. By fixing the final variable, `timeLimited`, to “true,” we can reduce the product line to a single product—one without a textual UI, with a graphical UI, and that is time-limited. This product, incidentally, is the Lunar Lander Demo product—something easily seen by referring to Table 15-3.

Without tool support, product-line selection must be performed manually—either on the fly in the minds of the stakeholders interpreting architectural models or by creating and maintaining explicit models of each product in the product line. Using an approach that supports the explicit specification of variation points and product lines such as Koala or xADL can make the selection process substantially easier. Specifically, in these approaches product-line selection can be performed automatically through the use of tools. xADL’s product-line selector tool allows users to bind values to guard variables, and then reduce a product-line architecture such as the one shown in Figure 15-23 into a single-product architecture such as the one shown in Figure 15-22 with a few clicks of the mouse.

15.2.7 Capturing Variations Over Time

The products in a product line are often alternatives to one another, intended for simultaneous development and release, as is the case with the Lunar Lander product line described above. However, product-line architectures also can be used to capture different versions of the same product over time. In general, different versions of the same product will have relatively similar architectures (unless something drastic such as a complete redesign or rewrite occurs). When this is the case, it is possible to enumerate the design decisions in both architectures and determine what changed from one version of a product to the next. These can be turned into variation points and encoded into a product line, where the different products are not alternatives, but rather the same product at different times.