

objectives are satisfied. A network protocol (such as HTTP/1.1) can be used, for example, to support this type of communication. The protocol supports moving MIME-encapsulated data throughout a distributed application; how a receiving component interprets and processes the encapsulated payload is a decision local to that component. Ensuring that the collection of components and connectors cooperate to achieve the objectives for the application is the responsibility of the system architect. For detailed examples of this type of architecture, see Chapter 4, including the discussion of the C2 style.

14.3.3 The Special Problems of On-the-Fly and Autonomous Adaptation

Adapting a system while it remains in operation presents special challenges, complicating the adaptation process. Some example situations where that complication cannot be avoided were given at the beginning of the chapter, and include applications whose continuous operation is essential to the success of some larger system, such as control software in a continuously operating chemical refinery.

Autonomous adaptation means that human involvement in the adaptation process is absent, or at least greatly minimized. Autonomous adaptation can be a complicating addition to on-the-fly adaptation, such as when the continuously operating system is isolated from external control, as in robotic planetary exploration. Such extreme circumstances are not the only ones where autonomous adaptation may be required, however. It may be, for instance, that rapid response requirements to partial system failure dictates the use of autonomous adaptation techniques.

Some of the challenges of on-the-fly adaptation can be seen through analogies. Living in a house that is undergoing a major renovation is sometimes necessary, and is seldom pleasant. The unpleasantness comes from attempting to carry on daily life while portions of the service architecture that we depend upon—plumbing, electricity, heat, security, and so on—are only partially or intermittently available. Life goes on, but not in the regular rhythms or with the usual comforts.

A more vivid, and instructive, example is surgery. The patient corresponds to the system being “adapted.” Clearly it is essential that while the adaptation is taking place—repair to a heart valve, for example—the patient must be kept alive. The various subsystems of the human (must) continue to function even though major disruptions take place. The various techniques and technologies of heart surgery, from anesthesia to cardiopulmonary bypass, are designed to maintain basic system performance while enabling major restructuring of a key component.

The key additional issues in on-the-fly adaptation are as follows:

- *Service strategy:* While the adaptation is taking place, will the system continue with a temporary service to fully replace the one being changed? Will it continue with a reduced level of service? A missing service? Can some service errors be tolerated?
- *System functioning during adaptation* may be maintained through the use of auxiliary components (such as the heart-lung machine in the surgery analogy), or may be degraded to a lower level of functionality (such as using a bathroom sink to wash dishes while the kitchen is remodeled in the home remodeling analogy).

- *Timing:* Identifying the occasions when the subject parts of the system can be altered. Depending on the service strategy adopted, the conditions under which the adaptation may proceed can be identified. Replacement of a component may only be allowed, for example, when it has finished servicing all outstanding requests to it, and has not initiated any requests on other components in the application.
- *Restoration or transfer of state:* If a component is replaced, does the replacement component need to know some or all of the state information that was held by its predecessor? The first calculations made by the new component may necessarily be functions of previous values computed and stored, in which case that state would have to be restored at the time the new component is started up.

An example may help make these concepts more clear. (Throughout this section we assume, consistent with the preceding parts of this chapter, that the smallest unit of change is a component and that component identity and boundaries are maintained in the implementation.)

Consider the case of a processor that controls the opening and closing of a valve in a chemical refinery based upon command inputs, temperature sensors, and flow meters. Such a processor may be designed in the sense-compute-control pattern discussed in Chapter 4. It may become necessary to replace one of the control components onboard, due to identification of an error in one of its algorithms. One plausible adaptation strategy is, at an arbitrary time, to cease processing of any incoming signals on its sensors, leave the valve's opening at whatever position it is in, swap out the offending component, swap in the new component, and then start the replaced component by first sampling all the input sensors, and then issuing new commands to the valve. Whether this strategy will work depends on, for example, whether any inputs missed during the swap were critical to the refinery's safety, or whether leaving the valve open in its last position could endanger the refinery before the new component was in place. For instance, if the valve was left in the "full on position" while the upgrade was taking place, and if the upgrade took very long to transpire, some downstream reaction in the refinery may be put at risk. A better strategy would likely entail buffering all inputs to the valve controller so that when the new component comes online it would ensure that every incoming value was processed, and instead of leaving the valve at its last setting, putting it into some safe position.

The appropriate service strategy for this example would likely be determined in consultation with the plant's chemical engineers. The amount of time the control processor could safely be preoccupied with the adaptation might well be determined by properties of the chemical processes in the refinery. If the valve is highly critical, a temporary controller may need to be put into place. Timing of the adaptation could be determined by properties of the plant, or perhaps by the state of the valve itself (perhaps the adaptation would occur only when the valve is closed).

Determining the Conditions for When to Effect a Change

The determination of the precise conditions under which it is appropriate to replace a component can be a function of the system as well as of the architecture. The system, as illustrated in the refinery example, may dictate that change happen only under certain external conditions. Alternatively, system domain analysis may reveal that change can happen at any time—even if it means that some outputs produced are erroneous—simply

because the system is so robust that it can tolerate some significant degree of error on the part of the software. For example, a satellite that gathers weather data, processes it, and sends the resulting information to the ground may be in this category. Weather data is continuous and is a phenomenon that changes slowly (relative to computer processing, at least), so simply losing some amount of the telemetry data is unlikely to matter.

Some applications will demand a much more conservative analysis, however. The principle is that it is safe to remove/replace a component only when such change will not adversely affect the functioning of the rest of the application. While precisely identifying all such conditions may be quite difficult, it is easier to specify sufficient conditions. Some sufficiency conditions are captured in the notion of *quiescence*. The intuition is that it is safe to replace a component when that component is inactive. A simple understanding of quiescence is that the component is not engaged in sending or receiving information, and internally is idle (all of its threads are idle). Inactivity, however, needs to be understood more comprehensively, in terms of any transactions to which the component may be a party. That is, a component C may initiate a set of actions that are performed by other components, itself staying inactive until the other components complete their task and return control to C. In this case, C is not quiescent until all actions that are part of a (conceptual) transaction are complete. Various shades of quiescence are defined in the sidebar, drawing from the work of Professors Jeff Kramer and Jeff Magee, who pioneered this area.

If an application is implemented with powerful, explicit connectors, such as message buses, then determining whether a component is engaged in communication is easy—the connectors may be used to make this determination. Knowing whether a component is engaged in a multiparty transaction, however, requires deeper analysis and will have to consider the behavioral characteristics of the architecture as a whole. The stronger the notion of quiescence required, the more complex will be the determination of when it is achieved.

A node in the active state can initiate and respond to transactions. The state identified as necessary for reconfiguration is the passive state, in which a node must respond to transactions, but it is not currently engaged in a transaction that it initiated, and it will not initiate new transactions. This passive state is so defined as to permit connected nodes to progress towards a passive state by completing outstanding transactions. In addition it contributes to system consistency by completing transactions.

For consistency during change we require a stronger property, that the node is not within a transaction and will neither receive nor initiate any new transactions i.e. all outstanding transactions are complete and no new ones will be initiated. Change quiescence of a node is defined as that state when the node is both passive and there are no outstanding transactions to which it need respond. Such a state depends not only on the node itself, but on the connected nodes. From "Change Management of Distributed Systems" (Kramer and Magee 1988) © 1988 ACM, Inc. Reprinted by permission.

**Quiescence,
According to
Kramer and
Magee**

Management of State

A component's *state* is the set of values that it maintains internally. When a component A is replaced with a component B, the question arises as to what B's state should be initialized to upon its startup. The easiest situation is one where no transfer or re-creation of corresponding state is needed. For example, many Unix filters do not maintain state; the output they produce is solely a function of the most recently read input, unaffected by

any preceding computation. Other applications may not be so designed. For instance, the user may have set a number of preferences; should the application be modified while the user is working with it, the preferences should be transferred. One effective strategy for dealing with transference of state is simply to externalize it: copy whatever state needs to be moved to a third party; when the replacement component comes online its first action is to initialize itself by drawing from the third party. This strategy, of course, presumes that the components were designed for replacement—a condition not often true.

Practical Tips

While the principles and issues discussed above apply to architectures generally, there are some practices and approaches that definitely ease the practical problems of supporting on-the-fly adaptation. Foremost are use of explicit, discrete, first-class connectors in the implementation and use of stateless components. To see the success of this simple advice, consider the architecture of the Web. As described elsewhere in this text, the Web is based upon these principles—and manifestly the Web is an application that continuously is undergoing change. Proxies and caches can be added and deleted; new Web server technologies put into place, new browsers installed. It all works because the interactions between the various components are discrete messages and the core protocol (HTTP/1.1) requires interactions to be context-free (“stateless” in the Web jargon)—a server does not need to maintain a history of interactions with a client in order to know how to respond to a new request.

Autonomous Change

Autonomous change is adaptation undertaken without outside control. The distinctive character of autonomous change is not the challenge of working without connection to, say, the Internet, but rather effecting the change process without involving humans. The desire to avoid involving humans is readily motivated: A self-adaptive system can change much more quickly than one involving people; consequently, self-adaptive change is likely to be much cheaper. Other situations may similarly motivate autonomous change. For instance, if a large number of very similar systems must be changed, but each with minor variances, an autonomous approach may work not only faster, but more accurately, as such repetitive tasks are often poorly performed by people—the tasks are simply too boring to keep one’s attention.

The difficulties entailed in achieving autonomous change are equally clear: All the activities discussed above must be completely realized as executable processes, from data observation, to analysis, to planning, to deployment, to run time system modification. Moreover, all these executable processes, and all the information involved in their execution, must reside on the autonomous system. Preeminent in this information is the architectural model of the system. Note that with this onboard, and kept faithful to the implementation, the system is continuously self-describing.

In practical terms, self-adaptive systems today are limited to rather simplistic situations for which a set of rule-based adaptation policies can be formulated. As previously discussed, this is appropriate for systems where the types of adaptation needs can be anticipated and encoded in simple terms. Expanding the range of systems and types of change for which autonomous solutions can be developed is an area of active research.

According to IBM, "Autonomic computing is an approach to self-managed computing systems with a minimum of human interference. The term derives from the body's autonomic nervous system, which controls key functions without conscious awareness or involvement. Autonomic computing is an emerging area of study and a Grand Challenge for the entire I/T community to address in earnest."

The initial autonomic computing manifesto (IBM 2001), as issued by IBM, focused attention on installation and management of IT infrastructure, such as corporate servers and networks, and their use in business processes. According to this manifesto there are at least eight key characteristics of autonomic systems:

1. "To be autonomic, a computing system needs to 'know itself'—and comprise components that also possess a system identity."
2. "An autonomic computing system must configure and reconfigure itself under varying and unpredictable conditions."
3. "An autonomic computing system never settles for the status quo—it always looks for ways to optimize its workings."
4. "An autonomic computing system must perform something akin to healing—it must be able to recover from routine and extraordinary events that might cause some of its parts to malfunction."
5. "A virtual world is no less dangerous than the physical one, so an autonomic computing system must be an expert in self-protection."
6. "An autonomic computing system knows its environment and the context surrounding its activity, and acts accordingly."
7. "An autonomic computing system cannot exist in a hermetic environment."
8. "Perhaps most critical for the user, an autonomic computing system will anticipate the optimized resources needed while keeping its complexity hidden."

—IBM 2001

Clearly IBM's vision for autonomic computing is consistent with the architecture-based approach to adaptation articulated in this chapter. Indeed, the first characteristic listed above is essentially a restatement of the requirement of maintaining an onboard model of a system's architecture, as discussed in the main body of the text.

References: (IBM 2002; Kephart and Chess 2003); Autonomic Computing Conference www.autonomic-conference.org/.

**"Autonomic Computing,"
IBM's Grand
Challenge
for the IT
Industry**

14.4 END MATTER

There is little question that applications will be adapted and changed over their lifetime. The real question is how expensive and how difficult will it be to effect those changes. While "designing for change" and the use of encapsulation is an important principle for assisting with change, much more powerful and capable mechanisms are required for meeting contemporary adaptation needs. This chapter has presented such mechanisms:

- A conceptual framework for governing the entire adaptation process.
- Architectural styles that assist with accommodating large-scale system change.

- Specific techniques for assisting with the major tasks of adaptation: monitoring, analyzing, planning, and effecting.

If a system's descriptive architecture and its prescriptive architecture are consistent, then adaptation can be orchestrated around changes to the architectural model. This approach, besides providing the intellectual tools for managing ordinary adaptation, opens the door to novel types of application evolution, such as wherein the architectural model is deployed with the application and adaptation processes on the target machine utilize the model in determining and carrying out changes.

This chapter concludes a three-chapter sequence on designing systems for non-functional properties. Chapter 12 introduced the main concepts and covered a variety of NFPs. Chapter 13 delved into the NFPs of security and trust; this chapter has examined the NFP of adaptability. The next chapter returns to a subject introduced in the very first chapter of the text: program families. With all the various elements of architecture-based development laid out in the intervening chapters, we next explore in depth how cost-effective development of a related set of applications can be achieved.

The Business Case

Adaptation is the means for preserving and leveraging sunk investment in software. To not adapt is to abandon the investment; the challenge is cost-effective adaptation.

An architecture-based approach to adaptation offers:

- New opportunities for products/applications.
- Better opportunities for maintaining deployed products.
- Better opportunities for responding to customer enhancement requests.
- Less downtime.
- New opportunities for third-party enhancement, while maintaining control of intellectual property.

The concept behind the last bulleted entry above is to expose parts of the architecture that enable third parties to add value through their proprietary extensions, but to do so without exposing the internals of components, or even allowing the third parties to become aware of the existence of large portions of the primary architecture. This strategy appears to be successful in, for example, commercial image-processing applications.

One risk of supporting adaptation is abuse based upon perceived ease. That is, if marketing, for example, is led to believe that the organization can effectively support product evolution with ease, then they may be less likely to carefully analyze new customer requests or to bundle requests for changes into meaningful packages.

14.5 REVIEW QUESTIONS

1. What are the fundamental causes or motivations for software adaptation?
2. What are the major activities of software adaptation?
3. For each of the major activities of adaptation, describe a technique that assists in performing that activity.
4. How can architectures be designed to make the task of third-party extension easier?
5. How do plug-in architectures work?
6. Why may quiescence be needed for run time replacement of a component?

14.6 EXERCISES

1. Consider the problem of modifying several of the lunar lander designs of Chapter 4 to serve as a Martian lander. Would the changes required be confined to components, or would connectors be involved?
2. Consider the problem of adapting the lunar lander designs of Chapter 4 to work by remote control. That is, instead of the Lander having a pilot on board, the craft is unmanned and a pilot on Earth must control the descent. What type of changes would be required to the components? To the connectors?
3. Consider the challenges faced in adapting a planetary exploration vehicle, such as the Cassini mission to Saturn. What would be a suitable architectural style for supporting in-flight system adaptation? Would you have a kernel that is nonupdatable? Why or why not?
4. What degree of quiescence would be required in your answer to question 3 when performing an update? To what extent does your answer depend on the type of update (that is, the degree of change involved)?
5. What are the differences (if any) between software architectures that support dynamic adaptation and fault-tolerant architectures? What circumstances are the approaches suited for?
6. Suppose you had optimized a connector in the implementation of a system for performance reasons and then need at some later date to modify the architecture. How would you approach adaptation of this system if the adaptation is performed offline? What if the adaptation must be performed dynamically?
7. Analyze the robotic architectures of Chapter 11 with respect to their ability to effectively accommodate changes such as requiring the robot to (a) perform a minor variation to an existing task, and (b) utilize a new tool, such as a new robotic arm, to perform an entirely new task.
8. Is preplanned change really an instance of dynamic architecture? Why or why not? What should the litmus test be?
9. (Research question) What is the role of dynamic adaptation (or just "adaptation") in the context of software product families? When should adaptation of a member of a family be considered creation of a new product?
10. (Research question) Does design of an application to support ease of adaptation conflict with the ability to ensure that any given instance of the application performs its intended task? In general, does ease of adaptation conflict with analyzability?

14.7 FURTHER READING

Since change has accompanied software development from the outset of the field there is no shortage of literature discussing how to adapt software. Indeed, the entire "Millennium Bug"/Y2K problem was one massive effort in software adaptation. In that case, of course, the change being effected was extremely narrowly focused. Sadly, for some Y2K systems, architecture changes were required since the two-digit fault was so implicitly part of those systems' design.

Stewart Brand's book on change in physical architectures (Brand 1994) is well worth an extensive read, for it has numerous insights on adaptation that have correspondences in software adaptation. If nothing else, you will discover why you love some buildings and hate others.

Some of the earliest work in architectural approaches to adaptation was performed at Imperial College by Jeff Kramer and Jeff Magee as part of the Regis (Magee,

Dulay, and Kramer 1994; Ng, Kramer, and Magee 1996) and Conic projects (Kramer and Magee 1990). Successive work with Darwin continues this theme (Magee and Kramer 1996).

Peyman Oreizy and colleagues are responsible for many of the key insights presented in this chapter, including the original conceptual model for the management of self-adaptive systems (Oreizy, Medvidović, and Taylor 1998; Oreizy et al. 1999) and the characterization of the architectural styles promoting adaptability presented in Section 14.3.2. The first of these two papers received an award in 2008, when it was recognized as the most influential paper from the 1998 International Conference on Software Engineering. As part of the award, the authors were invited to present a new paper on the topic, retrospectively analyzing the past decade's work in software adaptation and prospectively offering challenges for work

still to be done (Oriezy, Medvidović, and Taylor 2008). Additional recent work includes approaches based on knowledge-based systems (Georgas and Taylor 2004; Georgas, van der Hoek, and Taylor 2005). Gomma has explored the use of patterns to facilitate reconfiguration in (Gomma and Hussein 2004), and addresses the issue of quiescence.

Many more references are available from the proceedings of the leading workshops in the field, which include the Workshop on Software Engineering for Adaptive and Self-Managing Systems and the Workshop on Architecting Dependable Systems.

DISCUSSION SUMMARY

The current state of the art in architectural adaptation is characterized by a number of key trends and challenges. The first trend is the increasing use of formal methods and modeling languages to specify and verify architectural adaptations. This is driven by the need for rigorous and systematic approaches to handle the complexity of adaptive systems. The second trend is the growing emphasis on runtime adaptation, where architectural changes are made dynamically in response to changing requirements or environmental conditions. This is enabled by advances in monitoring, analysis, and decision-making techniques. The third trend is the integration of architectural adaptation with other software engineering concerns, such as security, privacy, and performance. This requires a holistic view of the system and its components. The fourth trend is the use of machine learning and artificial intelligence to optimize architectural adaptations. This involves learning from past experiences and making intelligent decisions about future adaptations. The fifth trend is the development of reusable architectural patterns and frameworks for adaptation. This helps to reduce the time and effort required to design and implement adaptive systems. The challenges in this field include the need for more standardized notation and modeling languages, the lack of mature tools and frameworks, the complexity of reasoning about adaptive systems, and the need for more rigorous evaluation and validation techniques.

Architectural adaptation is a key enabling technology for many emerging applications, such as autonomous systems, smart cities, and personalized services. It allows systems to evolve and adapt to changing requirements and environments, ensuring their continued relevance and effectiveness. However, the current state of the art in architectural adaptation is still in its early stages, and there are many challenges that need to be addressed. The key trends and challenges discussed in this summary provide a high-level overview of the current state of the field and highlight the areas that need further research and development. By addressing these challenges and leveraging the key trends, we can move towards more robust and effective architectural adaptation techniques, enabling the development of truly adaptive and resilient systems.

Domain-Specific Software Engineering

Software systems differ in terms of their purpose, size, complexity, provided functionality, quality needs, intended usage, expected or required execution context, involved stakeholders and their concerns, level of criticality to system users, and so on. An argument can be made that this is precisely why software is difficult to develop. However, such differences across individual systems are not unique to software engineering. They can be found in any complex system and any engineering discipline.

As an example, consider two different products of modern engineering: airplanes and television sets. Both airplanes and televisions are complex systems, but their functionalities differ a great deal. They also have very different quality requirements. While it is certainly important for a television to function reliably, in the case of airplanes reliability can be a matter of life and death. Travelers expect airplanes to be robust in the face of atypical situations such as storms, turbulence, loss of cabin air pressure or engine or electrical power, and physical damage to the aircraft. On the other hand, consumers understand when, say, an electrical power surge damages their TV set and are content to wait for the TV to be repaired or simply will buy a new one. People expect that an airplane may be in use for decades and that it will need to be refurbished and upgraded during that time. They do not have the same expectations of a television.

Airplanes and televisions are products of (very) different engineering problem *domains*. As a result of these differences, the respective skill sets of the engineers working in the two domains also are likely to be specialized and very different: It would be unrealistic to expect an electrical engineer working on airplanes to be able to switch to building televisions without (perhaps significant) retraining, and most people would feel uneasy if they were flying on an airplane built by engineers who usually design televisions. There are many obvious reasons for this—the main one is that the problems solved in

the two domains require different principles, techniques, processes, and tools that are honed separately over years or decades. Simply put, the two problem domains demand different solutions, and those developed for one domain are highly unlikely to fit the other.

Each application domain may further have several *subdomains*. For example, in the context of avionics, different subdomains may encompass rotary-wing aircraft (helicopters) versus fixed-wing aircraft (airplanes), military versus commercial aircraft, jet engine versus propeller engine aircraft, and so on. Within a single domain or one of its subdomains, organizations often focus on constructing *populations* of related systems. For example, Boeing engineers work on a wide variety of commercial and military aircraft that are likely to share some engineering characteristics. More easily recognized are the more closely related *families* of systems, or *product lines*, that exist within those populations. In the case of Boeing, an example would be the 7X7 family of passenger airliners. A family may be defined even more narrowly, for example, to encompass different models of the Boeing 747 jumbo jet.

It is reasonable to expect that even within the domain of avionics, the skills of an engineer working on rotary-wing aircraft may differ from those of colleagues working on fixed-wing aircraft. Likewise, an engineer building a military jet may have different experience and expertise than an engineer building a commercial airliner. Even more specifically, engineers working within Boeing's commercial airliner division likely approach certain aspects of their craft differently from their counterparts at Airbus. Additionally, there is a great deal of overlap regarding *how* different members of the Boeing 7X7 family of passenger jets are constructed; while many of the underlying principles are the same, the comparable Airbus airliners are likely to have been designed and built differently.

An analogous narrowing of the problem scope and specialization of the engineers' skills can be observed in other domains. For example, in the consumer electronics domain, a Philips engineer is likely to have somewhat different skill sets from, say, a Sony engineer. As another example, within the automotive domain, some General Motors engineers may focus on the broad population of GM vehicles while others may focus more narrowly on, say, the family of Cadillacs. Figure 15-1 illustrates this progressive narrowing of the problem scope and the resulting growth in commonality among systems.

As with any profession, software engineers have basic skills, techniques, guidelines, and tools that are applicable independent of the target domain. Separation of concerns, modularity, object-orientation, design patterns, UML, Java, CORBA, and so on, comprise a common "toolbox" at the disposal of software engineers. It would not be unfair

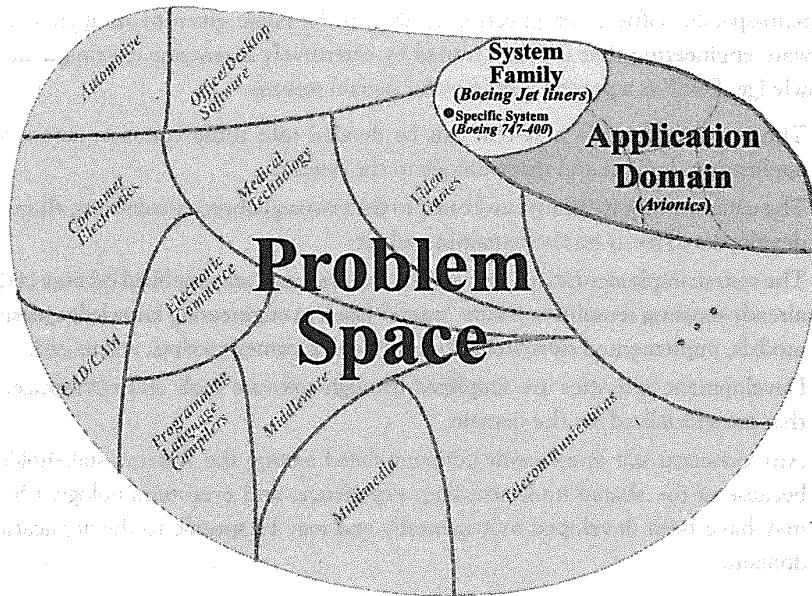


Figure 15-1. Engineers construct systems within many problem domains, which are broken down into subdomains, which in turn are broken down into families.

to think of these as equivalent to, say, Maxwell's laws, soldering irons, capacitors, resistors, and so on, that are at the disposal of an electrical engineer.

However, these basic tools and principles are only a rudimentary foundation for the construction of complex systems. They neither attempt to leverage the useful properties of a given problem domain nor help to identify and exploit the similarities that are likely to exist across systems in a given family. It would be too difficult, risky, and costly to attempt to build every software system using only these primitive tools. In the early days of computing, software engineers were forced to do exactly that. They had little or no knowledge about building software, especially within particular domains, so they had to develop software systems from "first principles."

The situation today is very different. A large amount of software engineering knowledge has been acquired through extensive experience (and costly failures) in many domains. Within these domains, many product populations and product lines have been developed and evolved. By leveraging knowledge from these experiences, engineers can build subsequent systems within the same domain or family more quickly, cheaply, and reliably. To use our earlier analogy, while the ability to build a television may not translate to the ability to build an airplane, the ability to build one airplane imparts a large amount of information about how to build another.

Domain-specific software engineering (DSSE) is the name given to an approach to software engineering that is characterized by extensively leveraging existing domain knowledge. DSSE is a powerful strategy for several reasons:

- The requirements for a system can be divided into those common across the application domain and those unique to the system.
- The common requirements can be tied to the existing canonical solutions, allowing developers to focus on the remaining subset.
- The system implementation, testing, and maintenance are simplified because of the already-existing reusable software “assets” (such as engineering knowledge, design models, implemented subsystems, test suites, deployment scripts, and so on).
- Development activities are simplified through software tools and environments that are specialized for the domain.
- Any concerns are more easily communicated among the system’s stakeholders because of the shared understanding, experience, and even terminology, which may have been developed incrementally and may be specific to the application domain.

As we will elaborate in this chapter, DSSE combines insights from three principal areas:

1. The *domain*, which scopes the discourse, the problem space, and the solution space.
2. *Business goals*, which motivate the work and help engineers decide why they are doing what they are doing.
3. *Technology*, which is used to facilitate development and reuse of domain- and business-specific assets.

The respective roles of and relationships among these areas are depicted in Figure 15-2, which provides a conceptual basis for the remainder of this chapter. It allows us to define, relate, and explain appropriately two key facets of DSSE: domain-specific software architectures (DSSA) and product families, also referred to as product lines (PL). Even though they are closely related, these two areas of software engineering have received mostly separate treatments in the literature.

This chapter introduces and discusses the DSSE concepts foreshadowed above, with a particular focus on the role of software architecture in DSSE. By the end of the chapter, the reader will better understand how DSSE is different from ordinary architecture-based software engineering, its benefits, the relationship between DSSAs and product lines, and how to apply the resulting concepts to capture explicitly and effectively architectural solutions that span multiple systems across a domain or, more narrowly, across a system family.

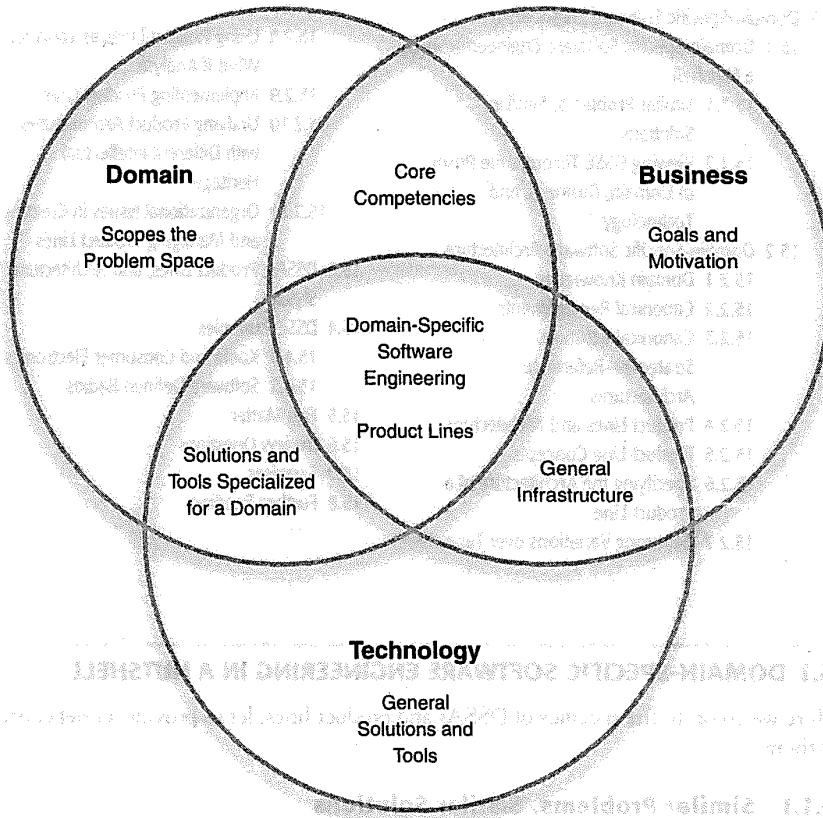


Figure 15-2. Domain-specific software engineering requires organizations and engineers to leverage different aspects of three inter-related areas: domain, business, and technology.

**Outline of
Chapter 15**

15 Domain-Specific Software Engineering	15.2.8 Using Product Lines as Tools for What-If Analysis
15.1 Domain-Specific Software Engineering in a Nutshell	15.2.9 Implementing Product Lines
15.1.1 Similar Problems, Similar Solutions	15.2.10 Unifying Product Architectures with Different Intellectual Heritage
15.1.2 Viewing DSSE Through the Prism of Domain, Business, and Technology	15.2.11 Organizational Issues in Creating and Managing Product Lines
15.2 Domain-Specific Software Architecture	15.3 DSSAs, Product Lines, and Architectural Styles
15.2.1 Domain Knowledge	15.4 DSSE Examples
15.2.2 Canonical Requirements	15.4.1 Koala and Consumer Electronics
15.2.3 Canonical Solution Strategies—Reference Architectures	15.4.2 Software Defined Radios
15.2.4 Product Lines and Architecture	15.5 End Matter
15.2.5 Product-Line Concepts	15.6 Review Questions
15.2.6 Specifying the Architecture of a Product Line	15.7 Exercises
15.2.7 Capturing Variations over Time	15.8 Further Reading

15.1 DOMAIN-SPECIFIC SOFTWARE ENGINEERING IN A NUTSHELL

Before we go on to the specifics of DSSAs and product lines, let us provide a brief context for them.

15.1.1 Similar Problems, Similar Solutions

For the purpose of illustrating our discussion, we show a highly simplified view of traditional software development in Figure 15-3: A team of software engineers is typically given a description of a problem that they are to solve. That description may be detailed or may be cursory; it may be written down precisely or stated verbally (and ambiguously) by a human customer or prospective user; the description may be relatively complete or it may emerge incrementally during the development process. In any case, the principal task of the software engineers is to find a way of taking the problem description, which exists in the *problem space*, and mapping it to a software system, which exists in the *solution space*. Doing so in general is difficult because the two spaces usually are characterized by different concepts, with different terminologies and different properties. Doing so is also difficult because, in general, there are often many possibilities for addressing a given software requirement, ranging from the programming language in which the requirement will be implemented, to the code-level constructs used to realize the requirement, to the hardware platform on which it will execute, to the different ways of modularizing the system (including choosing not to modularize it at all), and so on. This is, in part, why historically it has been such a challenge for developers to ensure desired properties in software systems: too many choices without a clear indication of which choice works best, and why.

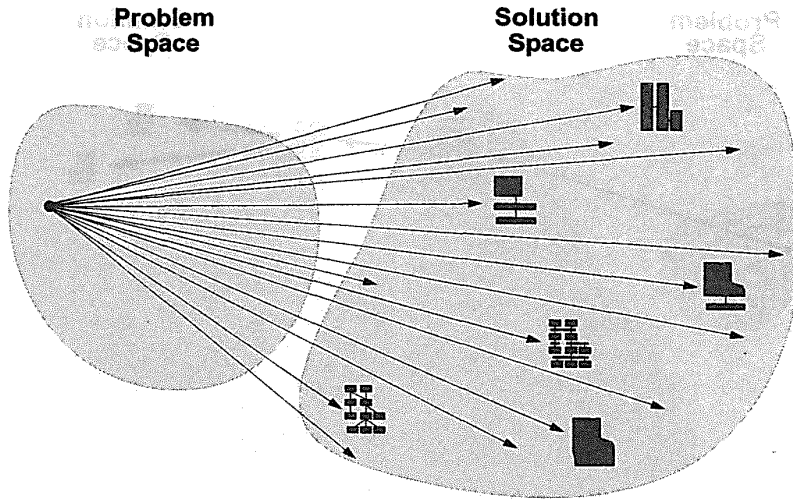


Figure 15-3.
A deliberately
simplified view of
traditional
software
development:
Any given
software
development
problem can be
solved in a large
number of
different ways.

Software architecture-based development addresses this problem in part by elevating the discourse to a higher plane with fewer choices: What are the principal components needed for the given system? What are their interactions? What are their compositions into system configurations that effectively solve the problem at hand? Figure 15-4 depicts this approach to system development. Again, the picture is deliberately oversimplified to illustrate the point: A problem will often have a more constrained number of software

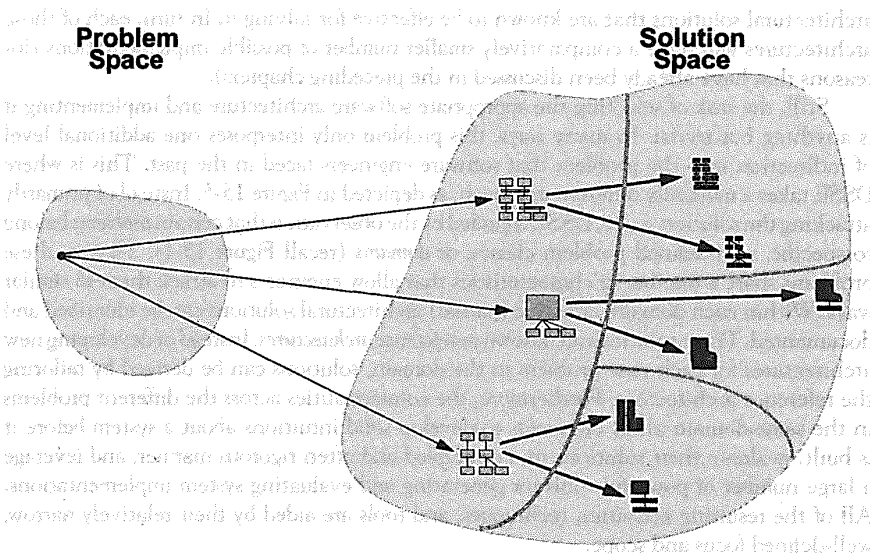


Figure 15-4.
A deliberately
simplified view of
architecture-
based software
development:
Any given
software
development
problem can be
solved by a finite
number of
software
architectures.

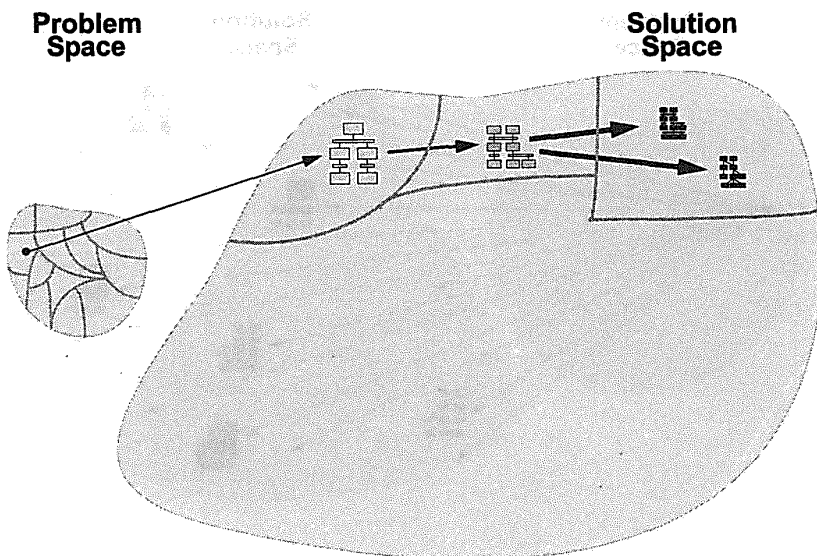


Figure 15-5. A deliberately simplified view of DSSE: Some software development problems belong to specific classes of problems for which known (partial) architectural and implementation solutions exist. Those partial architectures are then tailored to the specific problem at hand and implemented using well-understood techniques.

architectural solutions that are known to be effective for solving it. In turn, each of those architectures will have a comparatively smaller number of possible implementations (for reasons that have already been discussed in the preceding chapters).

Still, the task of selecting the appropriate software architecture and implementing it is anything but trivial. In many ways, this problem only interposes one additional level of indirection into the problem that software engineers faced in the past. This is where DSSE takes a markedly different approach, as depicted in Figure 15-5: Instead of primarily attacking the solution space, DSSE is guided by the observation that certain *problems* belong to specific, well-defined problem classes, or *domains* (recall Figure 15-1). That is, these problems share a number of characteristics that allow engineers to attack them in similar ways. Within each domain, effective (partial) architectural solutions can be identified and documented. These solutions are known as *reference architectures*. Instead of developing new architectures for each new problem in the domain, solutions can be derived by tailoring the reference architecture. Furthermore, the commonalities across the different problems in the same domain allow engineers to develop solid intuitions about a system before it is built, evaluate their solutions in a principled and often rigorous manner, and leverage a large number of powerful tools for generating and evaluating system implementations. All of the resulting activities, techniques, and tools are aided by their relatively narrow, well-defined focus and scope.

15.1.2 Viewing DSSE Through the Prism of Domain, Business, and Technology

As argued in the chapter's introduction and depicted in Figure 15-2, three principal concerns of DSSE are domain, business, and technology (Medvidović, Dashofy, and Taylor 2007). The prominence of each of the three concerns, and their exact mix, will differ across organizations and projects. Those differences have clear implications on the organizations and projects concerned, as well as on the stakeholders, the processes, and ultimately the products. We discuss each area in the diagram from Figure 15-2 in more detail below.

Domain: The domain, independent of business and technology concerns, establishes a problem space. It has defined characteristics, a vocabulary, a motivation (why this domain exists), and so on. This area will be further expanded below.

Business: Business, independent of any domain or technology concerns, is largely concerned with human goals: improving people's quality of life through the creation of new products, attaining money, power, notoriety, and so on. These goals motivate people to solve problems. Note that this is not meant to imply that domain-specific software must be sold or otherwise developed for the purpose of attaining monetary profits. However, the goal of using DSSE is to optimize certain aspects of software engineering: reducing cost or time to develop, improving the quality of products in the market (even open-source products), and so on.

Technology: Technology, independent of a domain or business goals, comprises tools, applications, reusable components, infrastructure, and methods that can be applied generally. In this sense, technology could be characterized as "solutions without problems."

Domain + Business: When business goals are applied to a particular domain, expertise and core competencies emerge. Business organizations specialize their skills to optimize them for particular domains: building televisions or airplanes, for example.

Business + Technology: Regardless of the domain(s) in which it operates, a business organization will acquire and develop technologies that are relevant to its overall goals but that can be applied to many domains. For example, any software development organization undoubtedly will have an infrastructure containing compilers, operating systems, networks, office applications, and so on that does not apply specifically to any domain.

Domain + Technology: This intersection contains tools, methods, and even architectures that are specifically applicable to a particular domain, but are independent of any particular business goal. For example, a programming language and compiler that are specifically developed for building aircraft software would fall into this category.

Domain + Business + Technology: This is the core of domain-specific software engineering: business goals motivating the identification and creation of a solution in the problem space of a domain, facilitated by the use of technology.

We have outlined the various concepts and interactions that comprise domain-specific software engineering. Now, we will examine how software architecture can be leveraged

and specialized for application in the context of DSSE. We will study two key architecturally relevant areas of DSSE: domain-specific software architectures and product lines.

15.2 DOMAIN-SPECIFIC SOFTWARE ARCHITECTURE

A domain-specific software architecture (Tracz 1995; Coglianese, Smith, and Tracz 1992; Tracz 1994; Tracz, Coglianese, and Young 1993) comprises a codified body of knowledge about software development in a specific application domain. Barbara Hayes-Roth (Hayes-Roth et al. 1995) provides a useful operational definition of a domain-specific software architecture:

Definition. A domain-specific software architecture (DSSA) comprises:

- A reference architecture, which describes a general computational framework for a significant domain of applications.
- A component library, which contains reusable chunks of domain expertise.
- An application configuration method for selecting and configuring components within the architecture to meet particular application requirements.

Figure 15-6 provides an generalized overview of a DSSA-centric software development process. In addition to the usual *application engineering* activities in which developers engage, DSSAs involve a number of *domain engineering* activities. These activities result in models of the application domain's relevant entities, their characteristics, and their relationships; definitions of the key terminology; canonical requirements; design- and implementation-level solutions that are reusable across systems in the domain; and tool support specialized to aid development within the domain.

This body of assets does not come for free. Instead, it is built over time, as engineers amass experience (both good and bad) of building individual systems within a domain and try to generalize that experience for use in future systems. In the remainder of this section we will discuss the characteristics of application domains and domain models, the requirements that remain stable and those that change within a domain, and the corresponding solution strategies. We will illustrate the discussion with an example DSSA derived from the different Lunar Lander architectures discussed in the preceding chapters.

15.2.1 Domain Knowledge

As we have discussed, one of the main concerns of domain-specific software engineering is the problem domain itself. In order to exploit the properties of the domain, its key characteristics must be captured in a domain model. Simply put, a *domain model* (Batory, McAllester et al. 1995) is a representation of what happens in an application domain. This includes the *functions* being performed, the *objects* (also referred to as *entities*) performing the functions and those on which the functions are performed, and the *data* and *information* flowing among the entities and/or functions. In the parlance of the preceding section, a domain model deals with the problem space. Therefore, the above terms are used in their ordinary sense (for example, function as an operation performed on or by an entity “living” in the domain), rather than in the sense that software engineers typically assign to them (for example, function as a method that returns a value in a C++ program).

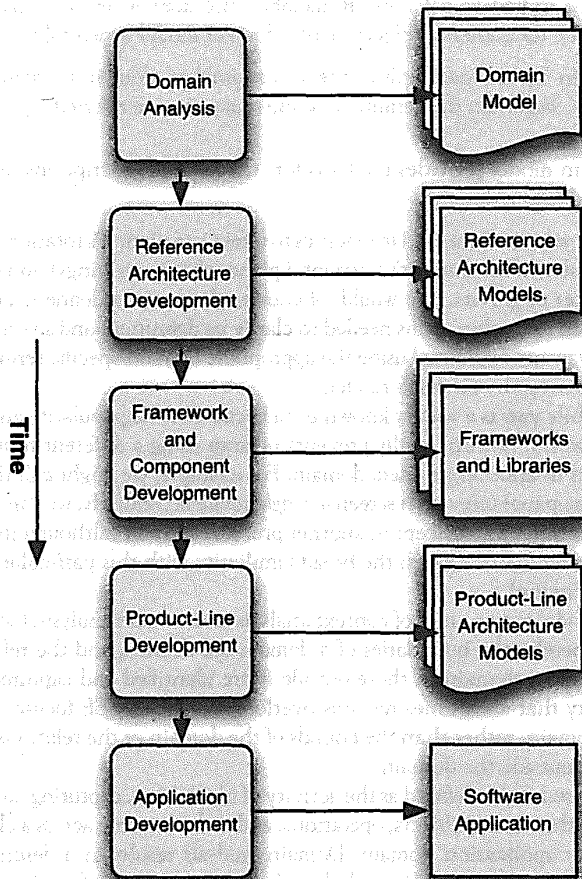


Figure 15-6.
Overview of the
DSSA process
and managed
artifacts.

If we take avionics as an example domain, the functions of interest would be aircraft takeoff, landing, taxiing, flying, pitch, roll, yaw, fueling, refueling (possibly in-flight), maintenance, and so on. Data and information flowing among these functions would be various pilot commands, instrument signals, warnings, status check messages, fuel consumption rates, data collected for the black boxes, and so on. The entities in the domain would include the flight instruments, the aircraft's rudder and wing flaps, the fuel tanks, the fuel transferred into the tanks during fueling and from the tanks to the engine during flight, the hydraulic fluids used to control the aircraft, and so on. It is this type of terminology, entities, their relationships, and operations on them that go into constructing a domain model.

Note that nothing in the above (very brief and informal) domain description belies the fact that a significant portion of a given avionics system's functions, data, and entities

eventually will be reified in software. Remember, the domain model characterizes the problem space. The fundamental objective of a domain model is two-fold:

1. A domain model standardizes the given problem domain's *terminology* and its *semantics*. Together, the terminology and semantics are referred to as the domain's *ontology*.
2. A domain model provides the basis for standardized descriptions of problems to be solved in the domain.

For example, *yaw* is a term used in avionics to refer to an aircraft's rotation about an axis that is perpendicular to the aircraft's horizontal plane (that is, its wings) and goes through the aircraft's center of gravity. We would, of course, also have to define what a center of gravity is, and then any other terms needed to clarify its definition, and so on, until all the terms related to yaw are explained using the appropriate domain-specific terminology. Yaw is produced by moving the aircraft's rudder.

Note that while yaw is a widely known term because of the popularity and ubiquity of flight in today's world, nothing really prevents us from using a different term to describe this same concept in another problem domain. For example, we might call the analogous movement of a flat-panel television screen *left-right rotation*. Similarly, we can use this same term to describe a different concept in another problem domain (although in this case we would risk confusing matters given the broad familiarity with this particular term and its above-defined meaning).

A domain model is a product of context analysis and domain analysis. *Context analysis* is the activity whereby the boundaries of a domain are defined, and the relationships of the entities inside the domain to those outside it are identified and captured. This is an important activity that sometimes remains overlooked, since DSSE focuses primarily on things *within* a domain, rather than the bounds of the domain or the relationship between things inside and outside the domain.

Domain analysis can be defined as the activity of identifying, capturing, and organizing the *domain assets*, that is, the objects, operations, and data recurring across a class of similar systems within an application domain. Domain analysis results in a description of the domain assets using a standardized vocabulary. Its goal is to set the foundation of making the assets usable and reusable when solving new problems within the domain. For example, yaw is a usable asset if it is described in a manner that allows an avionics (software) engineer to find it easily and understand its meaning unambiguously whenever needed. Yaw is reusable if it describes an operation of all, or at least most, aircraft within the domain (for example, large passenger airplanes). Note that this level of reuse, while clearly important to multiple projects conducted by an engineering organization, including its software engineering divisions, has little if anything to do with code.

A domain model will usually comprise several pieces of information that together present a useful picture of the domain assets and their interrelationships. These models can be grouped into four categories:

- Domain dictionary
- Information model
- Feature model
- Operational model

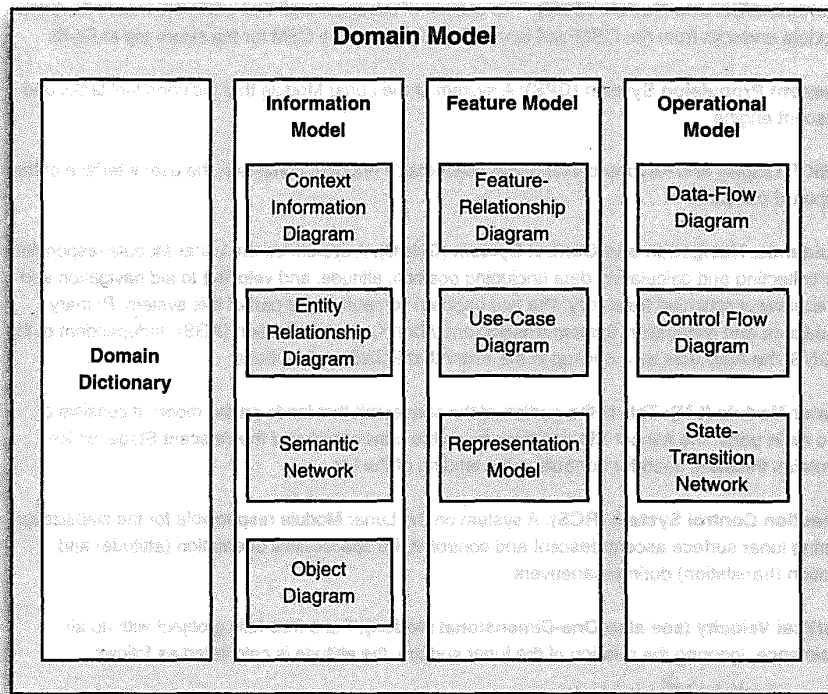


Figure 15-7.
Elements of a
domain model.

The different models and a broad cross-section of their associated diagram types are shown in Figure 15-7. We now discuss each of them in more detail.

Domain Dictionary

A domain dictionary represents the identification and definitions of terms used in the domain model. These terms may be widely used and understood outside the domain, in which case the goal of including them in the domain dictionary may be to define them carefully and avoid any inconsistencies and ambiguities in their usage. On the other hand, the terms may be specialized, or even invented for the sole purpose of fostering human stakeholders' communication and understanding within the domain. The domain dictionary should be updated over time with new terminology and with new or clearer understandings of existing concepts. A domain dictionary also becomes an indispensable tool for training new engineers working within the application domain. Figure 15-8 shows a partial domain dictionary from a hypothetical Lunar Lander DSSA.

Information Model

An information model is actually a collection of multiple models that may be used in different organizations and different DSSAs. The information model is a result of context analysis and information analysis. Context analysis results in defining the boundaries of

Figure 15-8.
Partial domain
dictionary for a
Lunar Lander
DSSA.

Command/Service Module (CSM): The section of the spacecraft that orbits the moon; the Lunar Module undocks from the CSM and lands, redocking with the CSM for the return trip to Earth.

Descent Propulsion System (DPS): A system of the Lunar Module that includes fuel tanks and descent engine.

DSKY: Display and Keyboard Unit of the spacecraft's onboard computer; the user interface of the onboard computer.

Guidance, Navigation and Control System (GN&C): A system on the Lunar Module responsible for collecting and calculating data (including position, altitude, and velocity) to aid navigation and to achieve a specified trajectory. The two onboard computers are part of this system: Primary Guidance and Navigation System (PGNS) and Abort Guidance Section (AGS). Independent of the PGNS, the AGS acts as a backup in the event that PGNS malfunctions.

Lunar Module (LM): This is the portion of the spacecraft that lands on the moon. It consists of two main parts: the Ascent Stage (which holds the crew cabin) and the Descent Stage, which contains thrusters used for controlling the landing of the LM.

Reaction Control System (RCS): A system on the Lunar Module responsible for the stabilization during lunar surface ascent/descent and control of the spacecraft's orientation (attitude) and motion (translation) during maneuvers.

Vertical Velocity (see also One-Dimensional Motion): For a free-falling object with no air resistance, ignoring the rotation of the lunar surface, the altitude is calculated as follows:

$$y = \frac{1}{2} * a * t^2 + v_i * t + y_i$$

where:

y = altitude

a = constant acceleration due to gravity on a lunar body

(see **Acceleration** for sample values)

t = time in seconds

v_i = initial velocity

y_i = initial altitude

When thrust is applied, the following equation is used:

$$y = \frac{1}{2} * (a_{burner} - a_{gravity}) * t^2 + v_i * t + y_i$$

where:

y = altitude

a_{burner} = constant acceleration upward due to thrust

a_{gravity} = constant acceleration due to gravity on a lunar body

(see **Acceleration** for sample values)

t = time in seconds

v_i = initial velocity

y_i = initial altitude

the domain and preserves information that may be otherwise implicit and scattered across many different systems and their artifacts. Information analysis then takes the result of context analysis (that is, the contours of the domain) and represents the intradomain knowledge explicitly in terms of domain entities and their relationships.

The information model ensures that the DSSA employs appropriate data abstractions and decompositions. Different elements of the information model are used by at least three types of stakeholders.

1. Requirements engineers use the information model as an aid in precisely specifying the reference requirements, that is, the requirements common across the applications in the DSSA (see the “Canonical Requirements” section below for a detailed discussion of reference requirements). They also use the information model to relate appropriately the application-specific requirements to the reference requirements.
2. Software architects use the information model to identify and appropriately relate the modules in the software system in the manner that reflects the characteristics of the domain, the interfaces exported by those modules, the data exchanged among them, and the mechanisms by which and constraints under which data is exchanged. This activity results in a reference architecture (see the “Canonical Solutions” section below for a detailed discussion of reference architectures). Software architects also use this information to relate application-specific architectures and designs to the reference architecture.
3. Software system maintainers use the information model to understand the manner in which any given application in the DSSA addresses problems. This enables them to relate properly the system maintenance and evolution requirements to the reference requirements and application-specific requirements, and to relate their proposed realization in the system to the reference architecture and application-specific architecture.

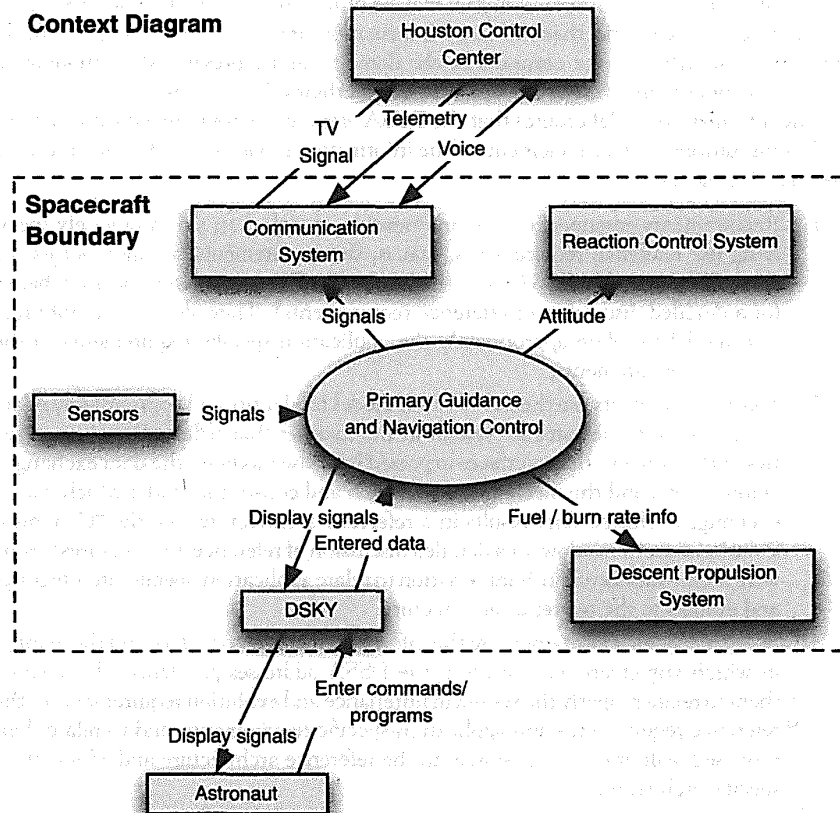
An information model usually consists of one or more of the below types of diagrams.

Context Information Diagram. This diagram captures the high-level data flow between the major entities in the system, their relationship to the entities outside the system (as well as outside the domain), as well as any data that is assumed to come from external sources. For example, in the Lunar Lander DSSA, the information exchanged among the spacecraft’s sensors, actuators, and computers is part of the context information diagram. At the same time, the topology of the moon’s (or planet’s) surface on which the spacecraft is landing is considered to be outside the domain. An example context information diagram from the Lunar Lander DSSA is shown in Figure 15-9.

Entity/Relationship (ER) Diagram. This diagram captures aggregation (“a-part-of”) and generalization (“is-a”) relationships among the entities within the domain. For example, in the Lunar Lander DSSA, fuel level is “a-part-of” the spacecraft entity, while a given thruster “is-a” Lunar Lander actuator. An example ER diagram from the Lunar Lander DSSA is shown in Figure 15-10.

Object Diagram. This diagram identifies the objects in the application domain rather than in the software. In other words, the object diagram describes entities in the problem space;

Figure 15-9.
Example context
information
diagram from the
Lunar Lander
DSSA.



the solution-space components used to realize the problem-space entities will be captured in the reference architecture, discussed below. The object diagram captures the attributes and properties of each object, as well as their interdependencies and interactions with other objects in the domain. An example object diagram from the Lunar Lander DSSA is shown in Figure 15-11.

Feature Model

The feature model is, in fact, a collection of several models. A feature model results from feature analysis. The goal of feature analysis is to capture and organize a customer's or end-user's understandings and expectations of the overall (shared) capabilities of applications in a given domain. The feature model is considered to be the chief means of communication between the customers and the developers of new applications. The feature model explicitly delineates the commonalities and differences among systems in the domain. Examples of features include function descriptions, descriptions of the mission and usage patterns, performance requirements, accuracy, time synchronization, and so on (Kang et al. 1990).

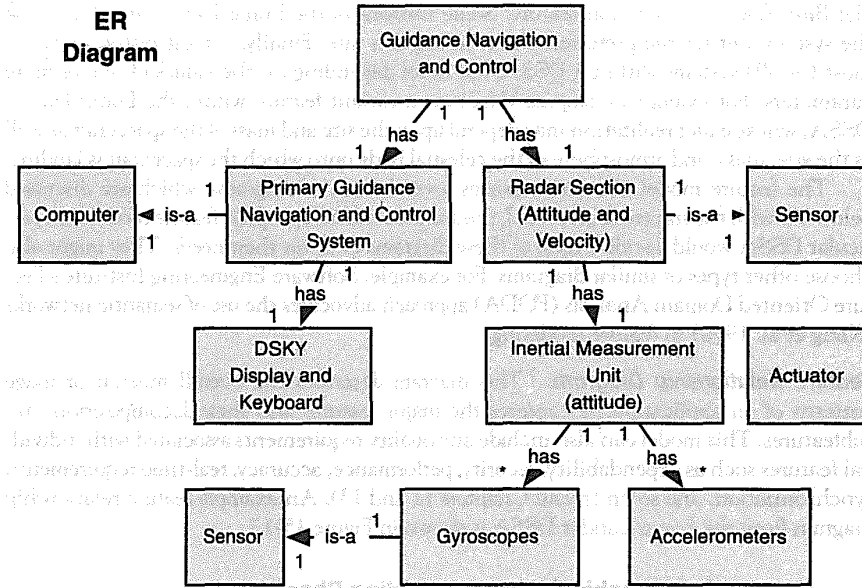


Figure 15-10.
Example ER
diagram from the
Lunar Lander
DSSA.

Such features are meaningful to the users and can assist the engineers in the derivation of DSSA that will provide the desired capabilities.

Features may be defined as mandatory, optional, or variant. Mandatory features are expected to recur across all systems in the domain. In the case of the Lunar Lander system, *Compute Altitude* is a mandatory feature: Any implementation of the Lunar Lander must be able to continuously compute the spacecraft's altitude. Optional features are those that are identified as useful for a subset of the systems within the given DSSA. For example,

Object Diagram

Object	Attributes	Operations
Landing Radar	Attitude Velocity	Sense Altitude Sense Velocity
Descent Engine	Throttle Level Nozzle Direction	Set Throttle Level Change Nozzle Direction (Gimbal) Turn On / Off
Thrust Engines	Firing Mode Attitude Translation	Set Pulse or Continuous Change Altitude Change Translation

Figure 15-11.
Example object
diagram from the
Lunar Lander
DSSA.

Get Burn Rate is an optional feature: Some versions of the Lunar Lander are able to ask the system user for the preferred fuel consumption rate. Finally, variant features exist in most (or all) systems within a DSSA, but differ depending on the values of one or more parameters. For example, *Compute Velocity* is a variant feature within the Lunar Lander DSSA, whose exact realization may depend upon the size and mass of the spacecraft as well as the size, mass, and atmosphere of the celestial body onto which the spacecraft is landing.

The feature model also encompasses several types of diagrams, which are discussed below. As with the information model, the architects developing the feature model for a particular DSSA would decide which of these diagrams best suit their needs. They might also choose other types of similar diagrams. For example, Software Engineering Institute's Feature Oriented Domain Analysis (FODA) approach advocates the use of semantic networks (Kang et al. 1990) in feature modeling.

Feature Relationship Diagram.¹ This diagram describes the overall mission or usage patterns of an application. It captures the major features and their decomposition into subfeatures. This model can also include any quality requirements associated with individual features such as dependability, security, performance, accuracy, real-time requirements, synchronization, and so on (recall Chapters 12 and 13). An example feature relationship diagram from the Lunar Lander DSSA is shown in Figure 15-12.

Figure 15-12.
Example feature
relationship
diagram from the
Lunar Lander
DSSA.

Feature Relationship Diagram—Landing Phase

Mandatory: The Lunar Lander must continually read altitude from the Landing Radar and relay that data to Houston with less than 500 msec of latency. Astronauts must be able to control the descent of the Lunar Lander using manual control on the descent engine. The descent engine must respond to control commands in 250msec, with or without a functioning DSKY...

Optional/Variant: Lunar Lander provides the option to land automatically or allow the crew to manually steer the spacecraft.

Quality Requirements:

Real-time requirements: The thrusters and the descent engine must be able to respond to commands from the computer system in real-time.

Fault tolerance: Lunar Lander must be able to continue in its flight path even when the main computer system (Primary Navigation Guidance & Control) goes down. Lunar Lander must be able to maintain system altitude even when one of the thrusters and propellant supplies goes down in the Reaction Control System.

Lunar Lander's computer system must function properly after a restart (restart protection).

¹The feature model is sometimes also referred to as "context model," and the feature relationship diagram as "context diagram." We deliberately refrain from using "context" in this setting to avoid confusing matters with the context diagram in the information model.

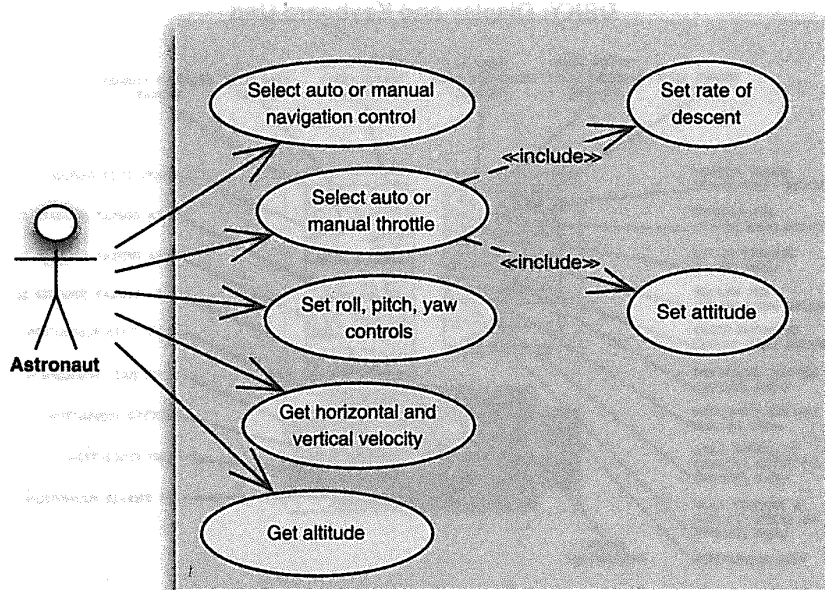


Figure 15-13.
Example use-case
diagram from the
Lunar Lander
DSSA.

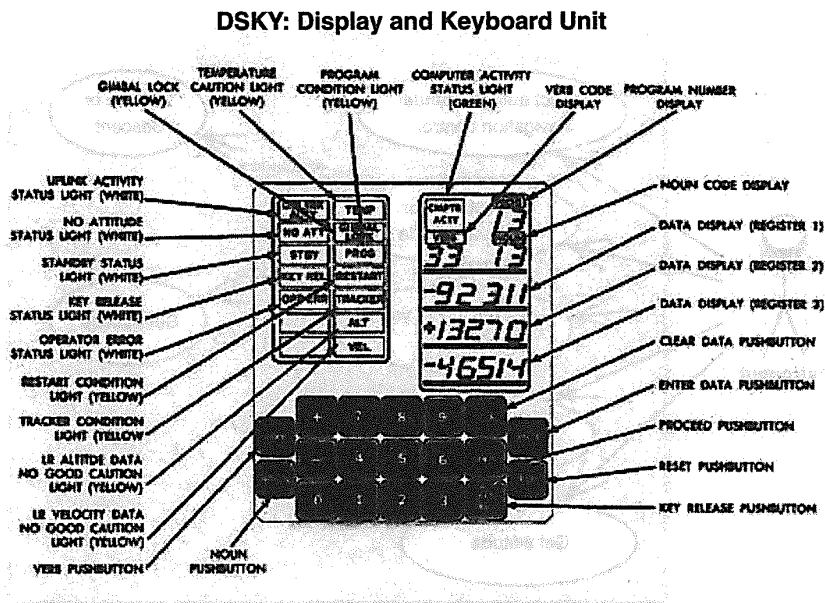
Use-Case Diagram. This diagram captures the usage scenarios in the system. Use cases have become prevalent with the introduction of the Unified Modeling Language (see Chapters 6 and 16). Use-case diagrams are elicited from domain experts, system customers, and system users, and encompass the manner in which a feature is expected to be used, as well as control- and data-flow among multiple features. An example use-case diagram from the Lunar Lander DSSA is shown in Figure 15-13.

Representation Model. This diagram describes how information is made available to a human user. In other words, the representation model details what the appropriate user interfaces are for the systems within the DSSA. This model also captures the information produced for another application: What kinds of input and output capabilities are available, and what is the expected, common format for the exchanged information? An example partial representation model from the Lunar Lander DSSA is shown in Figure 15-14.

Operational Model

The operational model is the foundation upon which the software designer begins the process of understanding (1) how to provide the features for a particular system in a domain, and (2) how to make use of the entities identified in the resulting domain model. The operational model identifies the key operations (that is, functions) that occur within a domain, the data exchanged among those operations, as well as the commonly occurring sequences of those operations. In other words, the operational model represents *how* applications within a domain work.

Figure 15-14.
Example
representation
model from the
Lunar Lander
DSSA. Original
DSKY diagram
from Apollo
Operations
Handbook,
Lunar Module
LM5 and
Subsequent,
Volume 1:
Subsystems Data
(LMA790-3-
LM5).



Issue commands via VERB (action) and NOUN (object)

Three 5-digit registers
(General Purpose)

Three 2-digit registers

(Indicate landing phase)

19 Keys

Warning Lights

- E.g.: VERB 6 NOUN 20

- VERB 6: Display in Decimal

- NOUN 20: Angles

70 pre-defined PROGRAMS

- E.g., PROGRAM for each descent phase:

- P63 (Braking)

- P64 (Approach)

- P65 (Landing)

The operational model is a result of operational analysis, which identifies the commonalities as well as differences in control-flow and data-flow among the entities in a domain. The domain-wide entities identified in the information and feature models, discussed above, form the basis for the operational model: The information model captures the data exchanged by the operations, while the feature model captures the operations themselves. The control- and data-flow of each individual application within the DSSA can be derived from the operational model.

Three representative types of diagrams used within the operational model are discussed below.

Data-Flow Diagram. This diagram focuses explicitly on the data exchanged within the system, with no notion of control. An example data-flow diagram from the Lunar Lander DSSA is shown in Figure 15-15.

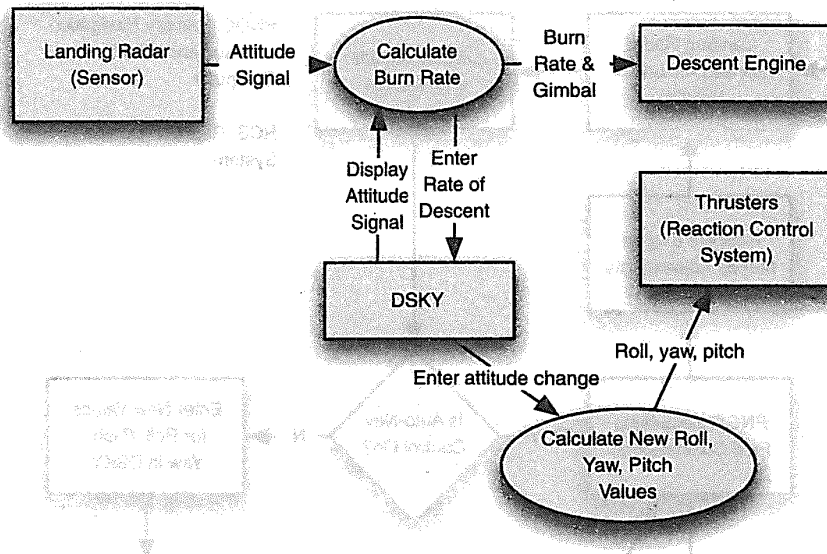
Data-flow Diagram

Figure 15-15.
Example
data-flow
diagram from the
Lunar Lander
DSSA.

Control-Flow Diagrams. This diagram focuses on the exchange of control within the system, without regard for data. An example control-flow diagram from the Lunar Lander DSSA is shown in Figure 15-16.

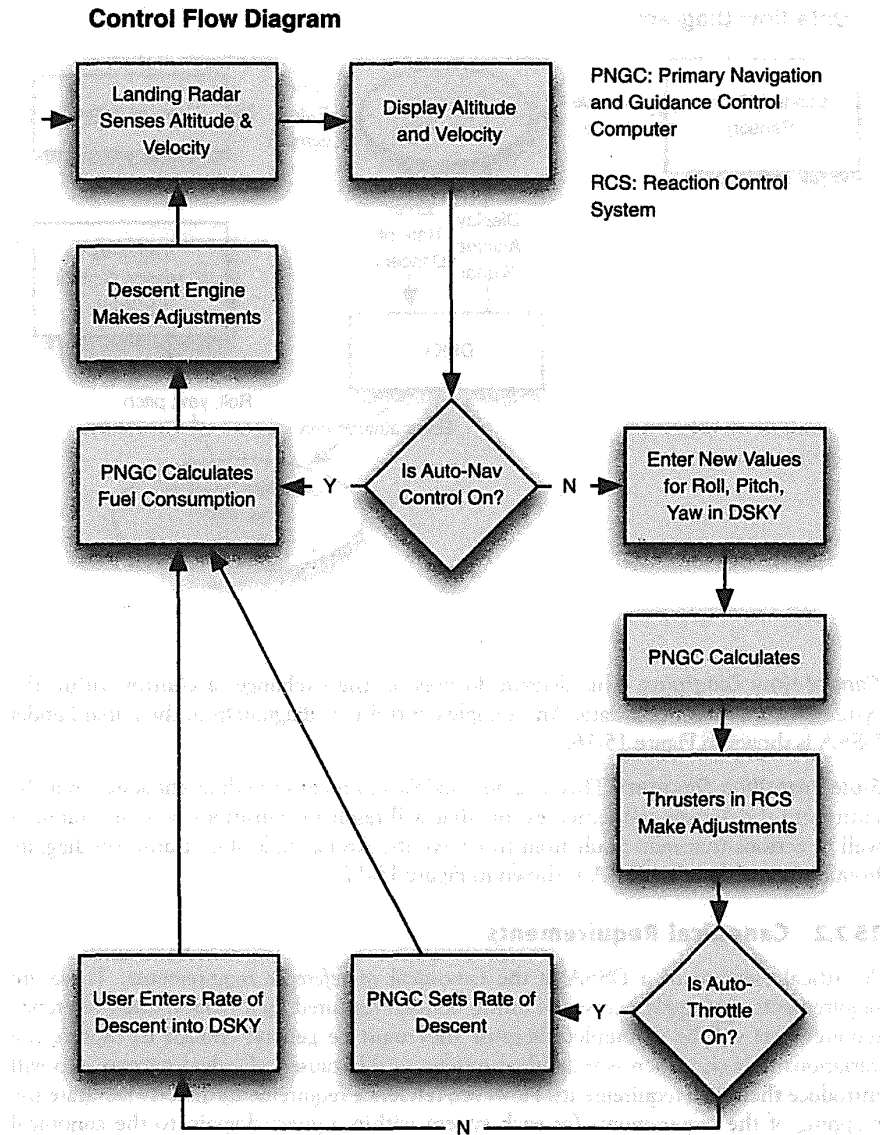
State-Transition Diagram. This diagram models and relates the different states that the entities in the domain will enter, events that will result in transitions between states, as well as actions that may result from those events. An example state-transition diagram from the Lunar Lander DSSA is shown in Figure 15-17.

15.2.2 Canonical Requirements

A critical element of a DSSA is the canonical or *reference* requirements. These are requirements that apply across the entire domain captured by a DSSA. Such reference requirements will be incomplete because they must be general enough to capture the variations that occur across individual systems and because individual systems also will introduce their own requirements. However, reference requirements directly facilitate the mapping of the requirements for each system within a given domain to the canonical domain-specific solution (discussed in the next section).

Similar to the requirements for any software system, the starting point for reference requirements is the customer's statement of needs. In a DSSA, this statement is general enough to apply to any system in the domain. A DSSA is an exercise in generalization from specific experience, so the statement may emerge over time from similar statements that were specific to individual systems constructed previously within the domain. The

Figure 15-16.
Example
control-flow
diagram from the
Lunar Lander
DSSA.



customer needs statement identifies the functional requirements for the DSSA at a high level of abstraction. It is usually informal, ambiguous, and incomplete. It is a starting point for deriving (reference) requirements, but it is not the same as reference requirements. As an example, Figure 15-18 shows a partial customer needs statement for the Lunar Lander DSSA.

State Transition Diagram

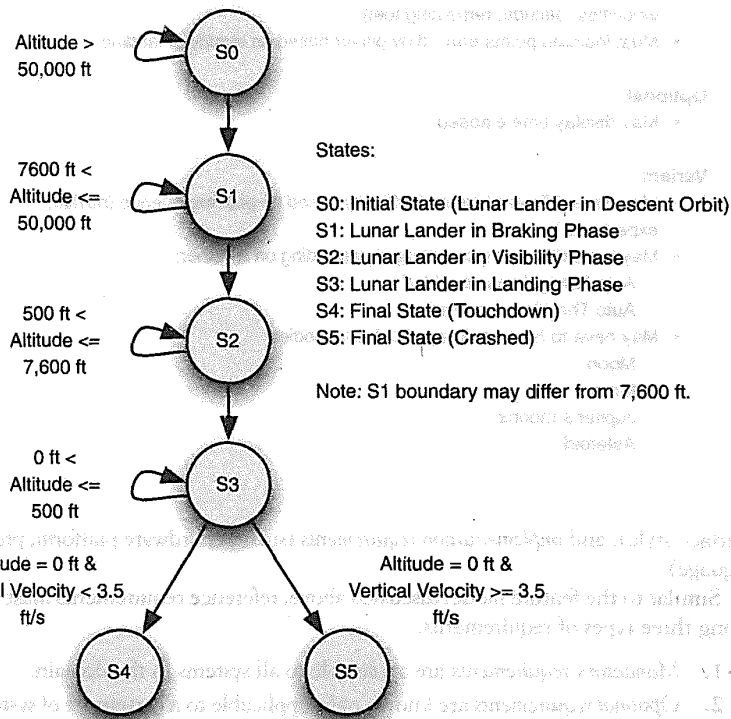


Figure 15-17.
Example state-transition diagram from the Lunar Lander DSSA.

Reference requirements are then derived from the customer needs statement as well as the requirements of the individual legacy systems within the domain. The reference requirements contain the defining characteristics of the problem space. These are the *functional requirements*. Reference requirements also contain limiting characteristics (that is, constraints) in the solution space. These are the *non-functional requirements* (such as security, performance, reliability), *design requirements* (such as architectural style, user

A family of related systems is needed that

1. Supports the Lunar Lander video game for different platforms (e.g., PC, Mac, Pocket PC, cell phone, etc.).
2. Supports different user interfaces, from simple textual to a sophisticated 3-D GUI.
3. Runs on a single machine or is decentralized across a network.
4. Works in a single-user or multi-user competition mode, and supports different variations (e.g., only landing from a given height, first entering orbit and then landing, landing on different types of terrain, landing on different types of celestial bodies, etc.).

Figure 15-18.
Example customer needs statement from the Lunar Lander DSSA.

Figure 15-19.
Example
reference
requirements
from the Lunar
Lander DSSA.

Mandatory

- Must display the current status of the Lunar Lander (horizontal and vertical velocities, altitude, remaining fuel)
- Must indicate points earned by player based on quality of landing

Optional

- May display time elapsed

Variant

- May have different levels of difficulty based on pilot experience (novice, expert, etc.)
- May have different types of input depending on whether:
Auto Navigation is enabled
Auto Throttle is enabled
- May have to land on different celestial bodies:
Moon
Mars
Jupiter's moons
Asteroid

interface style), and *implementation requirements* (such as hardware platform, programming language).

Similar to the feature model discussed above, reference requirements must distinguish among three types of requirements:

1. *Mandatory requirements* are applicable to all systems in the domain.
2. *Optional requirements* are known to be applicable to a certain set of systems within the domain.
3. *Variable requirements* are application-specific, and may be unknown at the time the DSSA, and more specifically reference requirements, are formulated.

In general, each requirement—whether functional, non-functional, design, or implementation—can be of any one of the three types (mandatory, optional, or variable). We provide examples of mandatory, optional, and variable requirements for the Lunar Lander in Figure 15-19.

15.2.3 Canonical Solution Strategies—Reference Architectures

When business goals and technology are applied to a problem within a domain, solutions emerge. A single product architecture represents a single solution. However, as we have covered extensively, when attacking problems in the same domain with similar business goals in mind and leveraging similar technology, similarities will emerge in solution architectures. These similarities can be captured in a *reference architecture*. Here, we repeat the definition found in Chapter 3.

Definition. *Reference architecture* is the set of principal design decisions that are simultaneously applicable to multiple related systems, typically within an application domain, with explicitly defined points of variation.

As sets of principal design decisions, reference architectures are themselves architectures. However, they are generic or variegated such that they are applicable to multiple systems simultaneously—in the context of the terms we have used in this chapter, to a population or a product line of systems. Points where the architecture can vary from family member to family member are explicitly defined as part of the reference architecture.

There are many different kinds of reference architectures; this definition is intentionally broad. They differ in how they express the commonalities and the points of variation in a population or product line. Three classes of reference architectures are:

Complete single-product architecture: An ordinary product architecture of a simple toy example or a complete system for a particular domain can be considered a reference architecture, since it might serve as an exemplar for future products in the same domain. Such reference architectures are relatively weak because they do not provide much engineering guidance as to how to use the reference architecture to create new systems, and they do not explicitly call out variation points.

Incomplete invariant architecture: A partial architecture can be specified that is constant and unchanging among all products. Parts of the architecture that vary from product to product are left unspecified, although some design guidance might be provided as to how to “fill in the blanks” in the architecture.

Invariant architecture with explicit variation: This kind of reference architecture can be used to simultaneously capture both the invariants *and* the variants in products in a domain. For each variation point, permitted alternatives are specified in detail.

Developing a Reference Architecture

Developing a reference architecture is a serious decision because it will be the guiding foundation for a number of future products. Determining when, how, and why to develop a reference architecture is critical to success.

When to Develop a Reference Architecture. An obvious question that arises in DSSE is, “When is the right time to develop a reference architecture?” Choosing an appropriate time is critical because a reference architecture will constrain and bind all solutions in the domain. Premature development of a reference architecture can be disastrous if the architecture is not validated or the stakeholders’ understanding of the domain is too incomplete: Instead of affecting just one product, a mistake made in a too-early reference architecture will propagate to many products and vastly increase the costs of that mistake. On the other hand, developing a reference architecture too late can be equally costly: If many diverse solution architectures have already been developed in a domain, then there will be little opportunity to adapt these existing architectures to fit the reference architecture (they will become “legacy systems”), and therefore substantially limit reuse benefits.

A good tactic that mitigates some risk is to adopt an incremental approach with respect to reference architecture. The reference architecture should still be developed with the entire family of products in mind—if this is not done, it may take on the characteristics of a single product and later have to be significantly adapted to incorporate the full family. If, as is often the case, a number of products have already been developed and they are being adapted to use a reference architecture, it is usually easier to adapt one or two products

at a time to use the reference architecture rather than try to unify the family all at once. Starting with the products whose existing architecture is closest to the target reference architecture is generally a good idea. This allows the reference architecture to be evaluated and refined without affecting too many products simultaneously.

Choosing the Form of a Reference Architecture. Deciding how to capture a reference architecture depends on the domain, the stakeholders, and the underlying business needs of the organization(s) involved. More precise modeling techniques (such as the use of explicit variation points) make it easier to detect and prevent phenomena such as architectural drift and erosion, but might also limit creative freedom and opportunities for innovation.

A complete single product architecture offers the least guidance to architects as to how to elaborate it into a family of products (unless the products in that family have architectures that are very close to the reference architecture). If this form is chosen for the reference architecture, explicit documentation of the underlying style and what is and is not allowed to change should be provided as well. It may also be useful to include several examples of complete product architectures in a family to illustrate this.

Incomplete invariant architectures have the advantage that they tell architects directly what must be present in a family member's architecture. However, they may lack details about how to actually complete the elaboration process. Again, important documentation about the style and how elaboration should proceed is important. Another (somewhat costlier) alternative is to combine this with complete product architectures: that is, as part of the reference architecture, provide both the incomplete invariant architecture *and* one or more elaborated product architectures (even if these are only samples or demonstration architectures) as these can serve as examples for future elaborations.

Invariant architectures with explicit variation points provide the most direction to architects; not only do they say exactly what must be present in every family member's architecture, but they also effectively define all the members of the family. A reference architecture in this form may have a large number of variation points, and the sum of all possible alternatives will far exceed the business needs of the developing organization. For example, imagine such a reference architecture that defines eight optional features that may be included or not included. This reference architecture defines a family of $2^8 = 256$ possible architectures. Not all of these architectures will be feasible (due, perhaps, to feature conflicts) or desirable to produce (due to market conditions). These reference architectures make it easy to select family members, but limit the scope of the products that can be in the family at all.

For a complex domain, a reference architecture will be extremely detailed, with multiple coordinated views, extensive documentation capturing the invariants and variation points, and guidance on how to refine the reference architecture into a concrete application. Rationale for the various decisions embodied in the architecture would also be included. A structural view such as the one shown in Figure 15-20 might be part of such a reference architecture. This depiction alone is not enough to interpret its intent; supporting documentation also is needed. For example, it may be that the components shown are intended to be the only components in the system, or implementers may be advised to add additional components as necessary. Certain points of variation are obvious: whether or not a relay satellite is used, the type of middleware, the implementation of the space link, the other sensors on the lander, and so on. The intended interpretation of all these

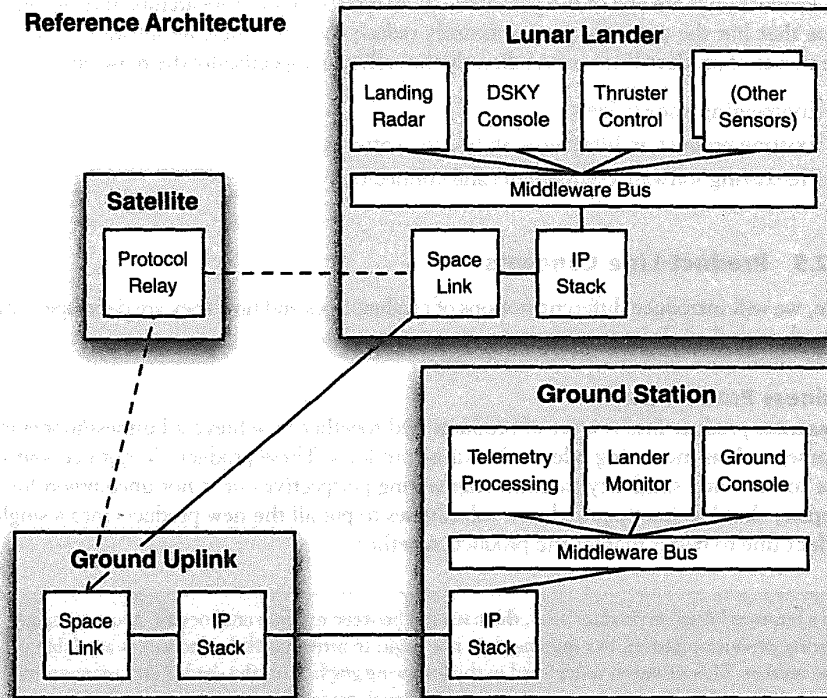
Reference Architecture

Figure 15-20.
Example
reference
architecture
structural view
for a Lunar
Lander system.

elements should be contained in ancillary documentation; this is typical of a reference architecture.

15.2.4 Product Lines and Architecture

There are many significant differences between designing a single-product architecture and a reference architecture; chief among them is that a reference architecture must serve as the basis for many different products simultaneously. This introduces a whole host of new issues: Not only must the architecture suffice for a single solution, but must be developed, visualized, and evaluated in terms of a large number of potential solutions (many of which may never be realized).

Here is where the technologies and techniques developed in the *product-line architecture* community are applicable. Product-line architectures give stakeholders tools with which to diversify an ordinary product architecture into an artifact suitable for describing many similar/related products: a *product line*. The techniques presented here allow stakeholders to develop multi-product architectures without losing the benefits that we have identified in the single-product case: explicit models, visualization, analyzability, and so on.

Product lines are one of the potential “silver bullets” of software architecture—a technique that has the potential to significantly reduce costs and increase software qualities. The power of product lines comes directly through *reuse*, specifically the reuse of:

- Engineering knowledge.
- Existing product architectures, styles, and patterns.
- Preexisting software components and connectors.

15.2.5 Product-Line Concepts

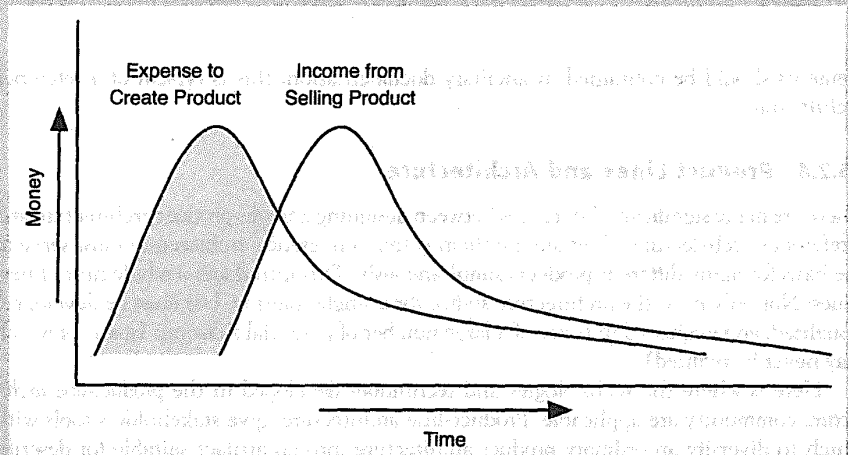
Here, we will introduce different notions of product lines and how they are developed and defined, as well as elements that make up a product line.

Business Product Lines

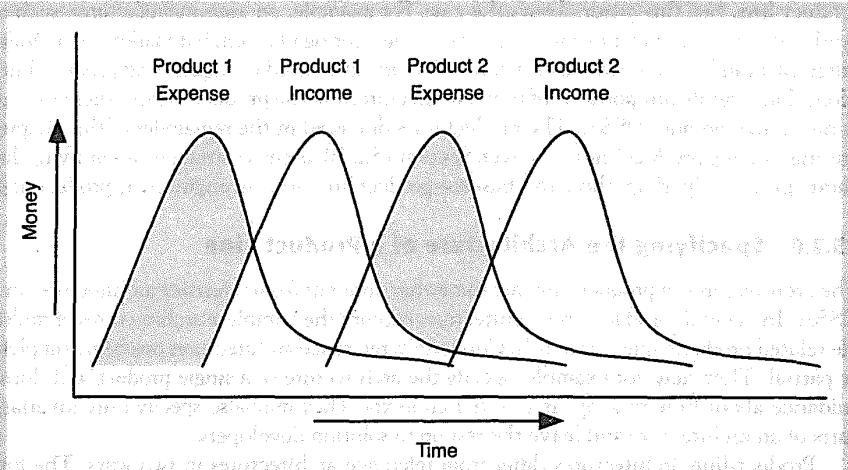
A business product line is a set of products tied together to achieve a business/monetary purpose, such as increasing sales by bundling products. These products do not necessarily have to have any similarity from an engineering perspective—it is not uncommon for a company that has just acquired new subsidiaries to put all the new products into a single product line to better market the products together.

Business
Goes Where
Money
Flows

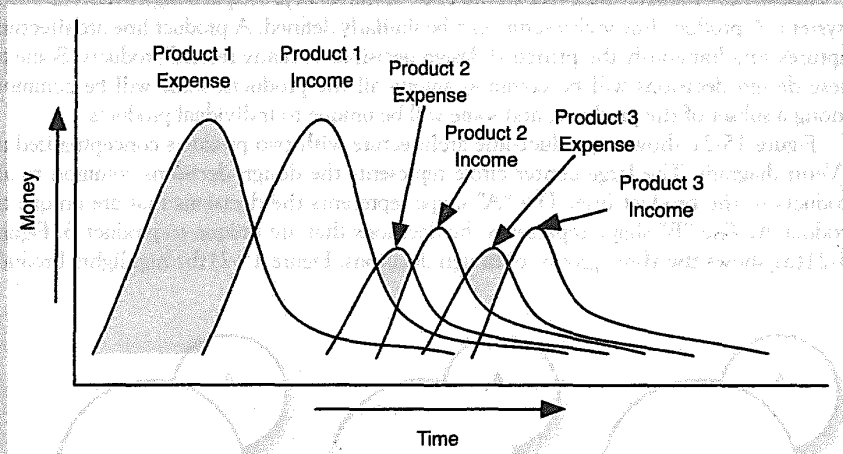
In a business based on product sales, there is a gap between expense and income. Money is spent during product creation, but income does not begin to arrive until the product is available in the market. This situation is depicted in the following graph, with the shaded area representing the initial outlay of capital required for product development.



If traditional development practices are used, additional products will have similar expense/income curves, like so:



The use of product lines, however, lowers the development cost and time for making a new product in the product line. In doing so, additional revenue can come in earlier, reducing the effect of expenses that are incurred, like so:



Note that the shaded areas (times when expenses exceed revenues) are much more limited when product lines are used.

Engineering Product Lines

Products in an engineering product line are tied together by similarities in how they are designed or constructed. Often, a product line will be both a business and engineering

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

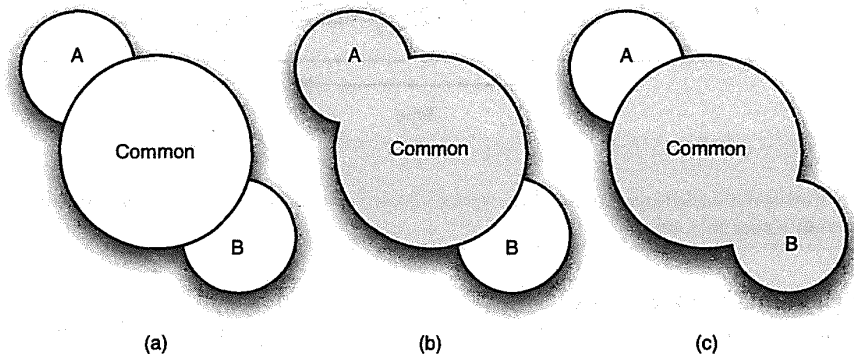
The architectures of product lines are somewhat different from reference architectures in a DSSA. In general, product-line architectures capture the complete architectures of multiple related products simultaneously. Conversely, reference architectures can be incomplete or partial. They may, for example, 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.

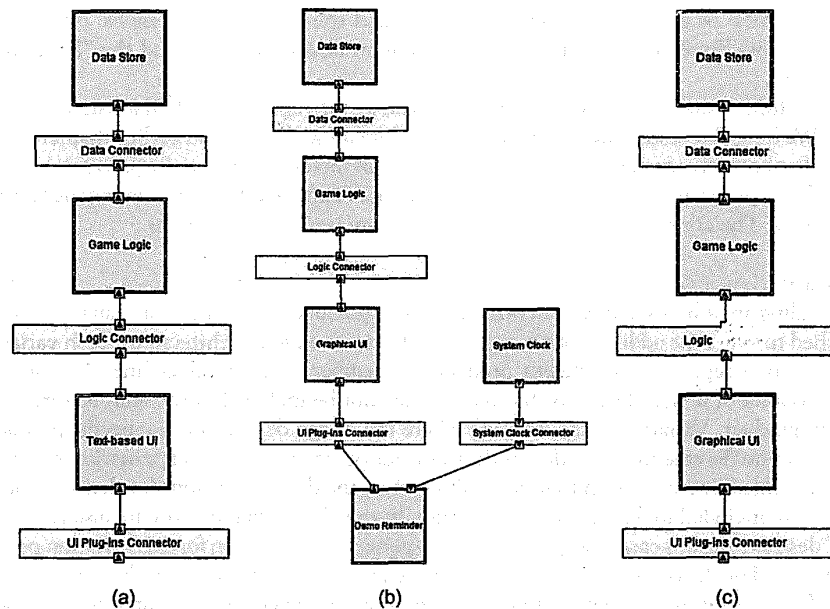
Lunar Lander Lite is intended as a freely distributed Lunar Lander game with a text-based user interface. It resembles the Lunar Lander games discussed in earlier chapters, such as those implemented in Chapter 9. Lunar Lander Pro is intended as a commercially sold Lunar Lander game with a fancier graphical 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	

Components and Connectors	Core Elements	Text UI	Graphical UI	Time Limited
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.

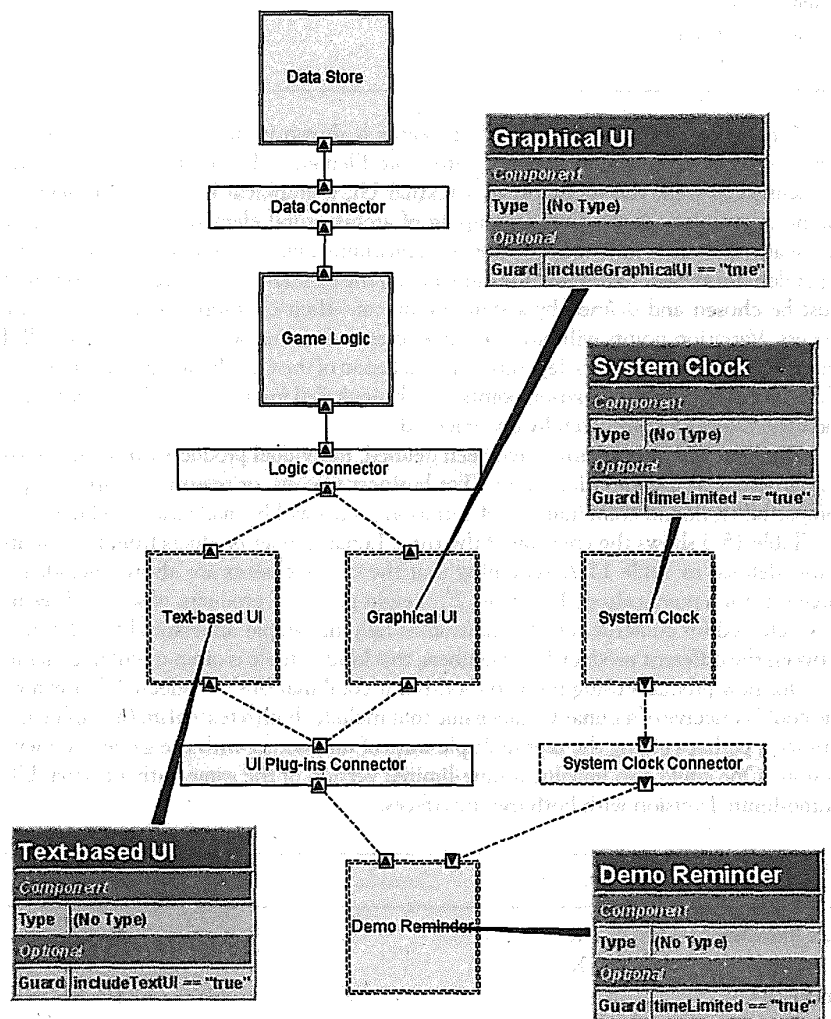
	Lunar Lander Lite	Lunar Lander Demo	Lunar Lander Pro
Core Elements	X	X	X
Text UI	X		
Graphical UI		X	X
Time Limited		X	

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.

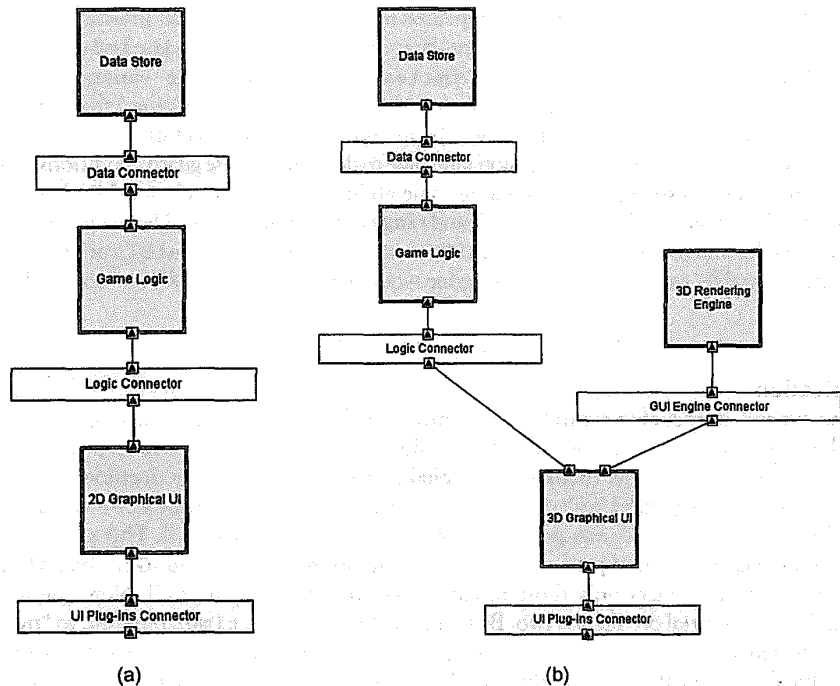
Consider the Lunar Lander product line. By fixing the `includeTextUI` variable to “false,” we have selected only products that do not include a textual UI. This reduces the total number of potential products from 8 to 4. By also fixing “`includeGraphicalUI`” to “true,” we can select only those products 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.

Figure 15-24.
Two versions of a
Lunar Lander
game: (a) version
1, and (b)
version 2.



Consider an evolving Lunar Lander game. The first version of the game might include a graphical user interface with two-dimensional graphics. A second version of the same game might differ by including a graphical user interface with three-dimensional graphics, supported by an off-the-shelf three-dimensional rendering engine. These two product architectures are shown in Figure 15-24(a) and (b), respectively.

Clearly, this situation mirrors the situation seen earlier when the products were different games to be marketed, rather than versions of the same game. Using the same mechanisms described above, the two versions of the game might be encoded into a single product line, with each product representing one version of the game. This product line is shown in Figure 15-25. Here, the version is selected using a single variable, *version*, rather than by choosing a combination of variation points.

15.2.8 Using Product Lines as Tools for What-If Analysis

Product lines can be used to capture variations across related products and across different versions of the same product. Another application of product lines is to capture design alternatives. Design is often an exploratory process—it involves making decisions and trade-offs that will affect the properties and construction of the target products. Questions arise: Should we include this component or that component? Should this part of the application run on the client or on the server? Often, it will be unclear which path is best, so multiple alternatives are considered and pursued until more information becomes available.

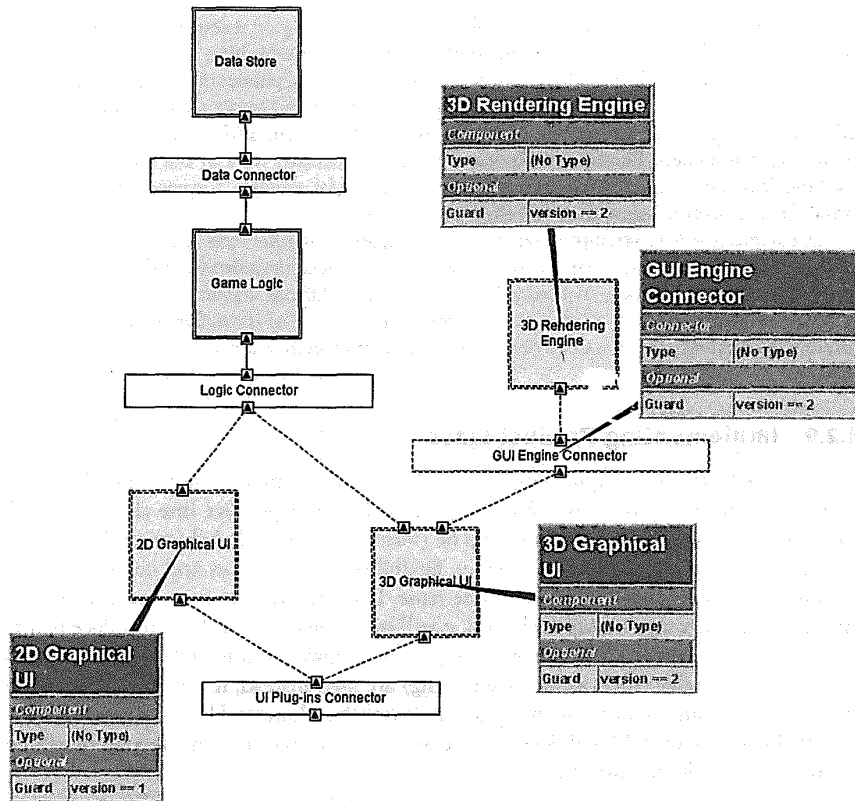


Figure 15-25.
Product line of
Lunar Lander
games where each
product is a
version of Lunar
Lander.

When this happens, product-line techniques can be used to simultaneously capture alternative architectures. Here, each product is not a system that is intended to be built, but one of many potential product designs. Points of variation in these product lines are decision points. The selection process can be used as a form of “what-if” analysis.

Software engineers have had to deal with variations, evolving artifacts, forking, branching, merging, and related concepts for years, particularly in the area of source code. In general, they are assisted in these regards by software configuration management systems. Configuration management systems can also be used to help address these concerns when dealing with artifacts such as architectural models, especially when the artifacts themselves offer no explicit support for these concepts.

One simple approach is simply to put architecture models into a version control system such as RCS (Tichy 1985), CVS (Fogel 1999), Subversion (Collins-Sussman, Fitzpatrick, and Pilato 2004), or ClearCase (Atria Software 1992). This is a viable option for versioning architectures as a whole, and it helps track the evolution of architectures over time. To some extent, alternatives can be created through branching revisions.

Configuration Management of Architectures

This approach does not work particularly well for creating or maintaining product-line architectures. Managing a product-line architecture means establishing a core set of design decisions and then combining those core decisions with variation points consisting of additional decisions. In general, design decisions are spread throughout and across architectural models, and this is why architecture description languages such as Koala and xADL embed the variation points within models. Because version control systems generally work at the level of files, it is difficult to use them to manage elements within files—for example, a component specified inside an architectural model.

A compromise is to attempt to separate out core parts of architectures and variation points into separate files, and then manage the evolution of these elements independently in a version-control system. This works in theory, but in reality it is difficult and inconvenient to have related design decisions for a single product-line member scattered across files, especially when automated merging tools are not available to merge them together again.

15.2.9 Implementing Product Lines

The major reductions in implementation costs from using a product line come about primarily through extensive reuse: Where two products in a product line share a similar architecture, the implementations should be (virtually) identical. Achieving this level of reuse must start in the architecture. Here, flexibility in connectors and communication methods is a primary driver of product-line reuse. If an architecture is chosen where components and connectors are tightly bound, it will be extremely difficult to introduce points of variation without recoding existing components. However, if flexible connectors and communication styles (such as message passing) are used instead, it is easier to introduce variations in the architecture without making extensive changes. However, having flexibility at the architectural level does not necessarily mean that the same flexibility will be present at the implementation level.

Chapter 9 discussed how implementing a system built using software architecture techniques is largely a mapping problem: understanding and controlling how design decisions made in the architecture map to implementation artifacts. In general, all the advice in Chapter 9 applies here as well. Implementing a product-line architecture is still a mapping problem, except that multiple products composed of core elements and variation points are involved. In this case, separating concerns in the implementation artifacts along the boundaries of variation points in the product line is critical.

Consider the situation presented in Figure 15-26. In Figure 15-26(a), good separation of concerns has not been maintained in the implementation. Various source files (File1.java, File2.java, and File3.java) are responsible for implementing both core and product-specific concerns. In this case, neither Product A nor Product B can be constructed without importing at least part of the implementation of the other. This makes it hard to evolve the products independently, and to integrate new products into the product line. Figure 15-26(b) shows a good separation of concerns: Here, implementation artifacts corresponding to products are kept separate from the common core and from each other. Both products can be built without using artifacts that (partially) implement other products.

One way of maintaining this separation of concerns is to use flexible architecture implementation frameworks, as discussed in Chapter 9. Many implementation frameworks help to enforce loose coupling between component and connector implementations, such

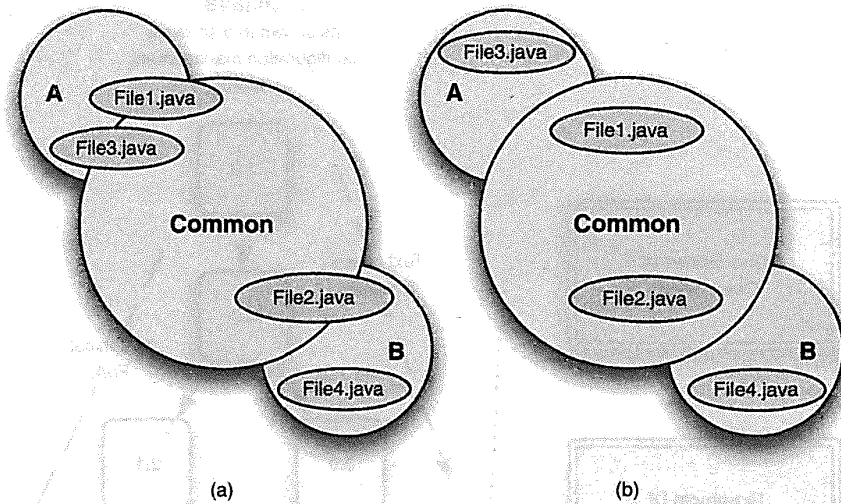


Figure 15-26.
Maintaining
separation of
concerns when
implementing a
product-line
architecture: (a)
bad and (b) good.

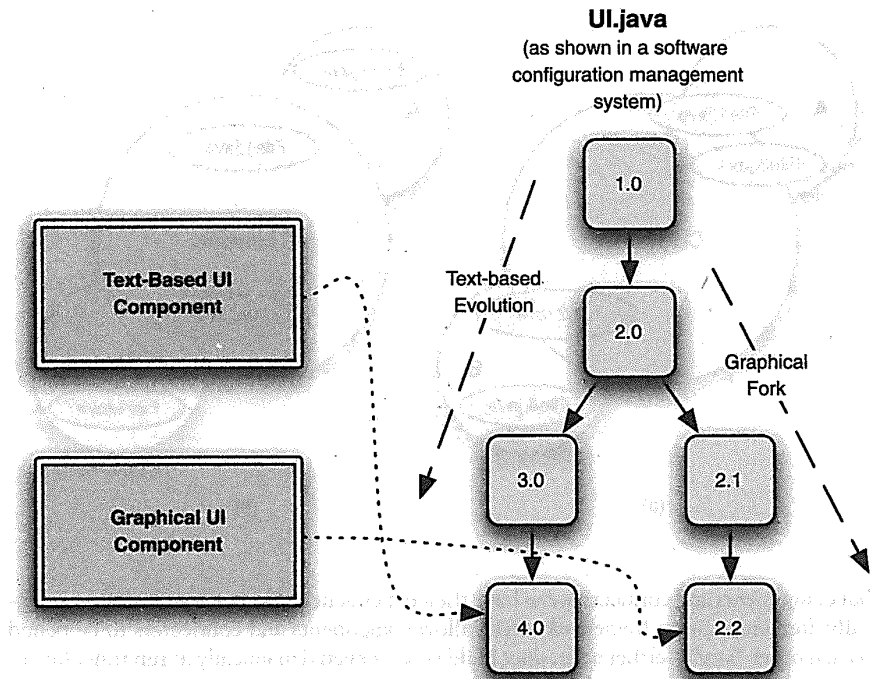
that components and connectors can have their dependencies changed easily and automatically. Implementation frameworks often allow components and connectors to be bound to each other “late”—either at product build time or even dynamically at run time. Implementations that employ late bindings of components generally enforce better separation and independence of components and help to drive reuse. Interface-implementation separation is also important: Well-defined implementation-level interfaces help to establish the services that are provided and required by a component (or set of components) without requiring a particular implementation. This separation of concerns further increases flexibility: Changes to the implementation of one component will not require changes to other components if the contract of the interface is maintained.

This is not always easy to achieve. Often, legacy products are incorporated into a product line. These products may not use flexible components, connectors, and communication methods. In this case, a careful refactoring should be considered, driven by the product-line architecture. Tight bindings in a product line are workable as long as they do not interfere with or cross the points of variation. For example, if three core components that are part of every product in the product line are tightly coupled, this is not a major problem: These components never will be separated, so making their bindings to each other flexible will be of limited near-term value. Instead, developers and architects should focus on making the bindings to components that exist only in a few products more flexible.

Product Lines and Source-Code Management Systems

Any mature development project will leverage some form of configuration management tool set to keep track of different versions of lower-level implementation artifacts: code files, resource files, and so on. Successfully managing product-line implementations requires an understanding of the relationship between versions of architecture-level elements (components, connectors) and versions of implementation-level elements (code, resource files)

Figure 15-27.
Dealing with
versions and
development
branches in a
configuration
management
repository.



as stored in a configuration management repository. Implementations that are cleanly separated along the lines of architectural elements greatly simplify the mapping problem.

One phenomenon that occurs more often in product-line development than in single-product development is the use of multiple versions of the same implementation artifact (for example, source file) in different places in the product line architecture. For example, recall the Lunar Lander product line shown in Figure 15-23. Perhaps, for whatever reason, the implementation of the text-based user interface and the graphical user interface were based on the same code, a single file called *UI.java*, that has evolved over time. At some point in the past, a fork of *UI.java* was created in the configuration management repository, without renaming the file. Thus, there are two development branches of *UI.java*: one for a textual UI and one for a graphical UI.

This situation is depicted in Figure 15-27. Here, the *Text-Based UI* component uses version 4.0 of *UI.java*, while the *Graphical UI* component uses version 2.2. In a situation like this, implementation mappings must not only refer to individual files, but also individual versions of individual files.

15.2.10 Unifying Product Architectures with Different Intellectual Heritage

Unfortunately, product lines are not always considered at the inception of a group of related products. The idea of using product lines often appears on an organization's radar only after

it has developed many products that have similar functionality and has duplicated an enormous amount of effort in doing so. Alternatively, company mergers and acquisitions can leave a single organization with the responsibility for developing, maintaining, and evolving very similar (and at one time, competing) products. At this time, the use of product lines becomes extremely attractive as a way of cutting costs by reusing engineering knowledge and architectural decisions from one product to the next. The difficulty is that the existing products in the product line often share little intellectual heritage, and the product architectures differ widely.

This is one of the most difficult problems in product-line development. The primary tension is that the goal of using a product line is to enable reuse, but the existing products exhibit little to no reuse. When this is the case, attempting to develop a product-line architecture that includes all the existing products may be of limited value, except as a historical artifact. Instead, processes should be forward-looking, attempting to extract the architectural lessons of previous projects and unify them into a product line for future products (or future product versions).

Before a product-line architecture can be developed, individual product architectures must be obtained. Architecturally savvy organizations already will have documented architectures for their products, but, as we have discussed, many product architectures evolve organically or diverge from their original intentions. In this case, architectural recovery (discussed in Chapters 3 and 4) is the first step toward product-line development.

The next step of the process is to normalize the existing product architectures (if necessary). Often, even when architectures for each product are available, they may not capture the same aspects of the systems under development, or capture those aspects at the same level of detail. They may not use the same terminology to refer to similar concepts. When this is the case (and it nearly always is) some amount of normalization must occur—some of the architectures must be adapted so that the commonalities and differences among the product architectures can be more readily identified. Luckily, architecture elaboration in this situation is made easier by the fact that there are actual, implemented products from which to extract the additional necessary information.

When normalized product architectures are available, they can be compared to identify common aspects and points of variation. Unless all of the architectures are expressed in the same notation at the same level of detail, it is probably not possible to automate much of this process. Instead, it is more useful to gather the architects and other stakeholders from each product into a series of meetings. Developing tables similar to Table 15-1 and Table 15-2 in order to understand the overlap between features and elements in the various products may be a useful exercise.

Developing a product-line architecture from this point forward is then a stakeholder-centric activity. There is no one right way to go about this, and depending on the domain and the organizational goals, the effort may go in one of many different directions:

No Product Line: It may be determined that, after examining the architectures, less reuse is possible than previously imagined. Perhaps the products were thought to perform many similar functions, but in reality they do not. Alternatively, the products may be so architecturally different that attempting to unify future versions of the products into a product line would cost more than the potential reuse would save. Here, the best alternative might be to abandon the idea of developing a product line

entirely, and focus on further distinguishing the products from one another in separate development efforts.

One Master Product: Often, the easiest course of action in developing a new product line is to identify the best existing product and use its architecture as a basis for a product line. In this way at least one product will fit naturally into the emergent product line, and innovative features from other products can be integrated as appropriate.

Hybrid: When no one product can be identified as the master for the purpose of developing a product line, the “best of breed” features of each product in the domain can be extracted and unified into a hybrid product line. Here, no one existing product will fit totally naturally into the product-line architecture. This is most useful when new products are not intended to be direct descendants of old products or when existing products are being unified.

15.2.11 Organizational Issues in Creating and Managing Product Lines

This chapter has largely covered the creation and management of product lines and product-line architectures from a technical perspective. However, using product lines is not a solely technical activity. A product-line-based approach needs to be as much organizational and cultural as it is technical. Creating a product line often means integrating the needs and opinions of stakeholders from fundamentally different perspectives into the same effort. This can foster all kinds of organizational and technical issues: friction between the teams, feature creep, difficulty prioritizing customers, least-common-denominator architectures, and so on.

Furthermore, an organization’s investment strategy in its products needs to change to match a product-line approach. Creating a product line incurs substantial costs that do eventually pay off, but not in the context of a single product. Rather, they are recovered through lower maintenance expenses across multiple products, the increased profits from selling more product variants, and the lower costs of developing future versions of the products. This requires organizations to think *horizontally*, across product and team boundaries, which often requires both an organizational and cultural shift.

Paul Clements and Linda Northrop have written extensively about these issues in their book *Software Product Lines: Practices and Patterns* (Clements and Northrop 2001); we encourage those interested to look to this and other sources for more detailed guidance in this regard.

15.3 DSSAS, PRODUCT LINES, AND ARCHITECTURAL STYLES

We have discussed domain-specific software architectures (DSSAs) and product lines, as well as their relationship to each other. A natural question arises at this point: What are the relationships among DSSAs, product lines, and architectural styles?

Recall that architectural styles represent a reusable package of design decisions that can be applied to different products to elicit desired qualities. In some sense, both DSSAs and product-line architectures can be seen in the same light. Usually, however, architectural styles are far more general than DSSAs or product lines. Most architectural styles can be applied across many domains. For example, the pipe-and-filter style has been used to develop

text-processing applications, typesetting systems, and program compilers. Typically, such diverse applications would not all be members of the same DSSA or product line.

Styles do, however, play an important role in the development of both DSSAs and product lines. It would be unusual for different products in the same DSSA or product line to have significantly different architectural styles. As noted above, one of the most common canonical requirements in the development of a DSSA is the architectural style that will be used to create applications within the DSSA. Typically, a reference architecture for a DSSA will follow this prescribed style. Similarly, individual products in the same product-line architecture will generally follow the same style. The use of a common style across a DSSA or product line is ultimately an effort to ensure consistent qualities in the products that emerge and to keep them from drifting away from the original intent of the product line.

Just as there can be points of variation in architectures, there can also be points of variation in architectural styles themselves: different sets of constraints to solve related problems. By applying product-line architecture techniques to the decisions that compose architectural styles rather than architectures, it is possible to create a family of architectural styles.

Recall two of the most basic architectural styles from Chapter 4: batch-sequential and pipe-and-filter, as shown in Figure 15-28. These styles actually have many constraints in common. For example, both enforce a linear ordering of components, each of which has one input and one output, and data moves unidirectionally through the line, being processed individually by each component. They differ in a few details: batch-sequential systems pass the data wholesale in a chunk, and only one component operates on the data at a time; pipe-and-filter systems pass data as it is available and components execute in

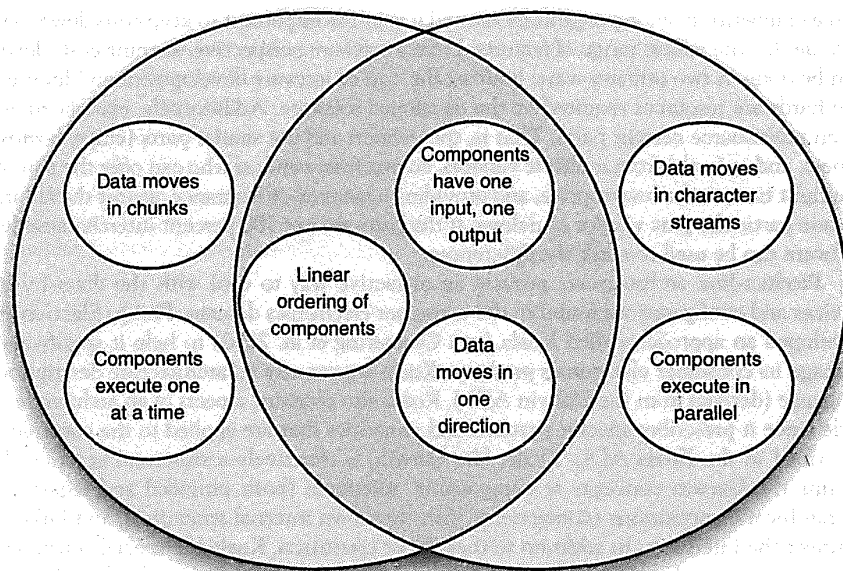


Figure 15-28.
An example of
related
architectural
styles.

parallel when possible. Here, explicit variation points can be used to express and reason about the relationship between architectural styles just as they are used to reason about the relationship among software products.

15.4 DSSE EXAMPLES

We now present two examples of how domain-specific software engineering techniques, including product-line architectures, have been applied in software development practice. The first example is Philips Electronics's use of the Koala architecture description language in the development of software for devices in the consumer electronics domain. The second example is the domain of software-defined radios, configurable devices that communicate using radio waves, with their internal functionality implemented by software.

15.4.1 Koala and Consumer Electronics

Consumer electronics is a dynamic and highly competitive domain of product development. For decades, devices such as televisions and cable descramblers were relatively simple with a few, well-defined capabilities. Over the years, they have become more and more complex, largely due to enhancements in their embedded software. The latest incarnations of these devices include graphical, menu-driven configuration, on-screen programming guides, video on demand, and digital video recording and playback. In a global marketplace, each of these devices must be deployed in multiple regions around the world, specifically configured for the languages and broadcast standards used in those regions.

The increasing feature counts of consumer electronic devices are accompanied by fierce competition among organizations, and it is just as important to keep costs down as it is to deploy the widest range of features. From a software perspective, keeping costs down can be done in two primary ways: limiting the cost of software development and limiting the hardware resources required by the developed software. Additionally, manufacturers often multisource certain parts. That is, they obtain and use similar parts (chips, boards, tuners, and so forth) from multiple vendors, buying from vendors who can offer the part at the right time or the lowest price, and providing a measure of insurance against the failure by one particular part vendor to deliver. If the parts are not 100 percent interchangeable, software can be used to mask the differences.

Product-line architectures provide an attractive way to deal with the diversity of devices and configurations found in the consumer electronics domain. Philips Electronics developed an approach called Koala (van Ommering et al. 2000) to help it specify and manage its consumer electronics products. Koala is primarily an architecture description language (derived from the Darwin ADL). Koala also contains aspects of an architectural style since it prescribes specific patterns and semantics that are applied to the constructs described in the Koala ADL. Koala, like Darwin, is effectively a structural notation: It retains the Darwin concepts of components, interfaces (both provided and required), hierarchical compositions (components with their own internal structures), and links to connect the interfaces. In addition to these basic constructs, Koala has special constructs for supporting product-line variability. Koala is also tightly bound to implementations of embedded components: Certain aspects of Koala (such as the method by which it connects

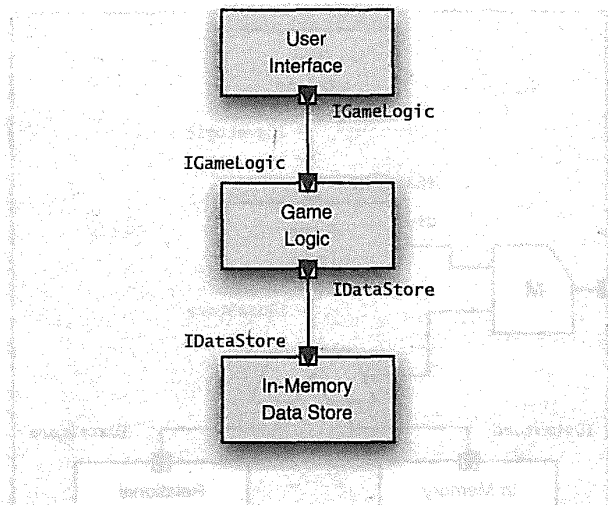


Figure 15-29.
Basic Koala
diagram of Lunar
Lander.

required and provided interfaces in code) are specifically designed with implementation strategies (such as static binding through C macros) in mind.

A basic Koala diagram with no product line variability for a Lunar Lander application is shown in Figure 15-29. As you can see, basic Koala diagrams are effectively equivalent to Darwin diagrams. Koala extends Darwin in the following ways:

IDL-based interface types: An interface type in Koala is a named set of function signatures, similar to those you would find in C. For example, the interface to the Lunar Lander data store in Koala might be declared like this:

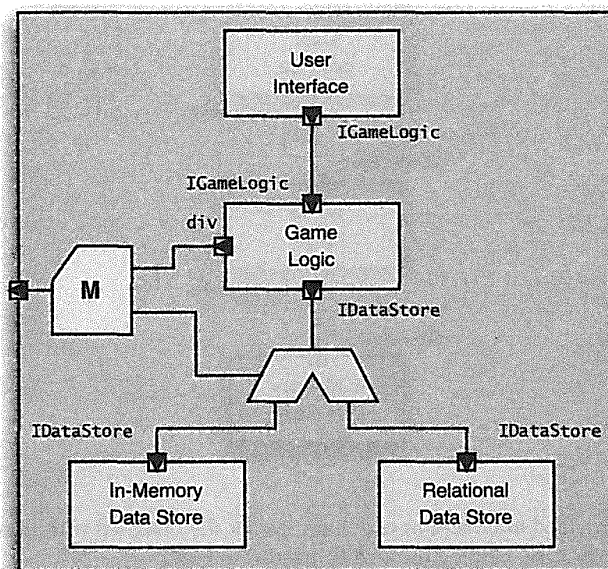
```

interface IDataStore{
    void setAltitude(int altitudeInMeters);
    int getAltitude();
    void setBurnRate(int newBurnRate);
    int getBurnRate();
    ...
}
  
```

The `IDataStore` interface type may be provided (or required) by any number of Koala components. When a provided and required interface are connected, the provided interface type must provide at least the functions required by the required interface type.

Diversity interfaces: One of the philosophies of Koala is that configuration parameters for a component should not be stored in the component; instead, configuration parameters should be accessed by the component from an external source when needed. This allows the application to be configured centrally, from a single component (or set of components) whose purpose is to provide configuration data for the application. Diversity interfaces are special required interfaces that are attached to components and are used by the component to get configuration parameters.

Figure 15-30.
Koala diagram of
Lunar Lander
with a variation
point.



Switches: A switch is a new architectural construct that represents a variation point. It allows a required interface to be connected to multiple different provided interfaces. When the variation point is resolved, only one of the connections will actually be present. A configuration condition determines which provided-required interface pair is connected. These configuration conditions are analogous to the guards discussed earlier in this chapter. A switch is connected to a diversity interface to get its configuration parameters, just as a component would. Depending on the values returned through the diversity interface, the switch will route calls to one of the required interfaces connected to it. This may result in disconnected components (that is, components that will never be involved). Disconnected components are not instantiated to save resources.

Figure 15-30 shows a Lunar Lander architecture in Koala with a variation point. Here, the application can use an in-memory data store or a relational data store to maintain application state. The required interface of the *Game Logic* component is now routed through a switch, which is configured by configuration parameters provided externally. The *Game Logic* component includes a new diversity interface *div* that allows it to access those external configuration parameters.

Optional Required Interfaces

Several components may provide similar, but not identical, services. For example, a basic TV tuner component may have only the ability to change frequencies, but an advanced TV tuner may be able to search for valid frequencies as well. It is possible for callers to a TV tuner to include an optional required interface and query whether the interface is actually

connected or not. If it is connected, the caller can make calls on the optional interface; if not, the caller should behave/degrade gracefully.

Through these mechanisms, Koala gives architects the power to specify and implement product lines of embedded systems. Koala's main strengths are its straightforward notation, intuitive graphical visualization, and tool support for implementing product lines efficiently. Although developed for the consumer electronics domain, Koala is generic enough for use in many domains.

A drawback of Koala is its lack of support for explicit software connectors (a trait it inherits from Darwin). Koala implies that communication among components happens exclusively through synchronous request-response procedure calls. There is no explicit support for asynchronous or message-passing interaction (message passing, in particular, is a well-known mechanism for supporting flexibility and diversity). However, in the target domain (embedded systems in consumer electronics devices), synchronous procedure calls are a very typical mode of interaction.

Using an architecture-centric approach gave Philips the ability to break up its software systems into reusable components that communicate through explicit interfaces, while still retaining the small size and efficiency necessary for embedded systems development. Taking inspiration from an existing architecture description language (Darwin), Philips defined a domain-specific ADL (Koala) with semantics that mapped well to its application domain. Additionally, Philips incorporated explicit support for variability into Koala. This gave it the ability to deal with the many market-induced variations in its product lines (to deal with multiple grades of products with different feature combinations, different requirements due to internationalization, and so on). Rather than maintaining separate architectural descriptions for each product, entire product lines can be captured in a single model. More than 100 engineers at Philips have used Koala, making it one of the most successful industrial applications of ADLs and domain-specific architecture-centric development.

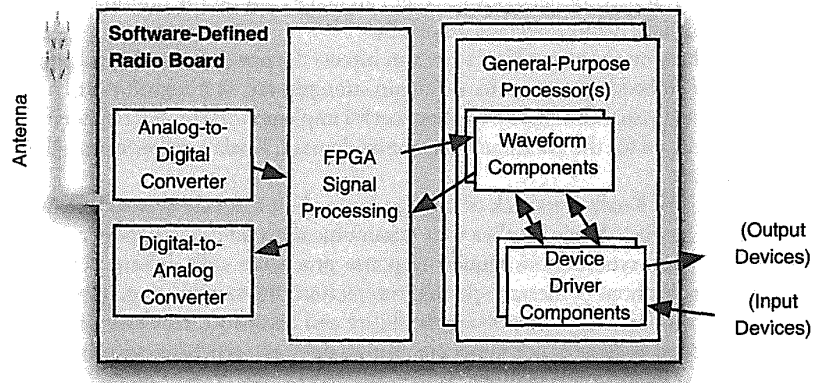
15.4.2 Software-Defined Radios

Communication over the air has been used to facilitate all kinds of social advances, from early audio and Morse-code transmissions to video transmissions for television, to the latest advances in cellular phones and broadband WiFi connections. Traditionally, radio hardware has been specialized for a particular kind of application: For example, an ordinary car stereo is limited to receiving analog AM/FM audio transmissions. It cannot, for example, decode TV signals (even the audio portion) or act as an Internet WiFi repeater. To do so would require hardware components to be changed out or adapted.

The increased capabilities of digital signal processing components as well as advances in reconfigurable hardware (FPGAs³ in particular) have made it possible to rethink how communication-over-radio devices are built from both a hardware and software perspective. The result has been an effort to develop so-called software-defined radios (SDRs) (Dillinger, Madani, and Alonistioti 2003). A software-defined radio is a device that can transmit,

³FPGA stands for Field Programmable Gate Array. An FPGA is effectively an integrated circuit consisting of an array of logic gates that can be reconfigured through software. Once reconfigured, the circuit can perform any function it is programmed for—processing data, cracking encryption, emulating a microprocessor, and so on. In general, FPGAs are slower and larger than special-purpose hardware, but the performance is reasonably close.

Figure 15-31.
Basic data
pipeline of a
software-defined
radio.



receive, and process signals over the air for a variety of applications, where the application is defined by the software that is loaded on the radio. Standard components of a software-defined radio include an antenna, analog-to-digital conversion, FPGAs for high-frequency digital signal processing, and general-purpose processors (such as PowerPCs) for further signal processing duties, as well as various I/O ports for interfacing with audio and video devices (displays, microphones, speakers, and so on).

The basic data flow through a hypothetical software-defined radio board is shown in Figure 15-31. Depending on the configuration of the FPGA(s) and the components and connectors loaded on the general-purpose processors, the radio can be used for all sorts of applications. In theory, a software-defined radio could act as everything from an ordinary FM radio to a television set to a wireless access point simply by loading different software onto the radio.

The domain of software-defined radios is ideal for the application of DSSE principles. A domain-specific software architecture has been developed for software-defined radios called the Software Communications Architecture (Modular Software-programmable Radio Consortium 2001); see the accompanying feature titled, "The Software Communications Architecture (SCA)." This particular DSSA is not a panacea, however, and the sidebar touches on many issues that may make it less than ideal.

The Software Communica- tions Architecture (SCA)

The organizations interested in developing software-defined radios are keenly interested in the design, particularly the software design, of such radios. This resulted in the development of a standard known as the Software Communications Architecture (SCA), which governs the design of SDRs (although, as the name implies, it is intended for future application to other kinds of communications systems).

The SCA is an interesting document to study, since it is the standard architecture that many organizations implementing SDRs are intended to follow.

The SCA looks at the software architecture of a software-defined radio as a set of interacting software components. These components are defined and interact via CORBA (see Chapter 4). As one might expect, the choice of CORBA as the sole software interconnection technique induces several architectural consequences on SCA systems, among them:

- The style of an SCA-based software-defined radio is a distributed objects style.
- Interfaces are a primary driver of interaction.

- Provided interfaces for various kinds of components are defined using CORBA IDL.
- Components communicate primarily through synchronous procedure calls to provided interfaces.

The bulk of the SCA specification is a collection of CORBA IDL interface definitions for various kinds of components. For example, filesystem components implement an interface very similar to the POSIX filesystem interface. Other interface definitions exist for different kinds of drivers and other service components.

Actual signal data in the architecture is broken up into digital data packets. Packets of data are passed from one component to another through a de facto call named `PushPacket`. A component that is to send a packet of data to another component will call `PushPacket` and pass the data packet to the target component. Chains of these components create processing pipelines and data flows.

The SCA raises a number of important questions based on what is and is not specified. For example, the architectural style embodied by the SCA is a distributed objects style, with a library of different kinds of components (objects) that can be used to construct SDRs.

It is unclear which came first: the style or the choice of middleware. The SCA developers may have decided that a distributed objects style was the right style for constructing software-defined radios and then chose CORBA as a middleware platform or framework for implementation. However, it is also possible that the SCA developers, encountering heterogeneous hardware platforms and component implementations (especially legacy component implementations embedded in non-software-defined radios), chose CORBA for its services in masking these differences, and the distributed objects style was just fallout from this decision. For the low-bandwidth control pathways in the architecture, distributed objects makes sense, but distributed objects is not a style that comes to mind for dealing with high-throughput data flows of packets passing through a radio. In this case, a hybrid style that had explicit support for streams, or at least asynchronous message-based data flows, might have been more appropriate.

Another question to ask is why CORBA itself was specified in the SCA, as opposed to other middleware that supports a distributed objects style such as COM, or an architecture framework that can be implemented with different kinds of middleware. The intent was likely to support interoperability among independently developed components: Because CORBA is not tied to one particular vendor, many CORBA implementations exist. However, interoperability between CORBA implementations is rarely perfect, and components developed against one implementation may have difficulty working in the context of another. It could be argued that the calling out of CORBA as a core aspect of SCA-based systems is a case of overspecification in the architecture.

The above descriptions talk about which aspects of a software-defined radio are specified by the SCA. It is perhaps more important to understand which aspects *are not* specified. The SCA surprisingly provides almost no guidance as to how to develop a radio based on the components and interfaces it defines. It contains little information about how to configure the components, to structure the radio application as a whole, or how to implement them. This leaves perhaps the most important and difficult decisions about constructing a radio entirely up to the radio developers themselves.

A final question to ask concerns the kinds of software qualities that are enabled by the use of the SCA. Portability, composability, and ease of integration may be enabled by this architecture because of the detailed interface specifications and the identification of specific middleware to connect elements. However, are these qualities the most important qualities in the development of a software-defined radio? Arguably, more important qualities such as performance and reliability are largely ignored by the SCA, and this is perhaps its biggest problem. At this time, the jury is still out on the success of the SCA; several commercial and

military efforts are underway to implement SCA-based radios. However, open-source and hobbyist groups also are implementing software-defined radios using their own architectures and frameworks and meeting with a measure of success. Time will tell whether the SCA will bear fruit or not.

The software-defined radio domain is also an example of where product lines can make a significant impact because the domain encompasses so much variability. Variability exists at many different levels of abstraction:

Network Topology: In this domain, radio devices do not stand alone: They communicate (via radio links) with other radios running other applications. Although software-defined radios can remain in one location, it is often the case that the radios themselves are mobile—mounted to vehicles or carried by individuals who move around. As such, the topology of devices can change as they move around or devices go on or offline.

Radio Configuration: A single software-defined radio board comprises a channel: a device that can be configured for one particular application at a time. In general, these devices are configured on a single board. More complicated radios support multiple channels by having multiple hardware boards mounted in the same radio backplane. Two-to-eight-channel radios are not uncommon for commercial or military applications.

Board Configuration: Each board in a software-defined radio can be implemented in a number of different ways. Although the basic elements (antenna, analog-to-digital conversion, and signal processing) are basically the same, they can be configured in different ways. Boards may include different FPGAs, different general-purpose processor configurations, different I/O ports and controllers, or different amounts of memory or storage.

Software Configuration: Each board generally has a number of processing elements on it that can be loaded with software: FPGAs and general-purpose processors, for example. Depending on the application, different components will be loaded onto the different processing elements.

Variability at each of these levels must be understood, controlled, and managed. The first thing to note is that these levels of abstraction are more or less hierarchical: Vehicles contain radios, radios contain boards, boards contain processing elements, and processing elements contain software elements. Modeling notations that directly support hierarchical architectures, such as Darwin, Koala, or xADL 2.0, are all good candidates for modeling such an architecture. Variability at each level is also somewhat limited: Specific software-defined radios will be developed for specific purposes (military, commercial, and so on). Thus, network topologies will have certain common characteristics. For a given organization, only a few different radio and board configurations will be produced: An organization may produce only a two-channel and a four-channel radio, for example. Board configurations may be limited to a “test board” and a “production board.” Software configurations will be partially driven by board configuration: Each device or controller on the board will likely have an associated device driver component; other components will be driven by the various applications loaded on the radio.

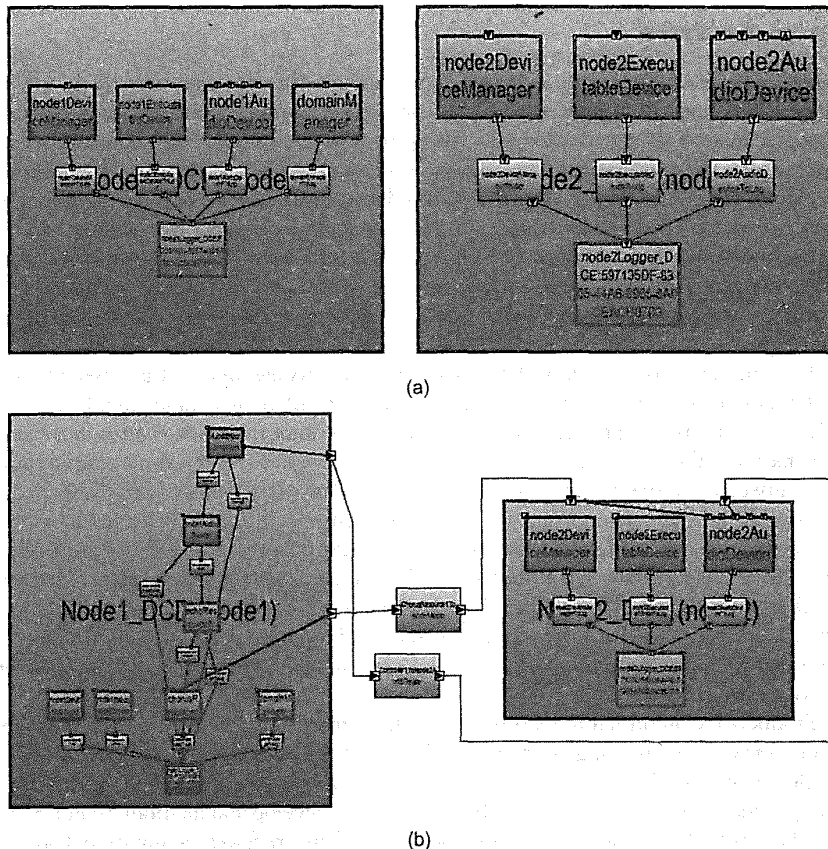


Figure 15-32. Software-defined radio structure with (a) just drivers and operating environment loaded, and (b) with a waveform application deployed.

Figure 15-32 shows two variants of the architecture of a software-defined radio called the SCA Reference Implementation (SCARI), modeled in xADL and depicted in xADL's graphical visualization. SCARI is a simple example of an SCA-based software-defined radio, intended to demonstrate how the SCA can be used to implement an SDR. Figure 15-32(a) shows the radio with only the device drivers and operating environment deployed. Figure 15-32(b) shows the same radio with an additional waveform application deployed. Each represents a different product-line variant, and by assigning guards to each the waveform components using techniques shown earlier in this chapter, waveform deployments can be selectively applied to the radio using product-line selection.

The extreme flexibility in the domain is induced by increasingly adaptable hardware, but is primarily enabled by software. Since software is so malleable, this should come as no surprise. Because of this, as well as other constraints that must be considered when dealing with SDRs—size, power consumption, performance, time to load and unload waveforms, and so on—SDRs are yet another domain where the capture and reuse of specialized engineering knowledge and experience can substantially reduce costs and risks in the development of new systems.

15.5 END MATTER

Domain-specific software engineering is, in many ways, the culmination of all the advice given in this book. It broadens the challenge of architecture-based development substantially: Instead of just applying architecture-centric principles to a single product, organizations and even entire communities must apply them across a range of related products. This is complicated by the fact that the products already developed in the domain were developed for different, often competing business goals, by isolated development groups that did not share engineering knowledge or architectural assets. This is perhaps the biggest challenge in architecture-centric software development, but it also has the greatest potential reward: a substantial reduction in costs and risks to develop new systems within an entire domain.

In some ways, the creation of a domain-specific software architecture resembles the development of an architecture for a single product—it still involves modeling, visualizing, analyzing, and applying principal design decisions. However, it is made even more difficult by the fact that the architecture must work for a wide range of systems. A constant tension exists between overspecification and underspecification—specifying too much and constraining the kinds of systems you can construct versus specifying too little and providing insufficient guidance and value. Key contributors may come from different backgrounds and organizations and may have fundamentally different ideas about how the architecture should be constructed. This chapter has provided a survey of the kinds of artifacts that make up a domain-specific software architecture, and provided some advice about how to create them. However, in the end, it still is a very human-centric process of negotiation and cooperation.

Product-line architecture techniques provide a more concrete way to apply domain-specific software engineering “in the small.” Product lines are typically developed within a single organization and target a much narrower set of products. Because of this narrower focus, product-line architectures are often far easier to develop and maintain than DSSEs, since there is far less variation in business goals and there are fewer organizational boundaries to cross. Substantial cost reduction and risk mitigation can still be achieved within the product line, especially if automated tools can help with activities such as product-line selection.

Domain-specific software engineering highlights how software architecture in practice is both an organizational and a technical issue. We have highlighted how business concerns motivate certain development practices and techniques. The next chapter will continue this theme by focusing on architecture-related standards. The creation, adoption, and use of standards are also driven by business goals, and standards are a big part of how software architecture is viewed and used in practice.

The Business Case

The primary attraction and benefit of domain-specific software engineering is found through *reuse*. Reuse, continual learning, and refinement of knowledge about how to build systems is the hallmark of any engineering discipline. Software reuse has therefore been widely recognized and often touted as a way of controlling the complexity of software systems, improving their reliability, speeding up their development, reducing their associated costs, and so on. However, a cursory look at the practice of software reuse indicates that most frequently it entails *scavenging*

source files for needed code fragments, *cloning* modules needed in multiple places in a system (which is, incidentally, something that good software engineering practice discourages), or *purchasing* entire commercial-off-the-shelf (COTS) solutions.

Therefore, reuse in software development is predominantly focused on source code. However, this is misaligned with the near-universal awareness that the key software engineering costs lie elsewhere—in requirements elicitation and system design—and that software architecture is a system's (and, often, a development organization's) principal asset. In other words, source code-level reuse is not the most effective reuse.

DSSE recognizes and addresses this discrepancy. It provides mechanisms for expanding the notion of reuse to include all concerns in the development of a software system (so long as that system falls within a particular application domain). This includes:

- The canonical, *reference* system requirements that are in some significant way shared across all systems in the domain.
- The terminology specific to a domain, and sometimes *invented* for the sole purpose of enabling effective software development in that domain.
- The canonical, *reference* architecture, with clearly identified variation points.
- The canonical, low-level system designs that realize the architecture.
- The analysis, simulation, and testing models and tools.
- The domain-specific implementation techniques, off-the-shelf components, and code generators.
- The system construction and deployment scripts.
- The predicted evolution patterns and associated mechanisms.
- The required knowledge and training of software engineers.
- The processes for managing different projects with shared characteristics, existing domain-specific assets, already developed systems, and systems currently under development, as well as planning future systems.

The argument here is that, in theory, this could be done for any software system. In practice, it is not possible. But if your domain is reasonably well defined and stable, and if you have a base of experience to draw from, leverage it!

If it seems as if this is a lot to manage, it is. But it is still easier, and less expensive, to manage known assets than it is to discover them. Organizations and communities constructing complex systems within a domain will undoubtedly still have to consider, capture, and codify all the aspects that are part of a DSSA. This cost, while high, provides adequate return on investment—the goal here is to capture once, reuse forever.

The explicit capture and use of product lines is an even more business-centric practice. Product lines facilitate reuse at the architectural level, and allow organizations to construct and manage new product variants with only incremental costs. As markets diversify and organizations do business with an increasing array of cultures, countries, standards, and customer needs, being able to tailor and market a wide variety of similar products becomes increasingly important. As a side benefit, the same product-line techniques can be used to help understand the evolution of individual products over time and to capture what-if decisions and rationale in more explicit ways, further bolstering an organization's core business knowledge.

A caveat: Domain-specific software engineering is largely a horizontal endeavor; it means surveying and unifying engineering activities across products, and perhaps even across business areas. Organizations that can adapt to this style of engineering can exploit DSSE. However, many organizations still treat product development as a series of independent “stovepipes,”

where personnel, engineering knowledge, and budgets are strictly partitioned across product boundaries. If this remains the case, an organization will never be able to exploit DSSE, because it cannot make the horizontal investments necessary to exploit commonalities and reuse.

15.6 REVIEW QUESTIONS

1. What is domain-specific software engineering (DSSE)? What problems in software development does it address?
2. What are the three main concerns addressed by DSSE? How do they relate to one another?
3. How does DSSE help to partition problem spaces and solution spaces? How does this make software development easier?
4. Enumerate the various artifacts that are part of a DSSA. What information does each artifact contain? What is its purpose?
5. What is a reference architecture? How does it differ from an ordinary software architecture? What different kinds of reference architectures can be created?
6. When should a reference architecture be developed?
7. What is a product line? How does it differ from a domain?
8. What is a product-line architecture? What is the relationship between a reference architecture and a product-line architecture? What are variation points?
9. Identify strategies for unifying multiple related products into a product line.
10. What is the relationship between architectural design decisions, features, and products in a product line?
11. How can product lines be used to capture variations in a single product over time?
12. How can product lines be used to capture what-if scenarios in product development?
13. What is product-line selection and how is it performed?
14. What are some strategies for unifying products with different intellectual heritages into a product line?
15. What is the relationship among DSSAs, product lines, and architectural styles?
16. How does Koala assist developers in domain-specific software engineering? What Koala features enable it to capture product-line architectures?
17. How can domain-specific software engineering techniques help in the construction of software-defined radios? How can software-defined radios form a product line?

15.7 EXERCISES

1. Select a problem domain with which you are familiar and try to produce a domain model as described above. Which of the domain modeling activities pose the greatest challenges? Why?
2. Survey some of the architecture-based development approaches presented in this book, and identify how (or if) they address the three primary concerns of domain, business, and technology. Is any one concern favored over others? Why?
3. Find two to four examples of systems that are in the same domain but are not built by the same organization. Create tables, like those in this chapter, showing the elements in each product, grouping them into features, and showing how features are composed into the products. Speculate on how you might use DSSE and product-line techniques to unify these products into a product line, or to extract a DSSA from them.
4. Use an architecture description language that supports product-line modeling such as Koala or xADL to capture at least three different Lunar Lander games with different features.
5. Download and review the SCA specification. Which aspects of a DSSA are covered by this specification and which are not? Is it an adequate reference architecture for the domain of software-defined radios?

6. Find an open-source project that has multiple product variations—perhaps for reasons of running on different platforms, providing different services, or

working in an international context. Determine how these product variants are managed. Is an architectural approach at work?

15.8 FURTHER READING

A tremendous amount of the general DSSE/DSSA information in this chapter comes from early work on domain-specific software engineering in the avionics domain by Will Tracz and Don Batory et al. (Tracz 1995; Tracz 1993; Tracz, Coglianese, and Young 1993; Tracz 1996; Batory, McAllester et al. 1995; Batory, Coglianese et al. 1995; Tracz, Coglianese, and Young 1993). These references not only provide deeper background on DSSE and DSSAs in general, but also discuss yet another application domain in which these techniques have been applied. Tracz has also written extensively on reuse in general, with most of his articles compiled into his book *Confessions of a Used Program Salesman* (Tracz, 1995).

The Koala (van Ommering et al. 2000) work is another excellent example of the successful use of architecture-centric development in practice. Although the early Koala work was focused on individual product lines, Rob van Ommering has expanded his work to cut across prod-

uct lines and address product populations (van Ommering 2002) as well.

There is extensive literature on product lines. Paul Clements and Linda Northrop (Clements and Northrop 2002) write about product lines from a high-level perspective, focusing on management issues, processes, patterns of practice, and case studies. Hassan Gomaa (Gomaa 2004) takes a UML-centric approach to modeling both product lines and patterns found therein. Klaus Pohl et al. (Pohl, Böckle, and van der Linden 2005) also take a primarily process-centric approach.

Another substantial body of work on product-line architectures and modeling comes from André van der Hoek et al. in the Ménage project (van der Hoek 2000; Roshandel et al. 2004; van der Hoek 2004), which captured options, variants, and versions explicitly in architectural models and which inspired the product-line features in xADL.

Standards

Over the years, various organizations have attempted to create standard architectural approaches, techniques, and tools for software systems. Other standards simply emerged as a result of common use among many organizations, such as the Unified Modeling Language (UML).

Some of these standards have achieved widespread adoption and significant backing in industry and government, and often are mandated for use on different projects. In general, these standards encapsulate a significant amount of engineering knowledge and guidance. However, it is important for software architects and other stakeholders to establish and maintain a rational perspective concerning these standards: what they are good for, what they can provide, and what they fail to provide. Often, the momentum behind a standard creates a (vastly) overinflated perception of its real value and impact. This chapter puts some of the most influential standards in context and identifies their strengths and weaknesses.

16 Standards

16.1 What Are Standards?

16.1.1 Why Use Standards?

16.1.2 Drawbacks of Standards

16.1.3 When to Adopt

16.2 Specific Standards

16.2.1 Conceptual Standards

16.2.2 Notational Standards

16.2.3 SysML

16.2.4 Standard Tools

16.2.5 Telelogic System Architect

16.3 Process Standards

16.3.1 Rational Unified Process

16.3.2 Model-Driven Architecture

16.4 End Matter

16.5 Review Questions

16.6 Exercises

16.7 Further Reading

Outline of Chapter 16

16.1 WHAT ARE STANDARDS?

Definition. A *standard* is a form of agreement between parties.

From a software architecture perspective, there are many different kinds of relevant standards: standards for notations, tools, processes, interfaces, organizations, and so on. Standards can come from different sources: individuals, teams, single-practitioner organizations, coalitions of such organizations, governments, or organizations whose specific goal is to create and codify standards such as the American National Standards Institute (ANSI), Ecma International, the International Organization for Standardization (ISO), and the World Wide Web Consortium (W3C). When a standard is controlled by a body that is considered to be authoritative, the standard is often called a *de jure* standard (meaning “from law”). *De jure* standards often:

- Are formally defined and documented.
- Evolve through a rigorous, well-known process.
- Are managed by an independent body, governmental agency, or multi-organizational coalition rather than a single individual or company.

Because of these characteristics, *de jure* standards tend to be the ones that are externally mandated for use in specific projects. The other kind of standard is a *de facto* standard (meaning “in fact” or “in practice”). *De facto* standards often:

- Are created by a single individual organization to address a particular need.
- Are adopted due to technical superiority or market dominance of the creating organization.
- Evolve through an emergent, market-driven process.
- Are managed by the creating organization or the users themselves, rather than through a formal custodial body.

The line between a *de jure* and a *de facto* standard is not always clear. For example, HTML is a standard notation for encoding Web pages and has all the above characteristics of a *de jure* standard, but the W3C has no authority to force individual Web publishers to follow the HTML standard—instead, market issues of browser compatibility tend to be the factor driving adoption. Similarly, *de facto* extensions to HTML have been created by companies such as Microsoft and Netscape that are widely used simply because of their creators’ dominance in the browser market. In the computing world, popular *de facto* standards often evolve into *de jure* standards: Netscape developed the JavaScript Web scripting language on its own and included support in its browser. Microsoft created a workalike language for its Internet Explorer browser called JScript. As more and more people became dependent on this technology, Netscape submitted it to a standards body (Ecma International), where it has been standardized as ECMAScript.

Both *de jure* and *de facto* standards can be *open*, *proprietary/closed*, or somewhere in between. Totally open standards allow public participation in their development; anyone is free to come in and suggest improvements or changes. Totally proprietary or closed standards mean that only the custodians for the standard can participate in its evolution. Standards bodies such as ANSI, ISO, and Ecma International tend to fall somewhere in

the middle; some of these allow virtually open membership, while others have a higher barrier to entry (voting of the existing members or a steep membership fee, for example).

Stakeholders and architects should be keenly familiar with the state of a particular standard before adopting it by asking key questions: Who supports the standard? How does it evolve? Is it dependent upon one specific company—that may be a competitor? What are its success and failure stories? How can feedback be incorporated into the standard's future development?

16.1.1 Why Use Standards?

Standards can provide many different tangible benefits to their users. Many of these benefits derive from standards' ability to create *network effects*. A network effect occurs when the value of a particular service or product goes up as more users employ it. The Web exhibits a classic example of a network effect: As more and more users publish content on the Web through the HTTP (Fielding et al. 1999) and HTML (Raggett, Le Hors, and Jacobs 1999) standards, the value of the Web as a whole increases for each of its users, since each of these users can access the new content without any additional investment.

Standards can ensure consistency across projects or organizations. Consistency is often the key driver of a network effect. Consider the value of the use of consistent symbology in a notation such as UML: As more stakeholders describe systems using UML, more people can develop an understanding of those systems without investing in learning a new notation.

Certain kinds of standards, such as interface or behavior standards applied to products, can be used to ensure interoperability or interchangeability. For example, the HTTP protocol can be seen as an interface standard; it dictates how to interact with a component, but not how that component must be implemented internally. Therefore, components developed by diverse organizations can all interoperate (on some level) as long as they conform to the HTTP standard. Interchangeability is a higher level of standardization; interchangeable software components can be swapped out for one another with little or no effort. In the world of software, interchangeable parts often exhibit different quality characteristics: One may be optimized for use in a single-processor system while another may be optimized for use in a distributed system.

Standards are bearers of engineering knowledge and can carry it from project to project. Often, standards are developed because organizations continue to reinvent the same basic techniques or tools over and over, at great expense. Standards reduce costs for the entire market by collecting and making the engineering knowledge available to everyone.

Standards also serve as targets for tool vendors. Organizations that adopt widely accepted standards generally will have the ability to buy off-the-shelf tools that implement the standard along with training supplements such as books, videos, and so on, instead of investing time and money to develop their own. For the most popular standards, adopting organizations often will have a choice of several competing tools, and market competition can drive further improvements in these tools.

Standards can also reduce the cost of powerful software tools. While software initially is expensive to produce, once it is created it can be replicated infinitely at little to no additional unit cost. In economic terms, software makes excellent use of *economies of scale*. By conforming to a standard, a tool can gain wider appeal and thus sell more units. As more units are sold, the developing organization can amortize the initial development cost across

all the units, greatly reducing the price it needs to charge to be profitable. For example, consider two software tools that both cost \$1 million to develop. If one tool sells 100 copies, then the price for each unit must be above \$10,000 just to recover the development cost. However, if the tool sells 100,000 copies, then the price could drop to only \$10 per unit. A tool that conforms to a widely accepted standard is much more likely to sell many copies than a tool that does not.

Over time, standards evolve. The procedure for evolving a standard is generally governed and managed by the standard's custodial organization. As standards are adopted and used, various weaknesses or drawbacks will be identified, and future versions of the standard can be developed to rectify some of them. Organizations that adopt the standard can take advantage of these lessons and the resulting improvements in the standard without bearing the entire brunt of the costs of developing them.

Having substantial control over a widely accepted standard can grant an enormous amount of market power to a custodial organization. If a single organization, such as a commercial company, develops a standard that becomes widely adopted, then that company has a great deal of control over how the standard is used and how the standard evolves. It can guide the standard's evolution in directions that match its own business interests—or drive it away from a competitor's interests.

Overall, the use of standards can carry many benefits. However, standards are not a panacea, and must always be evaluated with a critical eye.

16.1.2 Drawbacks of Standards

A host of benefits can be conferred upon organizations that adopt and develop standards. In doing so, organizations must work to mitigate or avoid some of their drawbacks.

The very nature of standards is somewhat at odds with the realities of software architecture. One message that this book has conveyed is that stakeholder needs and desired qualities of software vary widely from project to project and domain to domain. In contrast, standards often attempt to apply the same concepts and processes to widely different development efforts. This is probably the key weakness of most architecture-related standards in use today.

A related drawback is that the most widely adopted (and thus best-supported) standards are those that are the most general. This should not be surprising—the adoption potential of a particular standard is proportional to its applicability. However, this means that domain-specific standards that can impart a lot of specific guidance and value to projects within a domain are often the ones that have the least support—from tools, researchers, and so on.

The primary tension in standards development exists between overspecification—prescribing too much and limiting the utility of the standard—and underspecification—prescribing too little and limiting the benefits of the standard. As you might expect, the most broadly accepted standards tend to be the ones that are underspecified, leaving out significant semantic details. Without well-defined semantics, it is difficult to make quality judgments about a product simply because it conforms to a standard.

A related problem is the *least-common-denominator* problem. Because standards are often developed by a committee of diverse, often competing organizations, the individual goals of the organizations will tend to diverge along many dimensions. When many organizations are involved, divergent viewpoints are often simply left out of the standard, with

only agreeable elements included. When this occurs, the standards that emerge tend to be small and underspecified.

The opposite problem can also occur: When divergent opinions occur among the developers of a standard, everyone's solutions are included, usually leaving the choice of approach up to the user of the standard. These standards are underspecified in a different way: While there is no shortage of options for users, there also is little guidance as to which approach to choose; the fundamental benefits of using a standard such as interoperability and consistency are reduced.

One critical element that distinguishes de jure standards from de facto standards is market adoption. A de facto standard has become a standard because it fulfills some need for a significant number of users. For these standards, market forces and network effects are primary drivers of adoption.

De jure standards (at least, those that have not emerged from de facto standards) are not similarly motivated. They are often developed and codified without preexisting market adoption. The quality of these standards is thus brought into question—they have not competed in an open marketplace of ideas and thus may be inadequate solutions even for the problems they are intended to address. This mirrors a common criticism levied at academic solutions to problems: that they come from the “ivory tower” in that they may be well-grounded in theory but are untested in practice. This is especially troubling when these standards are mandated for use in projects. Here, there is a significant danger that being *standards-compliant* will be more important than being *good*.

De jure standards usually arise from good intentions. Often, recognized experts in a particular domain are tapped to help produce the new standard. Sometimes, these standards result from composing existing de facto standards as well.

De facto standards are not a panacea either—there are many examples of de facto standards that have different problems—for example, poor support for evolution of the standard, or widespread adoption inhibiting innovation.

The key questions to ask about any standard are:

- What problem does this standard address?
- What are the limitations of this standard? (That is, what problems does this standard not address?)
- Has the standard been successfully applied?
- Whether or not it has been applied, is the standard theoretically sound?
- Are the practical reasons for adopting this standard compelling?
- What benefits can be expected from using this standard? Alternatively, what work can be avoided by adopting the standard?
- Are there alternatives to adopting the standard? What are the costs and benefits of the alternatives?
- What strategy is in place for dealing with the evolution of the standard?
- What contingency plans are in place if the standard fails to evolve?

**De Jure
Standards:
The Ivory
Towers of
Industry?**

16.1.3 When to Adopt

One of the critical questions that every development organization must answer when faced with a new standard is when to consider adoption. How an organization affects and

is affected by a standard is largely governed by how and when it chooses to adopt that standard. Both early and late adoption have advantages and disadvantages.

Some of the benefits of early adoption include:

Ability to influence the standard: Early adopters can often become significant participants in the body that is responsible for creating the standard. They may then be able to move or evolve the standard in a way that is advantageous for them, or work to exclude competitors from similar participation.

Early to market: If the standard becomes successful, early-adopter organizations are the first to benefit. If the standard can be leveraged in the product market, then the organization's products will be among the first to support the standard.

Early experience: If the standard is successful or useful, early adopters will have more experience than others with using the standard, and can leverage this experience to their benefit.

Some of the drawbacks include:

Risk of failure: If the standard ends up not being successful, early adopters can be left holding the bag. Investment in the standard will not pay off because the standard will not be supported.

Moving target: As standards are initially developed, they go through numerous changes. Early adopters will have to adapt to these changes as they occur; if they develop products based on an early version of a standard, they may not be fully standards-compliant when the standard is eventually released.

Lack of support: Early in their development, standards will not have extensive support. Certain problems may not have been ironed out. Without support, organizations may end up developing support of their own, thus negating some of the cost benefits of adopting a standard at all.

Some of the benefits of late adoption include:

Maturity of the standard: Older standards tend to be more mature. They are more stable and undergo fewer changes; the changes that are made are done through a more principled process. Bugs in the standard likely have been worked out, or at least documented and worked around.

Better support: Older standards tend to be better supported in terms of tools, documentation, training materials, and so on. This support may be developed by other organizations, and can be bought rather than developed.

Some of the drawbacks of late adoption include:

Inability to influence the standard: It is, in general, more difficult to influence a standard after it has been developed and supported for a long time. Existing organizations in the standards body will have formed a community with its own culture, and the barrier to entry into this community will gradually grow over time.

Restriction of innovation: Older standards may not track cutting-edge developments in technology or methods. Becoming attached to an older standard may limit an organization's view to innovation.

So, when should an organization adopt a standard? The answer depends on the organization's goals. If the standard will substantially influence an organization's core business, it is probably more useful for the organization to take the risk and absorb the costs of getting involved early. If the standard fails and the organization has become involved, then only the original investment is lost; if the standard succeeds and the organization has not become involved, the organization may find itself being unduly influenced by a standard it cannot control. Alternatively, if the standard is simply a tool that the organization will use to improve its own products but will not be part of its core business, then waiting for others to foot the bill for maturing the standard is a smarter decision.

16.2 SPECIFIC STANDARDS

There are many standards related to software architecture that are widely accepted and in use today. We will attempt to categorize each standard based on its primary purpose, but recognize that many of these standards are difficult to peg in any one category.

16.2.1 Conceptual Standards

Conceptual standards tend to address the range of activities that go into creating and using architecture. Because of their wide scope, they tend to provide high-level guidance. For example, many focus on what should be documented, but not how (that is, in what notation) it should be documented. Conceptual standards often dovetail with other, more specific standards for methods or notations to form more complete and detailed approaches to software and systems development.

IEEE 1471

IEEE 1471 (IEEE-SA Standards Board 2000) is an IEEE recommended practice for architectural description that is often mandated for use by government contractors and other organizations. It recognizes the importance of architecture in helping to ensure the successful development of software-intensive systems. IEEE 1471's scope is limited to architectural descriptions; that is, architectural descriptions can be conformant to the standard, while other artifacts (products, processes, organizations) cannot. Architectural descriptions are a collection of products used to document an architecture: In this book, we refer to these artifacts as architectural *models*. The standard itself does not describe a specific notation for these models, however. Instead, it only specifies a minimal amount of required content in the models.

IEEE 1471 takes a stakeholder-driven view of architectures, much as we have done in this book. Architecture descriptions in IEEE 1471 specifically identify system stakeholders and their concerns. A complete architectural description will address all the concerns of all the stakeholders, although in practice this is more often an ideal than a reality.

IEEE 1471 architecture descriptions are also composed of views that are instances of viewpoints. The IEEE 1471 notion of views and viewpoints is consistent with the one used in this book. As we have discussed, views and viewpoints organize the architecture according to stakeholder concerns. Just as the standard does not advocate the use of any particular notation, it also does not advocate the use of any particular set of viewpoints—adopters of

the standard are expected to choose viewpoints that correspond to its stakeholders' concerns. The standard encourages implementers to maintain consistency among views and to document any known inconsistencies in the architecture description. It does not explain how consistency is supposed to be achieved or measured or the methods that should be employed to do so.

As a standard, IEEE 1471 is purposefully light on specification (it is difficult to call it underspecified because the limits of its scope are obvious and deliberate). The standard provides a good starting point for thinking about capturing architectures, and does not try to advocate a single approach for every domain. The definitions and concepts that it uses are more or less consistent with those found in this book.

Users must be careful not to attribute too much capability to the standard, however. Our experience is that there is a belief among many organizations that being IEEE 1471-compliant is the equivalent of "doing good architecture." In fact, being IEEE 1471-compliant indicates that a set of "architecture description" products have been developed and designated by the system's engineers, but does not ensure that these products are of high quality, that they capture the critical aspects of the system, that they are expressed in appropriate notations, or that any reasonable methodology has been used to develop and check these products for consistency. As such, IEEE 1471's contributions are largely in defining a vocabulary of concepts and their relationships to one another.

DoDAF: The Department of Defense Architecture Framework

DoDAF (DoD Architecture Framework Working Group 2004) is a U.S. Department of Defense standard for documenting system architectures. It is the successor to the previous C4ISR architecture framework and supercedes it. DoDAF (and standards like it) are often referred to as architecture frameworks. Here, "framework" generally designates a process or set of viewpoints that should be used in capturing an architecture. This sense of framework is different from the architecture implementation frameworks we discussed in Chapter 9.

Like IEEE 1471, DoDAF's contributions are mostly at the conceptual level. DoDAF more deeply prescribes the kinds of things that should be captured in an architecture description. Unlike IEEE 1471, it advocates a specific approach that is more divorced from stakeholder concerns, as it identifies specific viewpoints and the sorts of information that should be captured in each of them. Like IEEE 1471, DoDAF leaves many choices up to the user: for example, the kind of notation(s) to use to document the architecture views and how to ensure consistency among them.

DoDAF terminology conflicts somewhat with the terminology used in this book with respect to views and viewpoints. The correspondence is roughly as shown in Table 16-1. Throughout the remainder of this discussion, we will use our terminology (viewpoint set,

Table 16-1
Terminology
Differences

Concept	Our Term	DoDAF Term
A set of perspectives from which descriptions are developed	Viewpoint set	View
A perspective from which descriptions are developed	Viewpoint	(Kind of) product
An artifact describing a system from a particular perspective	View	Product

viewpoint, and view). However, the DoDAF standard itself uses the alternative terms (view and product).

DoDAF is a multi-viewpoint approach. The various viewpoints address architectural design decisions at many levels, from the most abstract (concepts, missions) to the reasonably detailed (individual topologies of components and connectors). DoDAF viewpoints are grouped into four sets.

The operational view (OV): Views in the OV viewpoint set identify “what needs to be accomplished, and who does it.” They define processes and activities, the operational elements that are part of those activities, and the information exchanges that occur between the elements.

The systems view (SV): Views in the SV viewpoint set describe the systems that provide or support operational functions and the interconnections between them. The systems in the SV are associated with elements in the OV.

The technical standards view (TV): The views in the TV viewpoint set identify standards, (engineering) guidelines, rules, conventions, and other documents intended to ensure that implemented systems meet their requirements and are consistent with respect to the fact that they are implemented according to a common set of rules.

All views (AV): DoDAF also recognizes some cross-cutting concerns that affect all views (AV). AV views include high-level overviews of the system and the environment in which the system will be deployed and a unified dictionary that defines terms used throughout all other views.

The DoDAF viewpoints are optimized for documenting complex architectures consisting of many interconnected and communicating nodes. These architectures are often called system-of-systems because the system as a whole is made up of constituent parts that are themselves complex systems with their own architectures.

To illustrate how the DoDAF is used to model a system-of-systems, we present a Lunar Lander application modeled in a subset of the DoDAF viewpoints. An OV-1 view is a stylized graphical representation of the system’s operation. It is a free-form overview of the application and important elements in it. The DoDAF views are not optimized for simple one-node systems like the basic Lunar Lander application we have covered in this book so far. However, in a more complex Lunar Lander system, communicating nodes such as ground stations, satellites, sensors, and so on might be depicted in such a view. Figure 16-1 shows an OV-1 view of a system-of-systems Lunar Lander example. Here, the Lunar Lander itself communicates with the ground utilizing a satellite to relay data to a satellite communication station, which in turn is connected to a ground station. This is the most conceptual and least rigorous DoDAF viewpoint; other OV products are less conceptual and more concrete.

Figure 16-2 shows another DoDAF OV view, an OV-4 “Organizational Relationships” view. This diagram shows the relationships among various organizations and suborganizations that are involved in a project. Here, the main space agency oversees various subagencies, with assistance from an oversight organization. One subagency, the *Lander Crew*, is expanded to show the individual roles that are assigned to personnel within that subagency. Note that the OV-4 diagram primarily depicts the relationships between people and not technological artifacts. The blending of organizations, personnel, and technology typifies OV diagrams in DoDAF.

Figure 16-1.
DoDAF OV-1
view of Lunar
Lander.

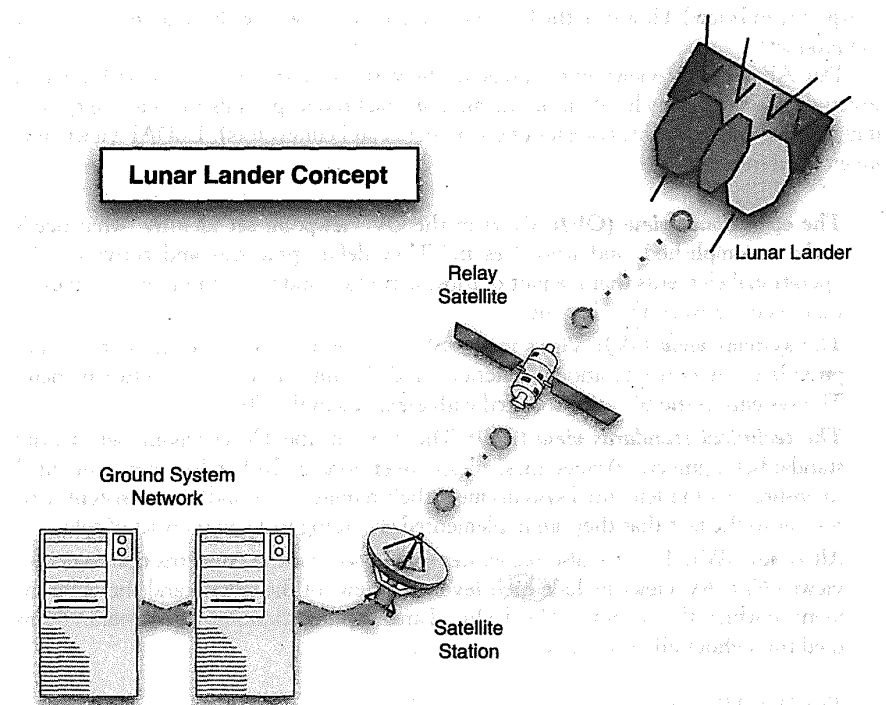
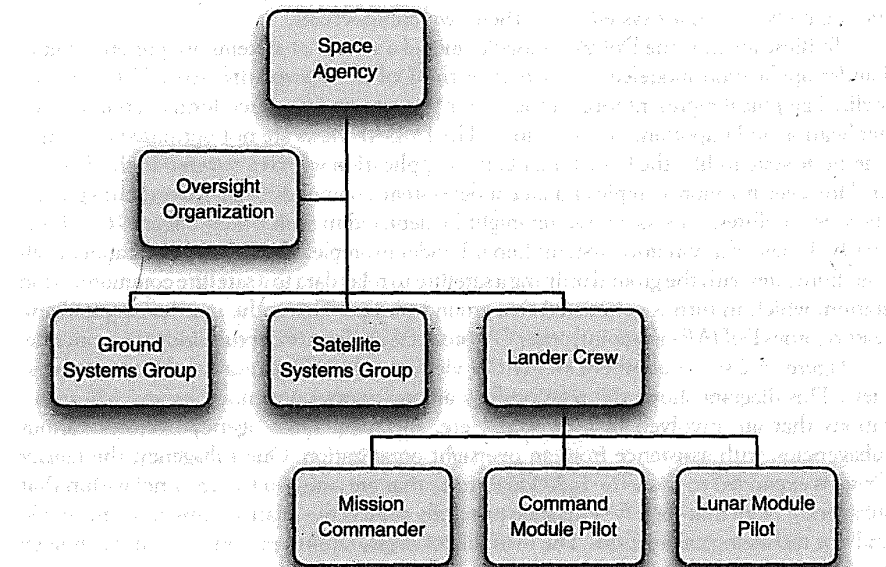


Figure 16-2.
DoDAF OV-4
diagram of Lunar
Lander.



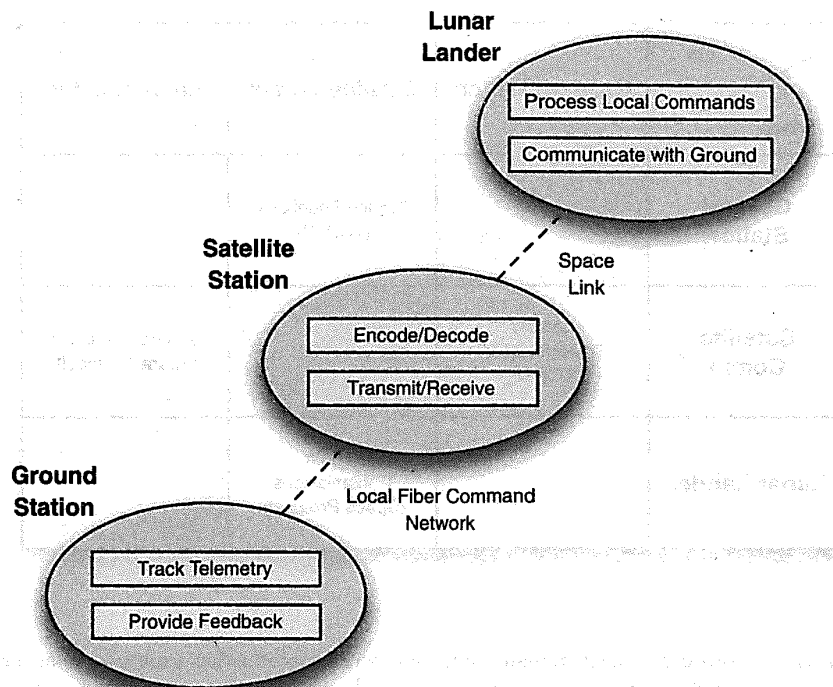


Figure 16-3.
DoDAF SV-1
diagram of Lunar
Lander.

DoDAF SV views tend to be more technical in nature. Figure 16-3 shows an SV-1 “Systems Interface Description” view. The nodes are intended to correspond to nodes shown in OV diagrams. This shows an initial breakdown of functional responsibilities among the nodes, as well as the interconnections between them. Many such SV-1 views can be created for the same system, depicting the system at different levels of abstraction or showing interconnections between individual functions on nodes, rather than just the nodes themselves.

DoDAF makes extensive use of what it calls matrix views, also known as “ N^2 diagrams.” These views are generally depicted as tables. Both axes of the table are populated with an identical set of nodes. These nodes generally represent concrete elements such as systems, subsystems, hosts, or devices. Each cell in the body of the table represents a possible interconnection between two of the nodes. If the two nodes are connected, details about the connection are listed at the intersection cell; otherwise, the cell is left blank. A matrix view contains all possible connections between elements.

Matrix views are often directly correlated with graph (that is, box-and-arrow) views of a system. Each node in the graph is listed on the axes of the table, and each line on the graph corresponds to a filled cell in the body of the table.

Figure 16-4 shows one such diagram, an SV-3 “Systems-Systems Matrix” diagram. In this diagram, the nodes on the axes are drawn from the earlier SV-1 diagram shown in Figure 16-3. Nodes on the diagonal are grayed out; communication between a node and

Figure 16-4.
DoDAF SV-3
diagram of Lunar
Lander.

to from	Ground Station	Satellite Comm	Lunar Lander
Ground Station		Ground Feedback (TCP/IP)	
Satellite Comm	Lander Transmissions (TCP/IP)		Ground Feedback (Space Protocol)
Lunar Lander		Lander Transmissions (Space Protocol)	

itself is not considered. Each remaining node describes information exchange from the node named on the vertical axis to the node named on the horizontal axis. Nodes where no information is communicated, or no link exists (for example, from the *Ground Station* directly to the *Lunar Lander*) are left blank. Here, as with other DoDAF diagrams, the amount of detail present is determined by the user. The diagram in Figure 16-4 is relatively simple, showing only the kind of data transmitted and the protocol used. However, the DoDAF standard lists many types of information that can go in each node: status, purpose, classification level (such as unclassified or secret), means (such as the kind of network or protocol used), interface standard, and so on.

DoDAF TV products focus on identifying and tracking the technical standards that will be used to implement the described system-of-systems. DoDAF assumes that many aspects of these systems will be implemented using reusable off-the-shelf components that conform to various published standards. TV products provide ways of identifying what those standards are.

A portion of a TV-1 diagram for the Lunar Lander is shown in Figure 16-5. The leftmost column of the table shows a number of areas in which standards apply. The second column shows subareas. The DoDAF and a partner specification called the Joint Technical Architecture (JTA) enumerate these areas. Each system or subsystem, drawn from the earlier SV-1 view in this case, is associated with standards drawn from these areas. Each system or subsystem can also be given an absolute or relative time frame in which it will comply with or leverage the standard. For example, in this diagram, the *Ground Station*, *Satellite Station*, and *Lunar Lander* are all running operating systems that conform to the POSIX standard (IEEE 2004). Not all subsystems must use all selected standards, of course. Here, the *Ground Station* communicates with the *Satellite Station* using XML, sent over a Fiber Distributed Data

Standards for SV-1 Systems			Ground Station	Satellite Comm	Lunar Lander
Service	Service Area	Standard			
Information Technology Standards	Operating System Standard	ISO/IEC 9945-1:1996, Information Technology - Portable Operating System Interface (POSIX) - Part 1: System Application Program Interface (API)	Baseline: 1 Jan 2020	Baseline	Baseline + 3 mos
Information Transfer Standards	Data Flow Network	Extensible Markup Language (XML) 1.0 (Fourth Edition) W3C Recommendation 16 August 2006	Baseline + 6 mos	Baseline + 6 mos	
	Physical Layer	FDDI / ANSI X3.148-1988, Physical Layer Protocol (PHY) -- also ISO 9314-1	Baseline + 3 mos	Baseline + 3 mos	

Figure 16-5.
DoDAF TV-1
diagram for
Lunar Lander.

Interface (FDDI) physical network. Different standards (not shown here) would facilitate the communication between the *Satellite Station* and *Lunar Lander* subsystems.

DoDAF describes, in great detail, the breadth of information that can be captured in an architecture description. It also categorizes that information into viewpoints and, where appropriate, describes potential points of correspondence between viewpoints. While it introduces substantial complexity, this breadth is DoDAF's primary strength. In some sense, DoDAF can be seen as a huge checklist of items that may be useful to capture or think about in the development and modeling of a system's architecture. As with any modeling notation, users must decide for themselves the kinds of models to create, the level of detail, and so on. DoDAF offers some advice in this regard, discussing different uses for architecture models and the viewpoints that are (in whole or in part) applicable to that use. However, even with this advice, users must still decide how they want to use the DoDAF on a case-by-case basis.

DoDAF does not, in general, mandate any particular method or notation for documenting the various viewpoints. Therefore, its users can choose any subset of the viewpoints they want and document them using any notation or tool they want. To its credit, the DoDAF specification provides examples for each viewpoint, including UML examples.

However, these examples rarely exercise all possible details that can be captured in a view. Following the examples too closely would result in a set of views that left out major details. To strengthen the connection between DoDAF and UML, the OMG has submitted a request for proposals for DoDAF UML profiles (Object Management Group 2005). However, UML is not ideal for all DoDAF viewpoints (UML has no tabular diagram that would work for the matrix viewpoints, for example). Users must select notations and then weigh the advantages and disadvantages of their selection. Choosing several notations will increase the difficulty of coordinating them. Selecting a single notation might leave out important details, or make some viewpoints cumbersome.

Takeaway Message. Compared to the leanness of IEEE 1471, DoDAF takes the same high-level approach but goes into much more detail. It is optimized for capturing the relationships among people, organizations, system design decisions, and technical standards for complex and composite systems. It provides users with an extensive amount of information about *what* sorts of things to capture when developing an architecture, but far less information about *how* to capture that information or evaluate the results. Additional extensions to the standard, such as the aforementioned mapping of DoDAF views to UML, can fill in this gap. Because of the wide scope of the standard, it may be difficult for users to identify the views that are important to capture for a particular system. Because of its specificity, it may give users a false sense of security—that everything has been covered when, in fact, there are some stakeholder concerns that are being short-changed. A remedy for this is to focus first on stakeholders and concerns (as IEEE 1471 suggests) and then identify which DoDAF views map onto these concerns.

TOGAF: The Open Group Architecture Framework

The Open Group Architecture Framework (TOGAF)* (The Open Group 2003) is the product of an iterative collaborative effort by members of The Open Group, a collection of more than 200 companies and other organizations. TOGAF is an “enterprise architecture framework.” It takes into account concerns beyond hardware and software, such as human factors and business considerations. Enterprise architectures tend to focus on organization-wide solutions to implement business goals. Elements in an enterprise architecture might include databases, applications such as spreadsheets and Web browsers, various client and server machines, as well as people in various roles. TOGAF borrows concepts and terminology from IEEE 1471; its notions of architecture, views, viewpoints, and so on are consistent with those discussed above in the context of IEEE 1471. Various versions of TOGAF have been developed over the years; each version has grown to encompass more aspects of architecture. TOGAF version 8 encompasses:

Business concerns, which address business strategies, organizations, and processes.

Application concerns, which address applications to be deployed, their interactions, and their relationships to business processes.

Data concerns, which address the structure of physical and logical data assets of an organization and the resources that manage these assets.

Technology concerns, which address issues of infrastructure and middleware.

*TOGAF is a trademark of The Open Group.

TOGAF consists of three major elements: an architecture development process called the ADM (architecture development method); a “virtual repository” of architecture assets (such as models, patterns, and architecture descriptions) extracted from architectures developed in practice, called the Enterprise Continuum; and a resource base consisting of guidelines, templates, background information, and so on to assist enterprises in following the ADM.

The centerpiece of TOGAF is the ADM, a process for developing enterprise architectures. Like many other modern software processes such as spiral model-based processes (Boehm 1988) and RUP (Kruchten 2003), ADM is iterative, meaning that activities are visited and revisited, with each subsequent iteration refining decisions made in earlier iterations. It recommends a sequence of steps and activities that should be taken, but leaves much up to the implementing organization—how to scope the project, how to make decisions within those phases, and so on. As shown in Figure 16-6, there are eight primary phases of the ADM, beginning with Architecture Vision (which is an effort to lay out

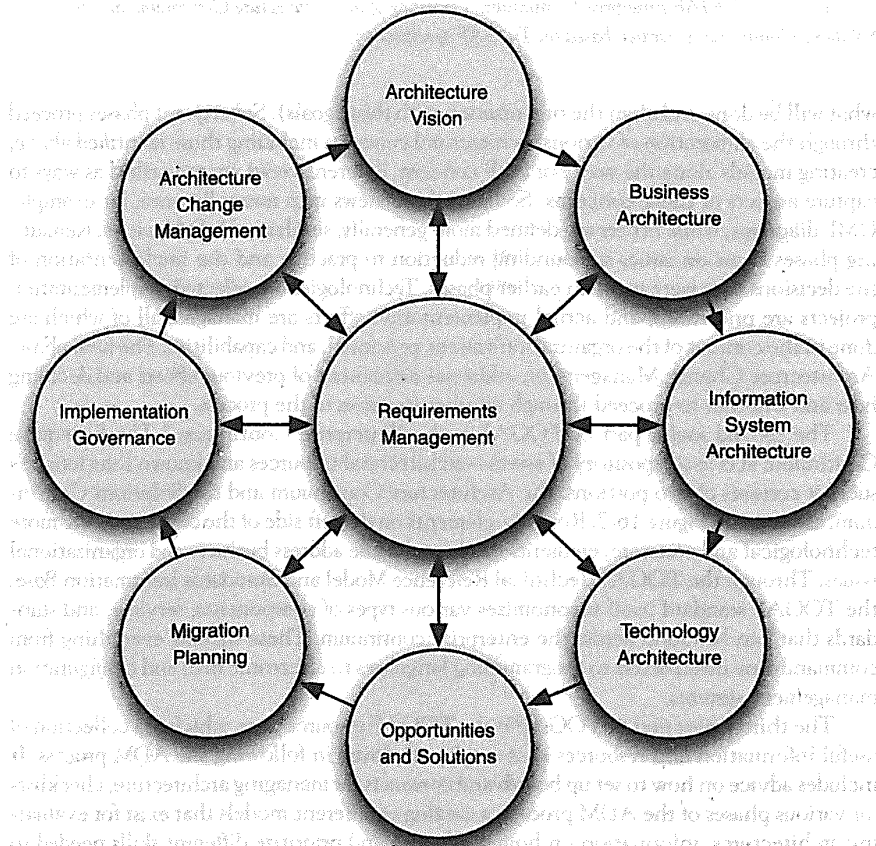


Figure 16-6.
TOGAF ADM
process main
steps (redrawn
based on
TOGAF
specification).

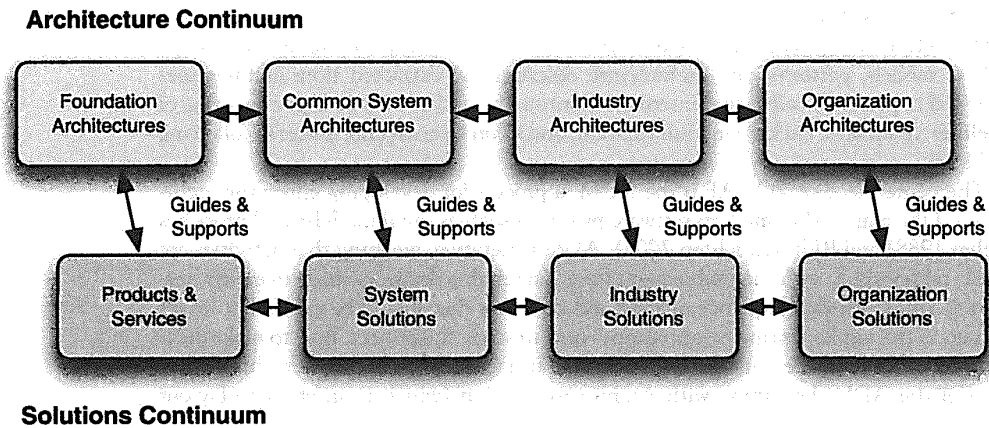


Figure 16-7. TOGAF Enterprise Continuum, consisting of the Architecture Continuum and the Solutions Continuum (redrawn based on TOGAF specification).

what will be done and align the organization with those goals). Subsequent phases proceed through the elaboration of various architectural concerns including those identified above, creating models along the way. For each concern, different views are prescribed as ways to capture aspects of those concerns. Some of those views map naturally onto, for example, UML diagrams, while others are defined more generally, similar to DoDAF views. Remaining phases focus on issues surrounding reduction to practice and the implementation of the decisions that were made in earlier phases. Technologies are selected, implementation projects are prioritized, and actual implementation efforts are managed, all of which are done in the context of the organization's assets, personnel, and capabilities. The final phase, Architecture Change Management, addresses assessment of previous efforts and deciding how and whether to proceed through another iteration of the process.

The second major part of TOGAF is the "Enterprise Continuum." The Enterprise Continuum acts as a repository of assets—architectural resources and known solutions. As such, it consists of two portions: the Architecture Continuum and the Solutions Continuum, as shown in Figure 16-7. Roughly, elements on the left side of the continuum are more technological and concrete; elements on the right side address business and organizational issues. Through the TOGAF Technical Reference Model and Standards Information Base, the TOGAF standard itself taxonomizes various types of components, services, and standards that can be found across the enterprise continuum. These include everything from command-line interpreters to programming languages to electronic mail and configuration management systems.

The third major part of TOGAF is the TOGAF resource base, which is a collection of useful information and resources that can be employed in following the ADM process. It includes advice on how to set up boards and contracts for managing architecture, checklists for various phases of the ADM process, a catalog of different models that exist for evaluating architectures, information on how to identify and prioritize different skills needed to develop architectures, and much more.

TOGAF is distinguished from other approaches by its large size and broad scope; the summary in this section only covers the major elements of the standard. TOGAF focuses on *enterprise* architecture, which is different in character from the kind of software architecture discussed in this book. Enterprise architecture is as much about the architecture of an organization as the architectures of its products. Elements in TOGAF architectures tend to be organizations, people, and large-grained assets (applications, platforms, machines, enterprise-wide services, and so on). Otherwise, TOGAF collects a large and diverse set of best practices for architecture. It prescribes not only an architecture development process, but defines a wide variety of views and concerns, as well as solution elements that can be used to satisfy those concerns.

Takeaway Message. TOGAF complements other architectural approaches by taking into account more business and organizational concerns. Its high-level approach, focusing on enterprise architecture, means that it is best used for dealing with coarse-grained elements and assets, constructing entire information systems and not just software applications. Like DoDAF, TOGAF provides a substantial set of best practices to follow and checklists to use for assessment, but individual users are responsible for the details. Following TOGAF helps to ensure thoroughness, but not necessarily quality.

RM-ODP

RM-ODP (ISO/IEC 1998) is an ISO standard for describing open distributed processing (ODP) systems. Although the characteristics of ODP systems are described over several pages of the RM-ODP standard, they can generally be described as information systems developed to run in an environment of independent, heterogeneous processing nodes connected by a network. Primarily, the RM-ODP standard defines five viewpoints and describes associated viewpoint languages for documenting the architecture of an ODP system. These viewpoints are:

Enterprise: A viewpoint focusing on the system, its environment, and its surrounding context.

Information: A viewpoint focusing on information and information processing.

Computational: A viewpoint focusing on the architectural structure of the system and how different components or objects are distributed across hosts.

Engineering: A viewpoint focusing on the infrastructure required to support distribution.

Technology: A viewpoint focusing on technology choices for the system.

These viewpoints (and their associated languages) have various levels of concreteness. The computational and engineering viewpoints are more concrete and thus more constrained because they have the goal of helping to guarantee interoperability and portability of components. The enterprise and information viewpoints are more general, and therefore their languages are less constrained.

Figure 16-8.
Sample
RM-ODP
enterprise view.

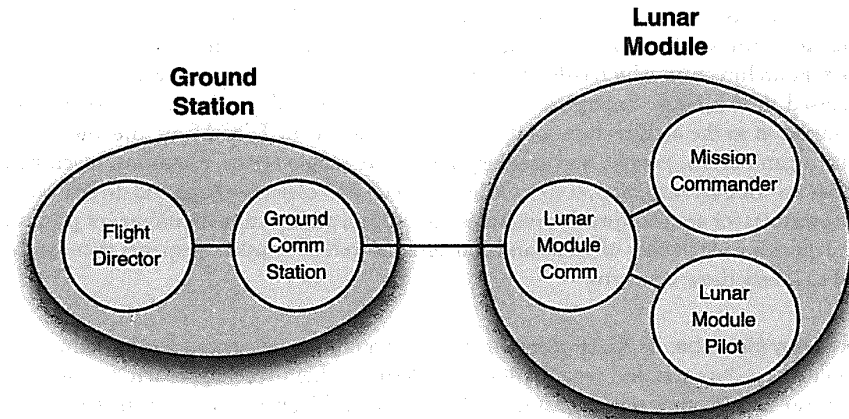


Figure 16-8 shows a sample RM-ODP enterprise view. Enterprise view elements include personnel and high-level application components, reminiscent of the kinds of elements addressed by TOGAF. The enterprise viewpoint looks at the system holistically, in terms of overall scope and its place inside the larger organization. Enterprise views can also describe agency boundaries—boundaries of responsibility or control.

In contrast, Figure 16-9 shows an example RM-ODP engineering view. This view shows how a ground system might communicate with a Lunar Lander via a complex communication channel implemented by two communicating network stacks. Here, the elements are more technologically oriented, and human elements do not appear. In general, both these views would be accompanied by text explaining the diagrams, the various elements, and their interconnections. The notation used above, with ovals and interconnections, is used in the RM-ODP specification but is not canonical: Many different notations (including, for example, UML) could be used to draw these diagrams.

Like DoDAF and TOGAF, RM-ODP focuses on *what* kinds of things should be modeled in an architecture, but not specifically *how* those things should be modeled. RM-ODP goes a step further than these two standards, however: Instead of prescribing a particular notation for each viewpoint, the RM-ODP specification prescribes a number of characteristics that a notation describing the viewpoint must have. Therefore, RM-ODP viewpoint languages are not languages in the traditional sense, but rather sets of requirements for languages. RM-ODP does include a separate specification describing, at a very high level, how various formal specification languages [for example, LOTOS (ISO 1989), Z (Spivey 1988), or SDL-92 (Telecommunication Standardization Sector of ITU 2002)] could be used to describe architectural semantics for RM-ODP architectures. Here, it lists an RM-ODP concept, and then identifies a formal specification language concept that could be used to implement that concept. This level of correspondence is not enough to give users serious guidance as to how to leverage these formal specification languages to implement RM-ODP.

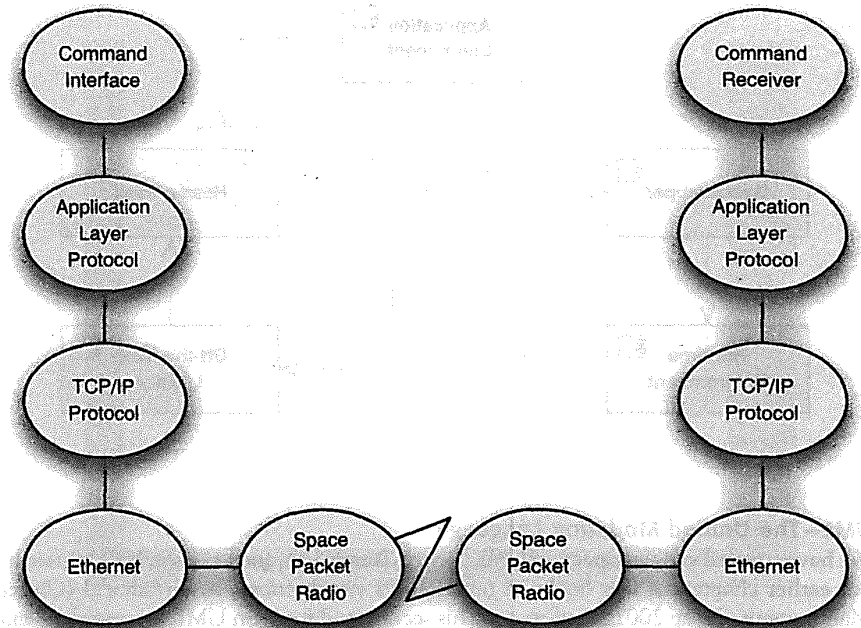


Figure 16-9.
Sample
RM-ODP
engineering view.

Takeaway Message. As a standard, RM-ODP is very similar to DoDAF, although the number of viewpoints is more limited and focused. Like DoDAF, an architecture specified according to the RM-ODP standard may or may not achieve certain qualities; the primary value added by RM-ODP is to ensure that architects think about and document certain aspects of architecture that impact various qualities—distributability, interoperability, and so on. It is eventually up to the architects to actually ensure those qualities. Although some mappings between RM-ODP concepts and formal methods are specified, there is a large gap between them that must be bridged by individual users. This gap is too large to assert that RM-ODP is particularly supportive of formal analysis activities.

16.2.2 Notational Standards

Notational standards define standard notations for writing down design decisions. Notational standards can be used to realize/implement certain elements of conceptual standards; for example, a notation such as UML can be used as a way of documenting the various DoDAF views. In a way, any codified architecture description language, such as those surveyed in Chapter 6, could be viewed as a de facto notational standard. In this section, we examine notations that have actually gone through a standardization process or committee and that have gained notoriety or use because they are standards.