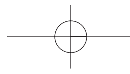
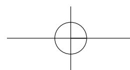


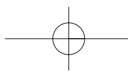
ONE

Introduction

So you want to become a software architect? Or perhaps you are already a software architect, and you want to expand your knowledge of the discipline? This is a book about achieving and maintaining success in your software career. It is also about an important new software discipline and technology, software architecture. It is not a book about getting rich in the software business; our advice helps you to achieve professional fulfillment. Although the monetary rewards are substantial, often what motivates many people in software architecture is being a continuous technical contributor throughout their career. In other words, most software architects want to do technically interesting work, no matter how successful and experienced they become. So the goal of this book is to help you achieve career success as a software architect and then maintain your success.

In this book we cover both heavyweight and lightweight approaches to software architecture. The role of software architect has many aspects: part politician, part technologist, part author, part evangelist, part mentor, part psychologist, and more. At the apex of the software profession, the software architect must understand the viewpoints and techniques of many players in the IT business. We describe the discipline and process of writing specifications, what most people would consider the bulk of software architecture, but we also cover those human aspects of the practice which are most challenging to architects, both new and experienced.





So what does a software architect do? A software architect both designs software and guides others in the creation of software. The architect serves both as a mentor and as the person who documents and codifies how tradeoffs are to be made by other software designers and developers. It is common to see the architect serve as a trainer, disciplinarian, and even counselor to other members of the development team. Of course, leadership by example will always remain the most effective technique in getting software designers and developers on the same page.

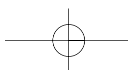
1.1 ADVICE FOR SOFTWARE ARCHITECTS

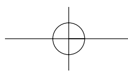
"Success is easy; maintaining success is difficult."—J.B.

If you have a focus for your career, gaining the knowledge you need in order to advance can be relatively easy. For software professionals, simply building your expertise is all that is needed in most corporate environments. For example, we often ask software people what books they have read. In the West, most professionals are familiar with design patterns (see Section 1.3). And many have purchased the book by Erich Gamma and co-authors that established the field of design patterns [Gamma 95]. Some have even read it. However, it always surprises us how few people have read anything further on this important topic.

For software architect books, the situation is even worse. Possibly the reason is that there are fewer popular books, but more likely it is that people are not really focused on software architecture as a career goal. In this book series, by publishing a common body of knowledge about software architecture theory and practice, we are eliminating the first obstacle to establishing a software architecture profession. However, making this information available does not automatically change people's reading habits.

So, if the average software professional only reads about one book per year, just think what you could do in comparison. If you were to read three books on design patterns, you would have access to more knowledge than the vast majority of developers on that important topic. In our own professional development, we try even harder—at least a book each month, and if possible, a book every week. Some books take longer than a week—for example, the 1000-page book on the Catalysis Method [D'Souza 98]. In our opinion, it contains breakthroughs on component-oriented thinking, but so few people are likely to read it thoroughly (except software architects), that it becomes a valuable intellectual tool





for making you (the reader) a thought leader, as the entire industry moves through the difficult transition to component-based development.

> *"Particularly for social systems, it's the perceptions, not the facts, that count" [Rechtin 97].*

Getting ahead on book reading is a clearcut way to differentiate yourself from the software masses. Converting your book learning to real-world success is also straightforward. You can apply your knowledge on your current projects. You can convert your knowledge into briefings and tutorials that put you in visible leadership and teaching roles. You can share you knowledge at conferences and professional groups. And you can write. The key transition that leads to success starts with sharing your knowledge one-to-one (i.e., inefficiently) and proceeding to share with many at a time. In our own careers, when we began to share knowledge in one-to-many situations, the appearance of success came with it. Since, for most people, appearance is reality, success is easy to attain. The much more difficult challenge is maintaining success, once you've achieved it.

Word of Caution

The software architecture career path is a difficult one for many reasons. While becoming a competent software architect can be difficult, maintaining your skills is usually even harder. Here are some key reasons why the architecture career is difficult:

- Nascent Body of Knowledge
- Confusion and Gurus
- Professional Jealousy
- The Management Trap
- The Software Crisis

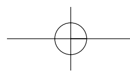
We discuss each of these in the subsections that follow.

Nascent Body of Knowledge

First of all, the body of software architecture knowledge is not well established. Software architecture is a relatively new field of computer science. Not much software architecture is taught in schools. Academics have not yet sorted out the fundamentals; there is still much discussion and disagreement on the basics.

However, many practicing software architects believe that sufficient knowledge does exist. The practice of software architecture is much more mature than many will admit. Hopefully, you will gain this understanding, too, after reading further.

ch01.qxd 9/20/00 12:45 PM Page 4



In the absence of widespread agreement about software architecture theory, you have to be your own expert. You have to acquire your own set of knowledge and a strong set of beliefs about how to do software right. No one book or software method will give you everything that you need to be an effective software architect.

"Technical problems become political problems" [Rechtin 97].

Confusion and Gurus

Many published software approaches claim to provide the benefits of software architecture, but most of them can't deliver on their promises. In fact, the software industry has created many technology fads and trends, on the basis of incomplete principles. When these approaches are applied in practice, software projects fail. And guess what? The overwhelming majority of corporate development projects do fail—by being cancelled, from overspending, or for underdelivery.

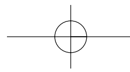
These failures are characteristic of a vast corporate software market, populated with companies that are struggling to deliver their internal software. New products and software development ideas are constantly being produced, in a never-ending attempt to meet the needs of the struggling software masses. Consequently, despite all the failures, the software products industry has thrived.

As a software architect, you have to be an evangelist and leader for your software team. From the myriad of conflicting software approaches and products you need to sort out what works and what does not. This is not easy, because a tremendous onslaught of marketing information generated by vendors and industry experts tends to contradict your architectural messages. It is your fate to have your architectural decisions frequently contradicted and obsolesced by the commercial software industry. One of your key skills as an architect is to make sound decisions that can survive the ravages of time and commercial innovation.

Professional Jealousy

The more successful you become, the more some people will resent your success. Many software professionals are genuinely nice people. But many people in our profession have large egos. We all have egos that can be abrasive, but whether you intend to compete on the basis of ego or not, professional competition can create serious problems in software organizations and in your career, unless you are careful.

ch01.qxd 9/20/00 12:45 PM Page 5



"Challenge the process and solution, for surely someone else will" [Rechtin 97].

Professional jealousy is a factor that you will have to watch for vigilantly. You must learn to conduct yourself with a certain degree of humility and be prepared to defend yourself when necessary. Never take any comment personally; it's always a mistake. Consider a situation where you are meeting someone for the first time and they appear to be acting quite rudely. In the eyes of people who are have known them for an extended period of time, they may very well be acting in their usual manner.

The Management Trap

As you become more successful in your software career, you may be joining the ranks of management, since most companies organize around a single management ladder. If you are good at what you do, it is natural for management to want you to mentor and supervise other people doing it, too. The company can try to get the productivity of several good performers based upon your experience.

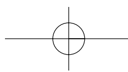
As your administrative responsibilities increase, your time to perform technical work can decrease dramatically. Because you spend less time on technical tasks and on maintaining your technical skills, you can lose your technical edge. If you chose a software career because you enjoyed technical work, you can lose one of your most important motivations for your work.

Being a software architect is quite different from being a manager. A software architect is a direct technical contributor, whereas a manager contributes indirectly by coordinating the actions of other people. Together, managers and architects make highly effective leadership teams. In our experience, combining the two roles can work only temporarily.

As you advance as a manager, eventually a superior will tell you to stop touching the keyboard (i.e., programming).

You as a software architect can avoid becoming a manager if you establish a personal professional policy. If you don't want management duties, you must learn how to say so. For many of us, one of the most difficult transitions is learning how to say "No." For example, you have to avoid lateral promotions that lead to management and administrative roles.

In some organizations you will become trapped in a management role, because the company does not have a technical ladder. At a certain level of



seniority (typical of software architects), you may be surprised, one day, to find yourself assigned responsibilities on the management organization chart. Once this is decided, it is very hard to reverse. The best approach is to declare your expectations (e.g., for technical assignments) when you first take the job. And repeat your policy often.

Defining Software Architecture

An increasing number of software professionals are claiming the title: software architect. In our opinion, very few of these people understand what software architecture is.

Have you ever been involved in a discussion of the question: "What is architecture?" The term "architecture" is one of those most often misused. Below we describe one of the common misuses; then we answer the question "What is architecture?" with a conceptual standard that is in widespread use today (see Section 1.2).

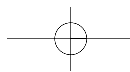
Misuse of the Term "Architecture"

Too often, architectures are used as sales tools rather than technical blueprints. In a typical scenario, a fast-talking technical manager (the "architect") presents a few high-level viewgraphs to convince you of the greatness of his product or system. This is a presentation of a marketing architecture. Most marketing architectures are directed externally at customers and not at software developers. Marketing architectures are fine for advertising the features of commercial products, but they provide only limited technical information for developers.

The problem with marketing architectures is that they are decoupled from the development process. The so-called architect is a manager who delegates most technical details to individual developers. Unless the architect manages the computational model (including subsystem interfaces), the architecture is unlikely to deliver any real technical benefits. Architectural benefits that are easily compromised include system adaptability (for new business needs) and system extensibility (for exploitation of new technologies).

Despite the many competing definitions, experts emphasize the importance of architecture, especially for component-based applications. As component reuse and software complexity increase, architecture is growing dramatically in importance. In subsequent sections we discuss several architecture-centered approaches, which support business change, technology innovation, and design reuse. Reuse of architectural designs benefits component reuse, because design reuse is often much more effective than software reuse alone.

ch01.qxd 9/20/00 12:45 PM Page 7



Before Architecture

High-quality, flexible software is one goal of architecture-centered development. In recent years, popular development approaches assumed that *bad software is better*. In other words, getting software delivered quickly is better than delivering quality software which supports change and reuse. Well-known process models and vendor regimes are founded on the *bad-is-better* principle.

Architecture-centered approaches accommodate reuse and change more effectively, because there is a planned system organization, specifically designed for these purposes, i.e., the system architecture. In our opinion, the practice of software architecture is essential for component-based development. *Bad is better* was the thesis; software architecture is the antithesis.

Of course, we do not want to lose the inherent benefit of *bad is better*, i.e., rapid delivery. Architecture-centered approaches utilize several techniques, including pragmatism, architecture planning, and architecture reuse, which jointly support increased productivity, reduced risk, and minimum time-to-market.

The Software Crisis

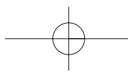
Many of us have serious misconceptions about the capabilities of current software approaches. Based upon surveys of corporate software projects in the United States, the realities of software development are as follows [Brown 98]. About one-third of all software projects are cancelled. Average projects expend twice as much budget and schedule as initially planned. After delivery, the majority of systems are considered unsuccessful because they have far fewer capabilities than expected. Modification and extension of systems are the most expensive cost drivers and very likely to create new defects. Overall, virtually all application software projects produce *stovepipe systems,* brittle software architectures that underperform on requirements.

The software crisis in corporate development became apparent decades ago, when procedural software technologies were popular. Subsequent, object-oriented approaches (such as the Object Modeling Technique) have been equally unsuccessful for corporate developers. These outcomes have been repeatedly confirmed by research [Brown 98].

Three key factors are exacerbating the software crisis:

- requirements change
- commercial innovation
- distributed computing

ch01.qxd 9/20/00 12:45 PM Page 8



A significant part of the problem is rising user expectations. User requirements for systems have increased much faster than corporate developers' capability to deliver. Requirements changes are more frequent, as businesses maneuver for competitive advantage with strategic corporate software.

Another confounding factor is the destabilizing force of accelerating technology innovation, in both commercial software and hardware platforms. Corporate developers have difficulty finding compatible configurations of software products and are forced to upgrade configurations frequently as new products are released. Software maintenance due to technology upgrades is a significant corporate cost driver.

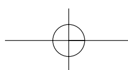
Owing to predominance of the Internet and geographically diverse enterprises, distributed computing is an essential feature of many new applications. Traditionally, software designers assumed homogeneous configurations, centralized systems, local communications, and infrequent failures. Today's highly distributed enterprises require heterogeneous hardware/software, decentralized legacy configurations, and complex communications infrastructure. The resulting computing environments have frequent partial system failures. Distributed computing reverses many key assumptions that are the basis for procedural and object-oriented software development.

The software industry has established object orientation (OO) as the mainstream technology. OO is the technology adopted by new corporate development projects because it is universally supported by software tool vendors. Masses of legacy programmers are training for object-oriented development (e.g., C++ and the Java programming language) as corporations create new strategic systems. Unfortunately, these developers and corporations are likely to become the next generation of disillusioned participants in the software crisis. However, the organizations that survive and thrive with this technology, must use it in sophisticated new ways, represented by componentware.

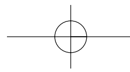
1.2 Software Architecture as a Discipline

As a professional discipline, software architecture has at least a dozen schools of thought. Some of the major schools of thought include:

- Zachman Framework [Zachman 97]
- Open Distributed Processing (ODP) [ISO 96]



ch01.qxd 9/20/00 12:45 PM Page 9



- Domain Analysis [Rogers 97]
- Rational's 4+1 View Model [Booch 98]
- Academic Software Architecture [Bass 98]

Alternative architecture approaches share concepts and principles, but their terminologies differ greatly. Each architecture school is relatively isolated from the others. In the literature of any given school, perhaps one or two other schools are acknowledged, however briefly. None of the schools appear to make any significant use of the results of the others. Since the terminology between these groups varies significantly, communication is difficult, especially between practitioners using different architecture approaches. Upon further study, we find that the goals of each school are quite similar, and each school has some unique value to offer.

In addition to these schools, there are many vendor-driven approaches that are tied to specific product lines, such as Netscape ONE, Sun Enterprise JavaBeans, and Microsoft BackOffice. In fact, every vendor appears to have a unique architectural vision for the future founded upon its own product lines.

Many vendors actually have minimal understanding of application architecture. Thus, I focus here on those approaches which consider key application drivers with appropriate product support for underlying capabilities.

Architecture Approaches

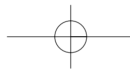
Here is a brief tour of the major schools of software architecture thought.

Zachman Framework

Derived from IBM research and practice, the Zachman Framework is a traditional architecture approach; i.e., it is decidedly non-OO. The Zachman Framework is a reference model comprising 30 architecture viewpoints. The refer- ence model is a matrix, which intersects two paradigms: journalism (who, what, when, why, where, and how) and construction (planner, owner, builder, designer, subcontractor). Architects choose from among these viewpoints to specify a system architecture.

Open Distributed Processing

A formal standard from ISO and ITU (telecommunications), Open Distributed Processing (ODP) defines a five-viewpoint reference model (enterprise, information, computational, engineering, and technology). ODP defines a comprehensive set of terminology, a conformance approach, and viewpoint correspondence rules for traceability. The product of seven years of standards



work, ODP is a recent adoption that fully supports OO and component-based architecture. In fairness, I should note that ODP is my primary approach to software architecture.

Domain Analysis

A process for the systematic management of software reuse, domain analysis transforms project-specific requirements into more general domain requirements for families of systems. The requirements then enable the identification of common capabilities, which are used as the basis for horizontal frameworks and reusable software architectures. An important capability of this approach is the definition of robust software designs, which are relatively resistant to requirements and context changes.

4+1 View Model

A four-viewpoint approach is under development by Rational Software. The viewpoints include: logical, implementation (formerly "component"), process (i.e., runtime), and deployment. The "+1" denotes *use case* specifications supporting requirements capture. This approach is closely aligned with the Unified Modeling Language and the Unified Process.

Academic Software Architecture

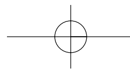
Academic software architecture comprises a community of computer science researchers and educators constituting an academic field. Their educational efforts are focused on basics and fundamentals. In their research contributions, this community avoids proven architectural standards and practices in order to achieve originality, theoretical formality, and other academic goals.

Common Principles

It is often said that the principles of software are simple. For example, let's consider (1) simplicity and (2) consistency. Architects agree that managing complexity (i.e., achieving simplicity) is a key goal, because it leads to many architectural benefits, such as system adaptability and reduced system cost. For example, a simpler system is easier to test, document, integrate, extend, and so forth.

> *"Explore the situation from more than one point of view. A seemingly impossible situation might become transparently simple" [Rechtin 97].*

ch01.qxd 9/20/00 12:45 PM Page 11



Simplicity is most necessary in the specification of the architecture itself. Most architectural approaches utilize *multiple viewpoints* to specify architecture. Viewpoints separate concerns into a limited set of design forces, which can be resolved in a straightforward and locally optimal manner.

Consistency enhances system understanding and transfer of design knowledge between parts of the system and between developers. An emphasis on consistency contributes to the discovery of commonality and opportunities for reuse. Architects agree that unnecessary diversity in design and implementation leads to decidedly negative consequences, such as brittle system structure.

Architecture Controversies

The principal disagreements among architecture schools include: (1) terminology, (2) completeness, and (3) a priori viewpoints.

Architects disagree on terminology due to their backgrounds or schools of thought. For example, when discussing software interfaces, the consistency principle is variously called: standard interfaces, common interfaces, horizontal interfaces, plug-and-play interfaces, and interface generalization. We can also argue that *variation-centered design* (from design patterns) and *component substitution* are largely based upon consistent interface structure.

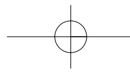
Unnecessary diversity of terminology leads to confusion, and sometimes to proprietary advantage. Some vendors and gurus change terminology so frequently that keeping up with their latest expressions becomes a time-consuming career.

Differences in terminology lead to miscommunication. In contrast, some distinct areas of disagreement among architecture schools can't be resolved through improved communications alone.

The notion of *complete models* is promoted by legacy OO approaches (e.g., OMT), the Zachman Framework school, and various others. These groups have promoted a vision that complete models (describing multiple phases of development) are a worthwhile goal of software development projects. Other schools would argue that multiple models are not maintainable, that unnecessarily detailed models are counterproductive, and that architectural significance should be considered when selecting system features for modeling.

These contrary notions can be summarized in terms of the principle of *pragmatism*. We side with the pragmatists for the above reasons and because most software systems are too complex to model completely (e.g., multithreaded

ch01.qxd 9/20/00 12:45 PM Page 12



distributed computing systems). Pragmatism is a key principle to apply in the transition from document-driven to architecture-centered software process.

The selection of architecture viewpoints is a key point of contention among architecture schools. Some schools have preselected a priori viewpoints. Some schools leave that decision to individual projects. The Zachman Framework is an interesting case, because it proposes 30 viewpoints, from among which most projects select groups of viewpoints to specify.

Variable viewpoints have the advantage that they can be tailored to address the concerns of particular system stakeholders. Predefined viewpoints have the advantage that they can accompany a stable conceptual framework and a well-defined terminology, as well as predefined approaches for resolving viewpoint consistency and architecture conformance.

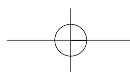
Innovative Software Architecture

There are many active and successful schools of software architecture thought. Software architecture is a discipline unified by principles, but divided by terminology. The various architecture schools can be viewed as different branches of an evolutionary progression.

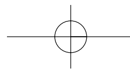
The Zachman Framework has evolved from the traditional non-OO approaches. ODP is an outgrowth from object-oriented and distributed-computing paradigms that has achieved stability, multiindustry acceptance, and formal standardization. Both Zachman and ODP approaches have enjoyed significant success in production-quality software development. Domain analysis has demonstrated its worth in defining robust, domain-specific software architectures for reuse. The 4+1 View Model is an approach undergoing development, in parallel with the Unified Process.

All of the above can be described as innovative software architecture approaches. They are being applied in practice, based upon various levels of proven experience. Academic research in software architecture is defining a baseline for architecture knowledge that resembles a lowest common denominator of the above approaches. Fortunately, the academic community has legitimized the role of the software architect, regardless of whether their guidance is useful to innovative architects.

In our opinion, software architects should have a working knowledge of the innovative approaches described above. In addition, they should utilize one of the product-quality architecture frameworks in daily practice. Component architecture development is a challenging area, requiring the best of stable conceptual frameworks supporting sound architectural judgement.



ch01.qxd 9/20/00 12:45 PM Page 13



The Architecture Paradigm Shift

The nature of information systems is changing from localized departmental application to large-scale global and dynamic systems. This trend is following the change in business environments toward globalization. The migration from relatively static and local environments to highly dynamic information technology environments presents substantial challenges to the software architect (Figure 1.1).

A majority of information technology approaches are based upon a set of traditional assumptions (Figure 1.2). In these assumptions the system comprises a homogeneous set of hardware and software which is known at design time. A configuration is relatively stable and is managed from a centralized system management configuration. Communications in traditional systems are relatively predictable, synchronous, and local. If the state of the system is well known at all times and the concept of time is unified across all the activities, another key assumption is that failures in the system are relatively infrequent and, when they, do occur, are monolithic. In other words, either the system is up or the system is down.

In the building of distributed application systems, most of the assumption are reversed. In a distributed multiorganizational system it is fair to assume that the hardware and software configuration is heterogeneous. The reason is that different elements of the system are purchased during different time frames by

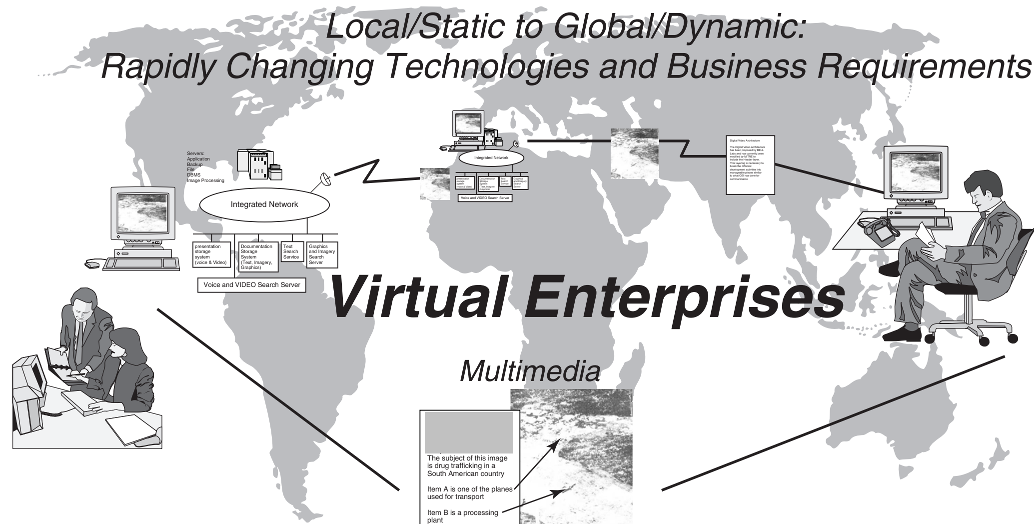
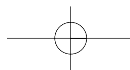
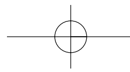


FIGURE 1.1 Virtual Enterprise Paradigm Shift





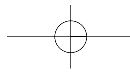
TRADITIONAL SYSTEM ASSUMPTIONS
- Homogeneous hardware/software
- Stable, centrally managed configuration
- Synchronous and local: processing, state, time, and communications
- Infrequent, monolithic failures

DISTRIBUTED SYSTEM ASSUMPTIONS
- Homogeneous hardware/software – evolving configurations
- Remote, autonomous processing
- Distributed, replicated, non-uniform: state and time
- Asynchronous, insecure, variable: communications
- Frequent partial system failures

FIGURE 1.2 Traditional and Distributed-Systems Assumptions

different organizations and many of the decisions are made independently. Therefore in a typical configuration you have a variety of information technology. It is also the case that hardware and software configurations are evolving. Occurring within any organization are turnover in employees and evolution of business processes. The architecture of the organization impacts the architecture of the information technology. As time progresses, new systems are installed, systems are moved, new software is acquired, and so on. When multiple organizations are involved, these processes proceed relatively independently, and the architect must accommodate the diverse evolving set of configurations.

In distributed systems, the assumption is that there is remote processing at multiple locations. Some of this remote processing is on systems that were developed independently and therefore have their own autonomous concept of control flow. This reverses the assumption of localized and unified processing resources. There are some interesting implications for the concepts of state and time. The state of a distributed system is often distributed itself. The state information may need to be replicated in order to provide efficient reliable access at multiple locations. It is possible for the distributed state to become nonuniform in order to get into error conditions where the replicated state does not have the desired integrity and must be repaired. The concept of time-distributed systems is affected by the physics of relativity and chaos theory. Electrons are traveling near the speed of light in distributed communication systems. In any large system there is a disparity between the local concepts of time, in that this system can only have an accurate representation of partial ordering of operations in the distributed environment. The total ordering of operations is not possible because of the distances between information process. In addition, distributed communications can get quite variable and complex. In a distributed system there are various qualities of service which communications



systems can provide. The communications can vary by timeliness of delivery, the throughput, the levels of security and vulnerability to attack, the reliability of communications, and other factors. The communications architecture must be explicitly designed and planned in order to account for the variabilities in services.

Finally, the distributed system has a unique model of failure modes. In any large distributed system components are failing all the time. Messages are corrupted and lost, processes crash, and systems fail. These kinds of failures happen frequently and the system must be architected to accommodate for them.

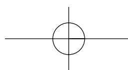
In summary, distributed processing changes virtually all of the traditional system assumptions that are the basis for most software engineering methodologies, programming languages, and notations. To accommodate this new level of system complexity, architects have three new needs.

First, architects need the ability to separate complex concerns, in particular to separate concerns about business-application functionality from concerns about distributed-system complexity. Distributed computing is a challenging and complex architectural environment unto itself. If systems are built with traditional assumptions, architects and developers are likely to spend most of their time combating the distributed nature of real-world applications. Problems and challenges of distributed computing have nothing to do fundamentally with business-application functionality.

The purpose of information technology is to establish new business processes. By separating concerns, we can focus on the business functionality that is the true purpose of the information system. Ideally, architects would like to separate distributed-system issues into categories of design, where the majority of components are purchasable as commodity communication infrastructure.

Object-oriented architects also need the ability to future-proof the information systems that they are planning. It is important to accommodate commercial technology evolution, which we know is accelerating and beginning to provide substantial challenges for architects and developers. Future-proofing also requires the ability to adapt to new user requirements, since requirements do change frequently and account for a majority of system software cost over the life cycle. It is important to plan information systems to support the likely and inevitable changes that users will require in order to conduct business.

A third need for object-oriented architects is the ability to increase the likelihood of system success. Corporate developers to date have had a very poor track record of creating successful systems. The object-oriented architect



is responsible for planning systems with the maximum probability of delivering success and key benefits for the business. Through proper information technology planning, we believe that it is possible to increase the likelihood of system delivery on time and on budget.

In confronting these three needs, authorities in software engineering and computer science tend to agree that architecture is the key to system success. Authorities in areas ranging from academia to commercial industry are declaring that software architecture is essential to the success and management of information systems. There is a long and growing list of software authorities who have come to this conclusion. Unfortunately, it is not always clear to everyone what software architecture truly is. In order to provide clarification, we need to take a look at some of the reference models which provide definitions of software and systems architecture (Figure 1.3).

The needs that we are discussing have been thoroughly considered by many authorities. There are two leading meta-architecture frameworks that guide the development of software system architecture. One of the popular frameworks originated at IBM and is called the Zachman Framework. The Zachman Framework predated the popularity of object orientation and took the perspective that separating data from process. In the Zachman Framework there are six information system viewpoints as well as five levels of design abstraction. The original Zachman Framework published in 1987 contained

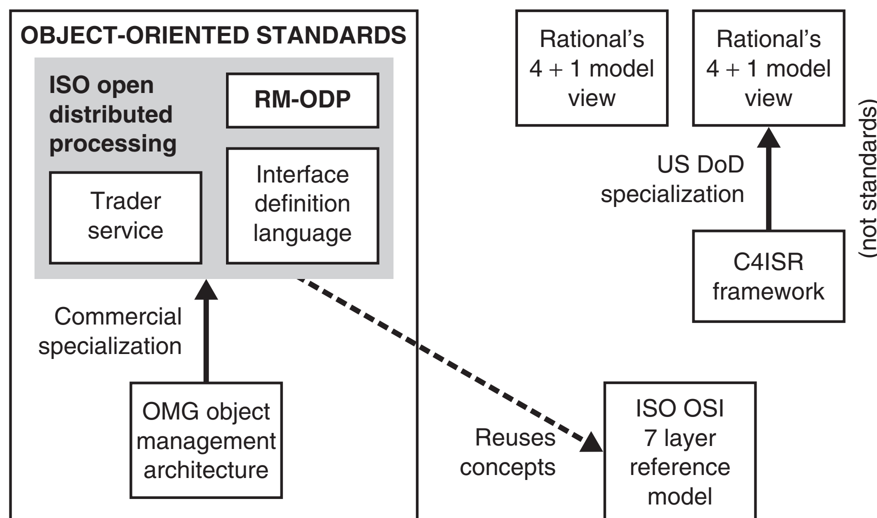
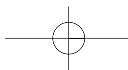
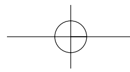


FIGURE 1.3 Software-Intensive Systems Architecture Reference Models



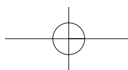
ch01.qxd 9/20/00 12:45 PM Page 17



viewpoints for the network, the data, and the process of the information system [Zachman 87]. A subsequent revision introduced three additional viewpoints. The current framework resembles the set of traditional journalistic questions, which include who, what, when, why, where, and how. Each viewpoint in the Zachman Framework answers a chief set of questions to assure that a complete system engineering architecture is created.

The Zachman Framework formed a matrix of architecture descriptions which are also organized in terms of levels. There are five levels of description above the information system implementation. They range from architectural planning done by individual programmers at the finest grain to the overall enterprise requirements from the investors' perspective of the information system. In total, the Zachman Framework identifies 30 architectural specifications, which provide a complete description of the information system. In practice no real-world project is capable of creating these 30 or more detailed plans and keeping them all in synchronization. When the Zachman Framework is applied, systems architects partition the viewpoint into various categories and create architectural specifications that cover all of the different Zachman descriptions without having to create the large number of specification documents that the Zachman Framework implies. One example is a very successful architecture initiative by the United States Department of Defense called the C4ISR architecture framework, where C4ISR stands for Command and Control, Computers, Communication, Intelligence Surveillance, and Reconnaissance. The C4ISR architecture framework is used to describe DOD information technology at the highest echelons of the organization. The primary benefit in this case is that different service organizations and agencies can communicate their architectural plan through common-viewpoint description.

Beyond the Zachman Framework, object-oriented architects have discovered additional needs for defining computational architecture and other viewpoints which are not obvious applications of the Zachman principles. The international standards organization (ISO) has also considered these architectural issues. Recently completed is the ISO reference model for open distributed processing called RM-ODP. This model belongs to a category of ISO standards called open distributed processing (ODP). ODP is an outgrowth of earlier work by ISO in open systems interoperability. The Open Systems Interconnection (OSI) seven-layer reference model identified an application layer which provided minimal structure and guidance for the development of application systems. In fact, the seventh layer for applications groups remote procedure calls, directory services and all other forms of application level services within the same architectural category, not defining any particular structure or guidance for this significant category of functionality.



A Standard for Architecture

Among the various architecture approaches, there is a useful international standard that defines what information systems architecture means, the Reference Model for Open Distributed Processing (RM-ODP) [ISO 96]. We will cover it as one way to think about software architecture. This model is representative of mature software architecture practice today.

RM-ODP defines five essential viewpoints for modeling systems architecture:

- Enterprise Viewpoint
- Information Viewpoint
- Computational Viewpoint
- Engineering Viewpoint
- Technology Viewpoint

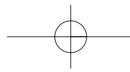
The five viewpoints provide a comprehensive model of a single information system.

An *enterprise* viewpoint contains models of business objects and policies. Enterprise policies include permissions, prohibitions, and obligations. An *information* viewpoint includes the definition of information schemas as objects. Three kinds of RM-ODP schemas include static, invariant, and dynamic. A *computational* viewpoint includes definitions of large-grained object encapsulations, including subsystem interfaces and their behaviors. These three viewpoints define architecture in a manner that makes distributed computing transparent. An *engineering* viewpoint exposes the distributed nature of the system. The distribution transparencies supported by infrastructure are declared explicitly. The allocation of objects onto processing nodes is also specified. RM-ODP defines a reference model of distributed infrastructure called a *channel* which is used to model all forms of middleware connections.

RM-ODP defines eight distribution transparency properties. It is interesting to note that only a handful of these properties are supported by major commercial infrastructures (without resorting to niche-market products). For example, CORBA products provide full support for access, location, and transaction transparency, with some support for failure and persistence transparency. Microsoft's Distributed Component Object Model (DCOM) provides support for persistence and transaction transparency, with limited support for the other properties.

Open distributed processing and its reference model are the result of ten years of formal standardization work at ISO. The reference model for open distributed processing is object oriented. It provides a referenced model that was

ch01.qxd 9/20/00 12:45 PM Page 19



intended to address three fundamental goals: (1) to provide a standard framework for further work and additional detailed standards under the open distributed processing initiative, (2) to provide a set of common terminology and concepts that could be applied for the development of product and application systems for open distributed processing, (3) to provide a guideline for object-oriented architects to specify software systems. This third purpose is directly relevant to the day-to-day practices of systems architects.

Open distributed processing includes several other standards which are significant (Figure 1.3). In particular, it has adopted the interface definition language from CORBA as a notation for a specified computational architecture. It also has a standard for the treasury service, which is the key directory service supporting the discovery of application functions in distributed systems. The trader service has subsequently been adopted as a commercial standard through the object management group. The group's object management architecture is a commercial specialization of open distributed processing.

All together, the OMG's consensus standards and the ISO open distributed processing form a set of software architecture standards that are useful intellectual tools for most software architects and developers.

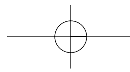
RM-ODP has three completed standards documents. Part one of the standards is a non-normative overview and summary of the overall concepts and terminology. All three parts of the adopted standard are cosponsored by the International Telecommunications Union ITU-T through their X.900 series. The cosponsorship of both ISO and ITU-T represents a broad international consensus on this guideline for object-oriented architecture.

Part two of the standard is the foundations document, comprising a glossary of standard terminology for object oriented distributed systems.

Part three of the standards is the architecture document. It defines the various viewpoints for object-oriented architecture along with their structuring rules and various open distributed processing functions which enable distributed computing.

Altogether, these three standards documents comprise less than 200 pages of documentation with the normative parts, part two and part three comprising about 100 pages. Even though this is a relatively short standard, it provides a great deal of valuable information. Many ISO standards are relatively inscrutable to the practicing software engineer; this standard is no exception. However, we believe that the effort to understand it is very worthwhile, given the challenges of distributed computing in business process change that need to be resolved.

Who supports RM-ODP? RM-ODP is the product of formal standards bodies including ISO and IEEE. The IEEE is an accredited standards organization



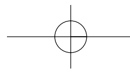
reporting to ISO; therefore, the IEEE is a voting participant and joint supporter of RM-ODP as well. RM-ODP is the formal standards basis for the object management group's object management architecture and all of the resulting technologies that the group has adopted which form the basis for distributed object computing and technologies that are available commercially. RM-ODP is also accurately used in several mission-critical industries which depend upon information technology for their income. In particular, RM-ODP is applied across the telecommunications industry through the telecommunications information network architecture consortium, and RM-ODP is actively used by telecommunication companies such AT&T, Lucent, and Nortel. In the telecommunications industry, information technology is their business, and distributed information systems success is essential to maintaining their competitive advantage.

Also applying ODP actively is the financial services industry. Companies such as Merrill Lynch, Morgan Stanley, and various mortgage lending organizations are applying RM-ODP to define new information systems concepts. The deployment of new information technologies is becoming one of the key competitive advantages that these companies have for creating new market channels to distribute and transact new financial instruments, and securities and perform other financial services. For these industries failure of information systems directly affects bottom-line profitability and is usually not an option. If these influential companies accept this architectural approach and apply it actively, can your organization afford not to consider its benefits?

The RM-ODP comprises five standard viewpoints. Each viewpoint is a perspective on a single information system (Figure 1.4). The set of viewpoints is not closed, so that additional viewpoints can be added as the needs arise. Another of their purposes is to provide information descriptions that address the questions and needs of particular stakeholders in the system. By standardizing five viewpoints, RM-ODP is claiming that these five stakeholder perspectives are sufficient for resolving both business functionality and distributed systems issues in the architecture and design of information systems. RM-ODP is an elegant model in the sense that it identifies the top priorities for architectural descriptions and provides a minimal set of traceability requirements which are adequate to assure system integrity.

The enterprise viewpoint of our RM-ODP takes the perspective of a business model. The enterprise models should be directly understandable by managers and end users in the business environment. The enterprise viewpoint assures that business needs are satisfied through the architecture and provides a description which enables validation of these assertions with the end users.

The information viewpoint defines the universe of discourse in the information system. The perspective is similar to the design information generated



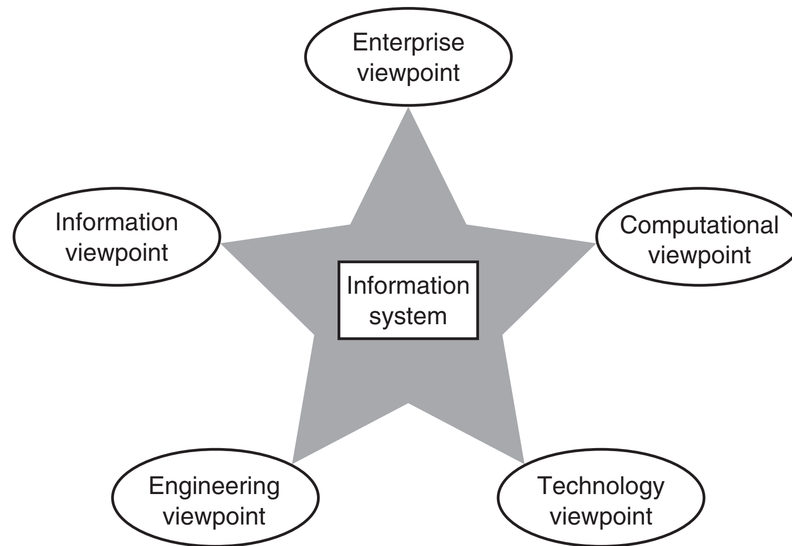


FIGURE 1.4 Architecture Viewpoint Perspectives

by a database modeler. The information viewpoint is a logical representation of the data and processes on data in the information system.

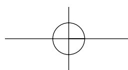
Each of the five RM-ODP viewpoints is object oriented, and they provide a complete model of the system from the given perspective. The information viewpoint is an object-oriented logical model of the information assets in the business and how these assets are processed and manipulative.

The computational viewpoint partitions the system into software components which are capable of supporting distribution. It takes the perspective of a designer of application program interfaces for componentware. The computational viewpoint defines the boundaries between the software elements in the information system. Generally, these boundaries are the architectural controls that assure that the system structure will embody the qualities of adaptability in management of complexity that are appropriate to meet changing business needs and incorporate the evolving commercial technology.

The engineering viewpoint of RM-ODP exposes the distributed nature of the system. Its perspective is similar to that of an operating system engineer who is familiar with the protocol stacks and allocation issues that are necessary to define the distributed processing solutions for the information system.

The fifth viewpoint is the *technology* viewpoint. It defines the mappings between the engineering objects and other architected objects to specific standards and technologies including product selections. The viewpoint is similar

ch01.qxd 9/20/00 12:45 PM Page 22



to that of a network engineer who is familiar with the protocol standards and products available commercially which are appropriate selections to configure the information system.

All five RM-ODP viewpoints are co-equal in the sense that they do not form levels of description; rather each viewpoint provides a complete model of the information system that is object oriented and corresponds to the other viewpoints. Each defines various constraints on the design of the information system that provide various architectural benefits for each of the system's stakeholders. The RM-ODP viewpoints enable the separation of concerns which divide the business and logical functionality of the system from the distributed computing and commercial technology decisions of the architecture.

The first three viewpoints identify informational and computational characteristics. The enterprise and information viewpoints are purely logical views of the business, represented as object-oriented models (Figure 1.5). The computational viewpoint is independent of the distribution of software modules, but it must define computational boundaries which are enabled for distribution. The CORBA IDL notation for specifying computational interfaces is appropriate for this purpose. IDL provides a way to define computational interfaces which are independent of the distribution and deployment issues in enterprise development. The first four viewpoints—enterprise, information, computational, and engineering—are independent of specific implementations. In other words, the majority of the architectural design is independent of the specific product selections which configure the system. This property of RM-ODP enables the evolution of technology components without impacting the overall architectural constraints defined in the first four viewpoints. The engineering viewpoint defines qualities of service and distribution transparencies which

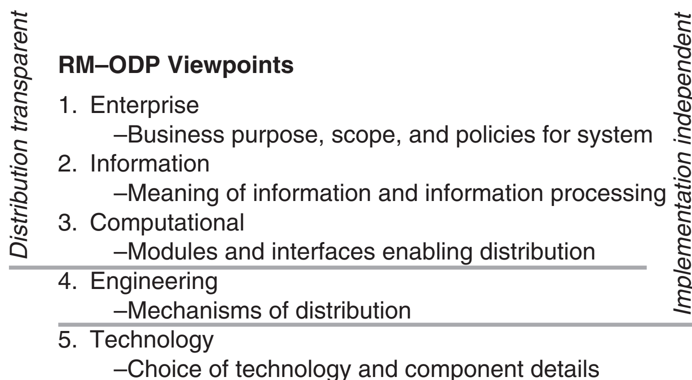
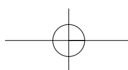
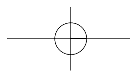


FIGURE 1.5 Characteristics of Architecture Viewpoints





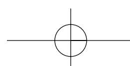
evolving technology selections must support. The terminology of RM-ODP assists in providing concise descriptions of these technology requirements.

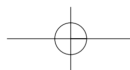
RM-ODP contains many terminology definitions which are useful concepts for object-oriented architects. Some of the key definitions in RM-ODP are the distribution transparencies. RM-ODP defines in distribution transparencies which specify the qualities provided by distributed computing infrastructure (Figure 1.6). Currently available commercial infrastructures provide some subset of these, such as location, and access transparencies provided by CORBA along with partial support for persistence in transaction transparency. Additional transparencies are available through niche-market products and through custom implementations which are enabled by proper architectural separation of infrastructure requirements from technology selections. Technologies which do not provide access transparency, such as Microsoft COM+ and the distributed computing environment, do not adapt well to the future evolution of distributed systems (Figure 1.7).

RM-ODP provides standard definitions for distributed infrastructure objects that enable abstract descriptions of engineering constraints. Figure 1.7 is an example of the engineering objects which RM-ODP defines. These engineering objects are capable of defining the characteristics of all forms of distributed infrastructure, including remote procedure calls, screening data interfaces, and asynchronous interfaces for signaling. Among the most important features of RM-ODP are its definitions supporting conformance assessment. After all, what is the purpose of architectural documentation unless we

Distributed Transparency	Architectural Guarantee
Access	Masks platform-protocol difference in data representation and invocation mechanisms
Failure	Masks failures and recoveries of other objects
Location	Masks the use of location information to find & bind to objects
Migration	Masks awareness of changes in location of the object from itself
Relocation	Masks changes in the location of an interface/service from clients
Replication	Masks the existance of replicated objects that support common states and services
Persistence	Masks activation and deactivation of objects (including the object itself)
Transaction	Masks coordination of activities to achieve consistency

FIGURE 1.6 Distribution Transparencies





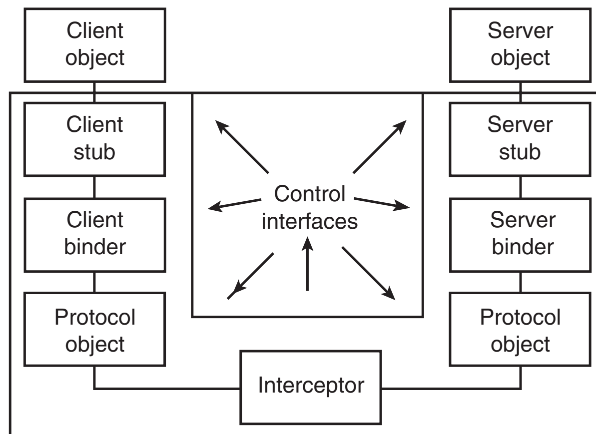


FIGURE 1.7 Distribution Channel Model

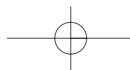
can assess conformance—that is, make sure that the implementation of the system corresponds to the written and intended architectural plans.

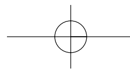
RM-ODP defines four categories of conformance and proceeds to specify how conformance is represented in an architectural plan. The first category is called *programmatic conformance.* This is the usual notion of behavioral testing of software interfaces. Many of the programmatic conformance points will occur in the computational viewpoint specification of RM-ODP based architectures.

Perceptual conformance includes testing at user interfaces in communications ports that represent external boundaries to the system. Usability and user interface testing can be defined through perceptual conformance assessment. *Interworking conformance* involves testing between systems implementations. It is not sufficient for individual systems to have programmatic conformance in order to guarantee interoperability. Interworking conformance includes interoperability testing between working implementations, which is an additional requirement beyond programmatic conformance.

Interchange conformance involves testing of the exchange of external media, such as disks and tapes. It assures that information that is stored on external media can be interpreted and assimilated in other systems that conform to the same standards. RM-ODP also defines correspondence requirements between the various viewpoints of application architecture. In general, the objects defined in each of the viewpoints do not have to be explicitly correspondent, because they represent independent description of the system representing various levels of granularity of descriptions and constraints.

Several key points of correspondence which must be assured. The computational viewpoint must support any dynamic behaviors that are specified in the





information viewpoints. The information viewpoint represents the information in the information system and its processing. Whenever a process occurs, it must be explicitly allocated to the internal operation of one of the computational modules or it must be explicitly allocated to a particular computational interaction—in other words, invoking a software interface to cause the processing of information. In addition, there is an explicit correspondence requirement between the computational and engineering viewpoints. In general, engineering objects outnumber computational objects, because the engineering viewpoint exposes the objects in the distributed infrastructure, which may be numerous. For every computational interface defined in the computational viewpoint, there must be an explicit correspondence to engineering interfaces in the engineering viewpoint objects. The computational boundaries must map onto distributed engineering objects so that the distribution strategy is clarified by the architecture.

Applications and Profiles

Open systems standards (such as RM-ODP) are purposefully generic so that they apply to all domains. To make standards deliver their benefits, a *profile* is required. A profile is an implementation plan for how the standard is applied within a context. Several profiles of RM-ODP are in use today.

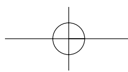
The 4+1 View Model is a viewpoint-based architecture approach supported by OO tools such as Rational Rose. The viewpoints include:

- Use Case View
- Logical View
- Process View
- Implementation View
- Deployment View

The *use case* view models enterprise objects through a set of scenarios. The *logical* view includes object models of packages, classes, and relationships. The *process* view represents control flows and their intercommunications. The *implementation* view defines the modular structure of the software. The *deployment* view identifies the allocation of software onto hardware. An architecture defined as a 4+1 View Model covers aspects of all 5 RM-ODP viewpoints.

RM-ODP is being applied in several industries, including financial services and defense. For example, the United States Department of Defense (DoD) has a profile of RM-ODP, called the Command, Control, Communications, Computers, Intelligence, Surveillance, and Reconnaissance Architecture Framework (C4ISR-AF). C4ISR-AF defines three viewpoints: operational

ch01.qxd 9/20/00 12:46 PM Page 26



architecture, system architecture, and technical architecture. An information viewpoint is also specified.

Before applying the framework, DoD services defined their architectures using disparate conventions. C4ISR-AF is currently used by all DoD services to describe their architectures. The framework is enabling technology exchanges across diverse system development programs. Reuse opportunities and common interoperability solutions are being identified and defined as a result.

Viewpoint Notations

Within each viewpoint, the RM-ODP approach uses formal notations (or specification languages) that support architecture description.

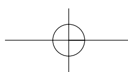
One of the most useful notations for specifying computational viewpoints is the ODP interface definition language (ODP IDL). ODP IDL is a related international standard that is identical to CORBA IDL. It enables the specification of object encapsulations that can be implemented on multiple infra structures, such as CORBA, Microsoft COMT, and the Adaptive Communication Environment (ACE). Since ODP IDL is programming a language independent, a single interface specification suffices to define interoperable interfaces for C, C++, Ada95, COBOL, Smalltalk, the Java programming language, and Microsoft IDL. These mappings are defined by open systems standards and supported by commercial products.

Another useful notation for describing architecture viewpoints is the Unified Modeling Language (UML). UML is an object-oriented notation recently adopted by the Object Management Group. UML is also supported by Microsoft in its respository and development environment technologies.

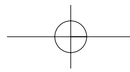
Although it is not widely publicized, RM-ODP is providing architectural benefits in multiple industries. RM-ODP is a formal standard that defines how to describe distributed OO architectures. In practice, RM-ODP's viewpoints, models, and transparency properties are useful conceptual tools for object-oriented architects.

1.3 DESIGN PATTERNS AND SOFTWARE ARCHITECTURE

We view software architecture as an eclectic practice, combining ideas from many areas of computer science and software engineering. Reuse of these ideas and existing knowledge is paramount to the effective practice of the architec-



ch01.qxd 9/20/00 12:46 PM Page 27



tural discipline. Luckily, the popular movement of design patterns has codified and documented a great deal of software knowledge for this purpose. We believe that software architects should also be pattern literate.

What the design patterns community has done is to make the reuse of lessons learned into a popular, trendy approach. Patterns represent a rejection of originality as a technical goal, including an active avoidance of the Not-Invented-Here (NIH) syndrome.

Design Patterns

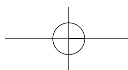
Design patterns are a significant extension to object-oriented paradigm. Design patterns are documented representations of software engineering knowledge. They are intended to capture expert-level knowledge and important lessons learned. Design patterns are a departure from previous object-oriented guidance in several respects. Patterns document essential design knowledge, transcending original object-oriented notions. Originally, object orientation was based upon modeling of the natural world as objects. To design effective software systems, more sophisticated structures are needed that are unique to software.

Design patterns have more stringent requirements for documenting knowledge. Design patterns should represent *proven solutions*, not merely wishful thinking about how software should be done. This concept is embodied in the so-called *rule of three.* Informally, the rule of three states that: "A single design occurrence is an event, two occurrences are a coincidence, and three occurrences are a pattern." To the design patterns authors, there is a more literal meaning, that patterns are proven solutions applied by one or more communities of experts on a recurring basis.

Design patterns also introduce the notation of *design force,* also called *issues* or *concerns*. Design patterns document these forces explicitly and elaborate the solution in terms of resolving the design forces.

In order to facilitate problem solving, it is useful to find ways to separate design concerns—design elements which are implicitly responsible for resolving all potential concerns, those that are potentially unstable (when subject to scrutiny), and those that may require voluminous documentation to justify the design. Explicit reference models for separation of concerns have been proposed for software engineering and other fields of engineering endeavor.

Figure 1.8 also contains a software design-level model proposed by Shaw and Garland showing three levels [Shaw 96]. In comparison, the software community does not have a sophisticated view of how to separate design concerns,



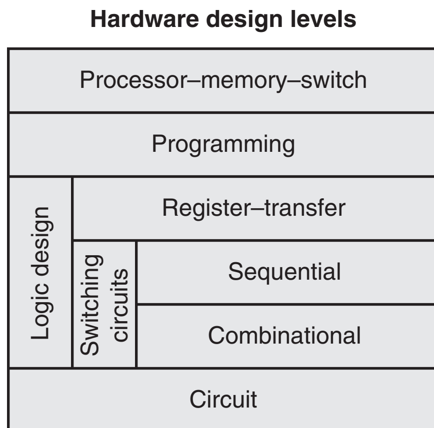


FIGURE 1.8 The Concept of Design-Level Models

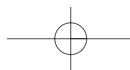
and it is also not known what the components are that comprise each of these levels. In the software design model, the *machine* level represents the binary software that is part of the operating system and commercial products that cannot be modified by the application developer. The *code* represents the program that is the domain of application development, and the third level is the *architecture,* which provides a model of how the system is partitioned and how the connections between the partitions communicate. The shortcomings of this simple model are that it does not represent any significant separation of concerns and that important properties such as interoperability between systems are not considered.

Software Design-Level Model

Figure 1.9 shows the software design-level model that we propose in our book called *CORBA Design Patterns* [Mowbray 97a]. This model was originated by one of the founders of the design pattern movement, Richard Helms, and describes in a recursive fractal fashion what the various levels of software design are in terms of objects. At the micro levels we have individual objects, and the design principles that apply to those individual objects are usually object specific. There is a class of patterns called idioms which represent design guidance for language-specific issues. These issues are fairly fine grained.

The next level up is called *micro architecture* patterns. In micro architectures we have small configurations of objects, generally a handful of objects that give us sophisticated ways of organizing our software structure to support variability in other qualities of design. The framework level then takes a number of micro architecture patterns and combines them into a partially completed

ch01.qxd 9/20/00 12:46 PM Page 29



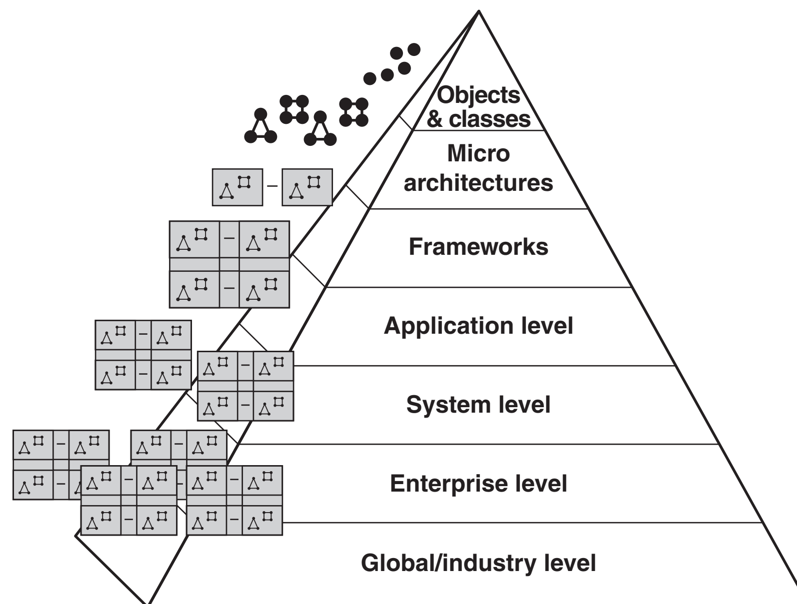
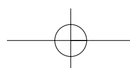


FIGURE 1.9 Software Design-Level Model

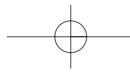
application with reusable software. Above the micro level, we have completed applications and systems. The *application* level represents the application of zero or more frameworks to provide an independent program. We encounter issues such as user interface programming which are significant in software development. At the *system* level, we take a number of applications which play the role of subsystems and integrate those applications to create a working system environment. The system level is where many of the design forces applicable to programming are changed in terms of their priorities. Management of complexity and change become critical at the system level and above.

At the *enterprise* level, we have a number of different systems which are integrated across an organization or virtual enterprise of organizations working in conjunction. The enterprise level is the longest scale of internally controlled operating environments.

The global industry level is represented by the Internet, the commercial market, and the standards organizations, which comprise the largest scale of software systems. Figure 1.10 represents the separation of design forces which occurs as we move throughout these various levels. Overall, the management of risk is a force which applies at all levels when we make software decisions. At the finer-grained levels, management of performance and functionality issues is very important and perhaps dominates any of the other design forces



ch01.qxd 9/20/00 12:46 PM Page 30



▶ **The universal force**
- Management of risk

▶ **Primal forces**
- Management of performance
- Management of functionality
- Management of complexity
- Management of change
- Management of IT resources
- Management of technology transfer

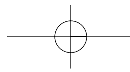
FIGURE 1.10 Prevalent Forces in Software Decisions

that apply horizontally across all the levels. Looking at the system level, the key design forces here include the management of change and the management of complexity. We come to this conclusion due to the writings of other authors. In particular Horowitz writes that the adaptability of systems is the primary quality which is missing where the majority of system cost is due to changes in requirements reference [Horowitz 93]. Shaw and Garland identify the management of complexity as the key design force at the system architecture level [Shaw 96].

Above the system level the environment changes on a more frequent basis. Each system must be modified to support individual business processes; at an enterprise level with multiple systems the change accumulates as people move and the organization evolves on a daily basis. Management of the resources at the enterprise level and of technology transfer to support capabilities such as design and software reuse becomes more significant and important. At the *global* and *industry* level, the management of technology transfer become predominant. When something is published on the Internet, it is instantly accessible on a global basis to virtually any organization or individual. Using the management of technology transfer design force, it is important to manage the information that the enterprise discloses in terms of software intellectual capital as well as the information that the organization exploits.

Figure 1.11 shows the overall priorities for these horizontal design forces as they apply to the coarser-grained levels. Here we show that at the system architecture level the management of change is the predominant force, because it is linked directly to the cost of the system in published work. We also identify as a second priority the management of complexity, because it is a design force that is emphasized by academic authorities in software architecture. Priorities at the other levels are indicated to show how the perspective of each of the architectural designers at these levels varies by the scale of software design. We see these as guidelines for making sure that the appropriate priorities are

ch01.qxd 9/20/00 12:46 PM Page 31



Horizontal Force	Component Programming	Component Integrator	Software Architect	Enterprise CIO	Global CEO
Performance	1				
Functionality	2	1			
Complexity		2	2		
Change			1	2	
IT Resources				1	2
Tech. Transfer					1

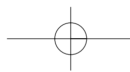
FIGURE 1.11 Priorities for Key Design Forces

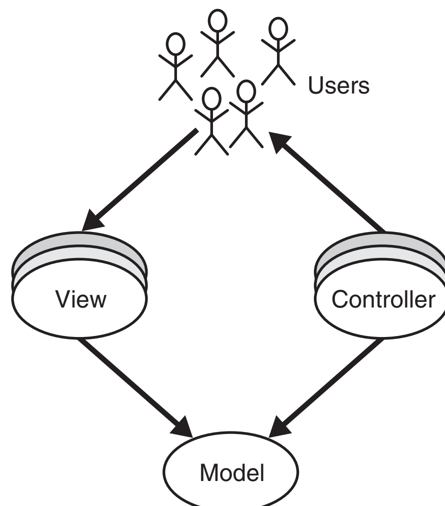
allocated to decisions that are made at each of these levels. The reference model helps us to organize patterns knowledge and identify priorities for design forces that are horizontal across all the levels. Design patterns are a modern approach to providing technical guidance. The breakthrough that design patterns provide is the capability of applying lessons learned and reusing design information across organizations.

Design patterns represent a high-quality academic research movement that has its own conference series and visibility at most other technology events. The origin of design patterns comes from actual bricks-and-mortar building architecture. The original vision for design patterns included a design level model which we did not discover in other authors' work. We believe that design patterns represent the right approach for documenting guidance and solving technical problems in software architecture and system development. Figure 1.12 shows an example of a popular design pattern called the model view controller. This is a pattern that applies at the framework level and provides an approach for reusing software objects that contain data and processing which must be viewed and controlled in many different ways.

The model view controller pattern includes model objects, view objects, and controller objects. The model object is the reusable component. It represents the data in the system and the encapsulating processes which need to be represented and controlled in several ways. The view objects represent various visualizations of that information, and there can be many simultaneous views that may be presented to groups of users. The controller objects represent various business processes or mechanisms for controlling the processing of the data. The model view controller pattern has been around at least since the invention of Smalltalk and has been reapplied at several different scales of

ch01.qxd 9/20/00 12:46 PM Page 32





Synonymous uses:

Smalltalk–80:
model–view–controller

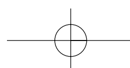
Jacobsen's reuse classes:
entity, interface, control

OMG business objects:
entity, interface, process

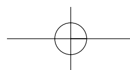
FIGURE 1.12 Model-View Control Pattern

software by various groups, including UML's business classes and the OMG business object task force which defines business objects in an analogous set of categories [Mowbray 97b]. Figure 11.6 shows the overall structure of design patterns. The essence of any design pattern is a problem–solution pair. The problem is explained and expanded in terms of the applicable design forces and any contextual forces which may be present. The solution resolves the design forces in a particular manner. The selection of any solution is a commitment, and a commitment provides some benefits as well as consequences. In addition, selection of a solution may lead to additional problems where other patterns are appropriate.

Design patterns are distinguished from other forms of software literature in that design patterns are presented in terms of a standard outline or template. Several templates have been published that meet the needs of various software design models. Figure 1.13 is a listing of the template developed for the CORBA design patterns [Mowbray 97a]. In this template there is a separation between the solution description and the variations of the solution, which may vary by structure and by scale of application. Making this separation allowed the authors to clarify the base solution at a particular scale and then to describe the variations and nuances of applying the pattern in separate sections of the template. The design pattern template is a rhetorical structure that assures consistent coverage of the key questions that people may need to answer in order to apply the design information. In particular, when justifying the application



ch01.qxd 9/20/00 12:46 PM Page 33



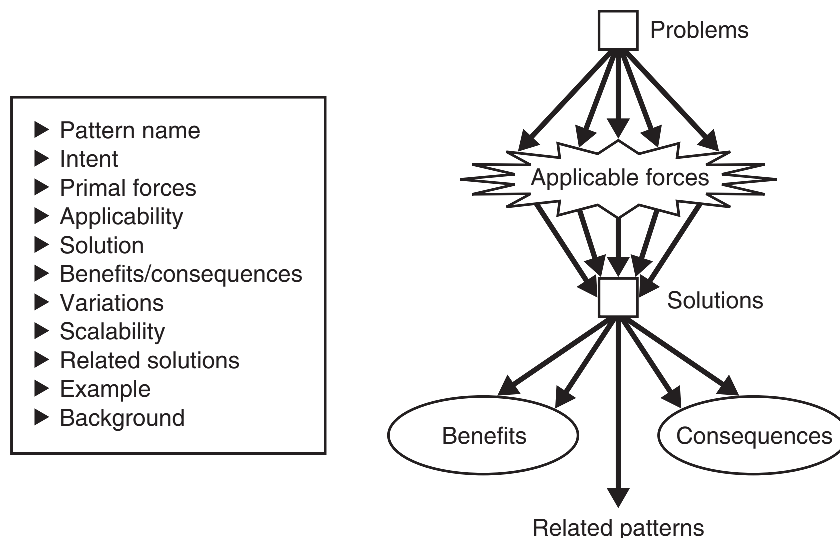
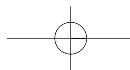


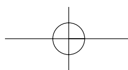
FIGURE 1.13 An Example Pattern Template

of a pattern, it is important to understand the benefits and potential consequences of the pattern to understand the real tradeoffs in design. If the design pattern authors have properly documented the pattern, they have identified those points of debate explicitly so that the users of the pattern do not have to reinvent that information.

Figure 1.14 is an example of a CORBA design pattern that applies in general to technologies beyond CORBA for system building. The problem is that most systems have vertical solutions, where each software module has a unique interface corresponding to each software implementation. The vertical solutions lead inevitably to stovepipe interdependencies between the modules in the system. By adding the common interface pattern to a system, we can capture the common interoperability information so that the software modules can interoperate without explicit dependencies upon particular implementations. The common interface pattern is a fundamental principle that is applied in standardization work and in software architectures in general.

Figure 1.15 shows a related pattern which applies the common interface in a more general and sophisticated context. In this pattern, called the horizontal vertical metadata pattern, we have a static architecture for a system defined in terms of a common interface with vertical interface extensions; also we are adding some dynamic architecture elements represented metadata. A key tradeoff described in the pattern talks about how dynamic architecture and static architecture can be varied to represent different portions of the design. Dynamic





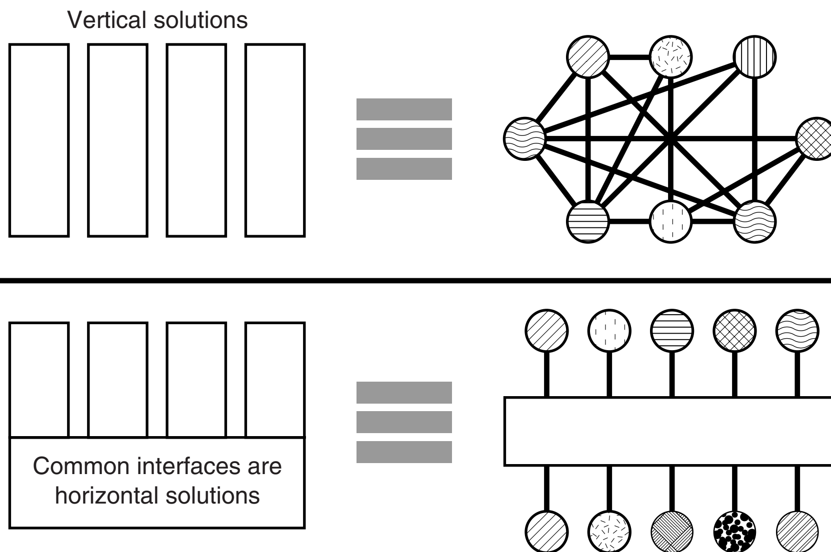


FIGURE 1.14 Common Interface Pattern

architecture is one of the key solutions for implementing variability and adaptability in software architectures.

Figure 1.16 shows how the horizontal-vertical-metadata pattern is actually an instance of a more general concept that is applied across standards organizations and profiling entities all the way down to a system level of

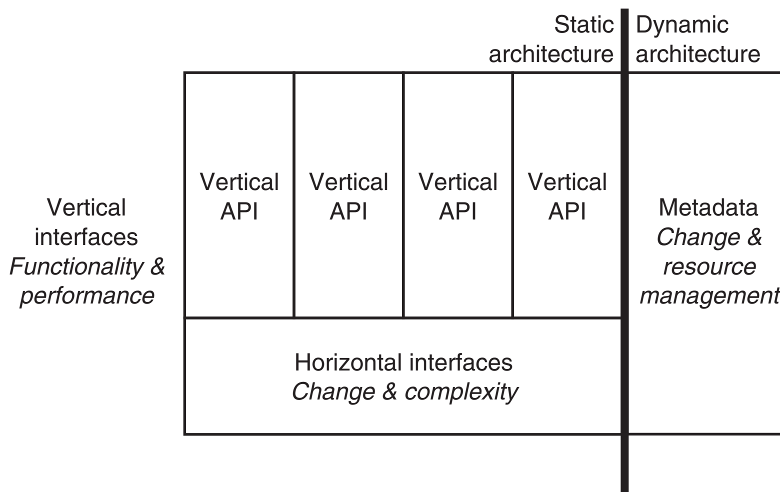
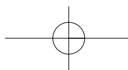
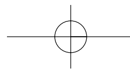


FIGURE 1.15 Horizontal-Vertical-Metadata Pattern



ch01.qxd 9/20/00 12:46 PM Page 35



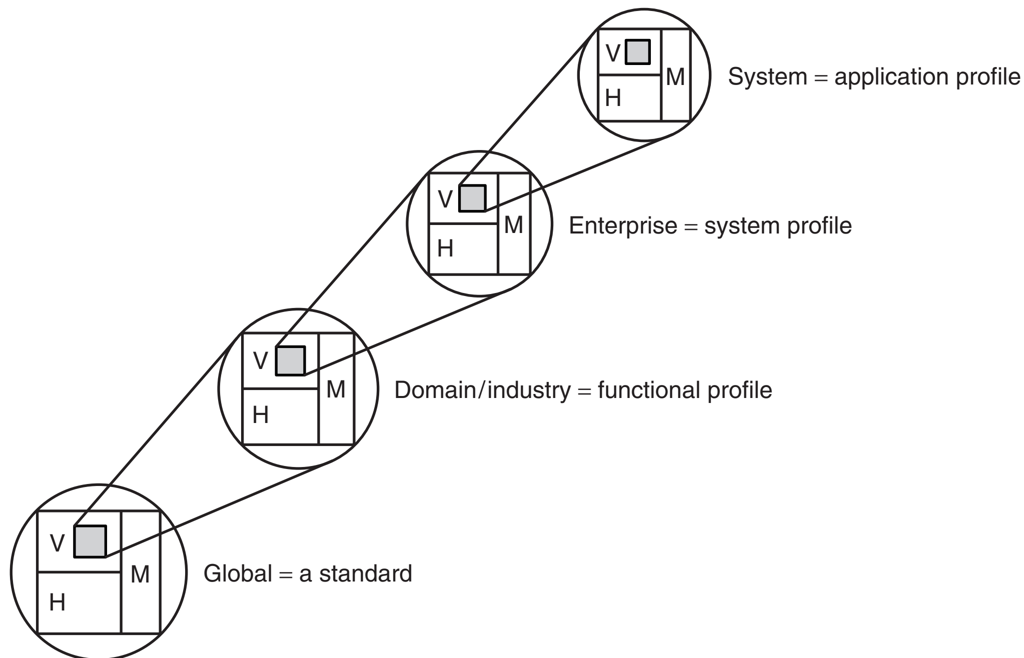
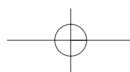


FIGURE 1.16 Pattern Applicability at Multiple Scales

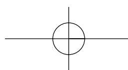
deployment. This application of the horizontal-vertical-metadata pattern is directly analogous to the functional and system profiles that we described in Chapter 4, where the functional profile is a vertical extension of a global standard. A system profile is a vertical extension of a functional profile, and any particular application system is a vertical instance of a system profile.

Figure 1.17 shows an application-level pattern and how it is applied. We present this example to give you a flavor of what is involved. In this case we are showing a UML sequence diagram. Before the pattern is applied, there is a simple request and return transaction which actually causes the client program to block while it is occurring. It turns out that this is the default behavior of most distributed computing infrastructures such as remote procedure calls and CORBA. We can improve the performance of this configuration by adding a moderate amount of complexity and, after applying the pattern, we can return a reference to the result which will be computed in parallel and then retrieved later (Figure 1.17).

Figure 1.18 shows a table of several examples of design pattern languages. Much of the available pattern documentation addresses a specific software design level. More recent work on CORBA design patterns and



ch01.qxd 9/20/00 12:46 PM Page 36



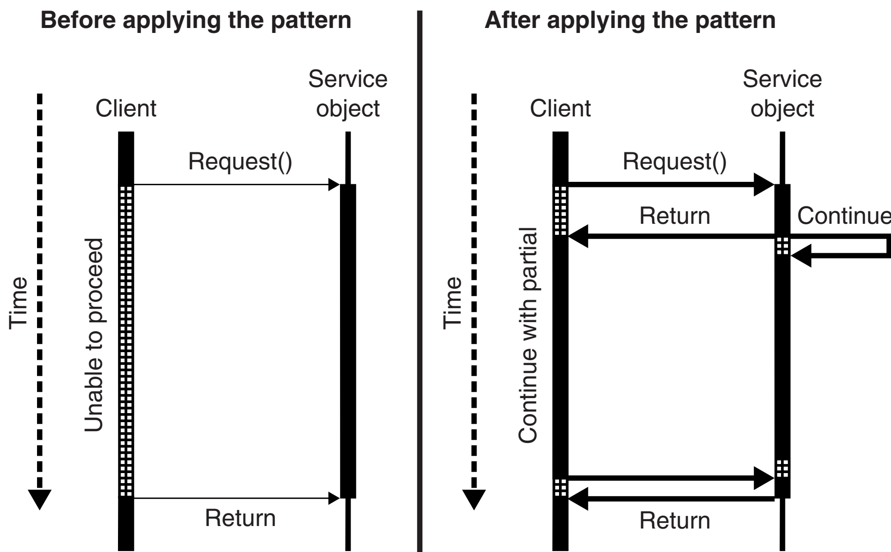
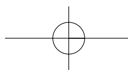


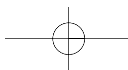
FIGURE 1.17 Partial Processing Sequence Diagram

pattern-oriented software architectures has addressed several levels of abstraction where these level are explicit. At the idiom level of design patterns we are concerned with individual objects. Idiom documentation has been widely available in the form of programming language guidebooks. Idioms represent expert programming techniques. These are techniques that one would rediscover after substantial use of a language. If software engineers are maintaining software written by other people, it is essential to understand idioms in order to understand the intentions of the programmers applying these sophisticated ideas.

	Gamma Design Patterns	**Buschmann Architecture Patterns**	**CORBA Design Patterns**	**Fowler Analysis Patterns**
Software Scale	Micro-Architecture	Micro to System	System	Objects to Micro
Most Useful to	OO Programmer	System Architect	System Architect	OO Analyst
Key Horizontal Forces	Change	Change Complexity Performance	Change Complexity	Functionality Change

FIGURE 1.18 Comparison of Design Pattern Languages





ch01.qxd 9/20/00 12:46 PM Page 37

One of the first published design pattern languages described microarchitecture patterns [Reference Gamma 96]. The goal of the gamma pattern language was to invent a new discipline of variation-centered software design. The gamma pattern language is organized in terms of several categories including creational patterns, structural patterns, and behavioral patterns. When applying the gamma patterns, complexity of design is increased with the benefit of potential support for potential modification of the software. Gamma patterns have become very popular and are applied widely in software engineering organizations today.

AntiPatterns

A recent development in the patterns community is called AntiPatterns. An AntiPattern differs from an ordinary pattern in that it is a solution pair rather than a problem–solution pair (Figure 1.19). An AntiPattern starts with a problematic solution. The reason why the solution is there is due to various contextual forces. The AntiPattern solution leads to various kinds of symptoms and consequences, and the consequences can be quite devastating. The AntiPattern proceeds to

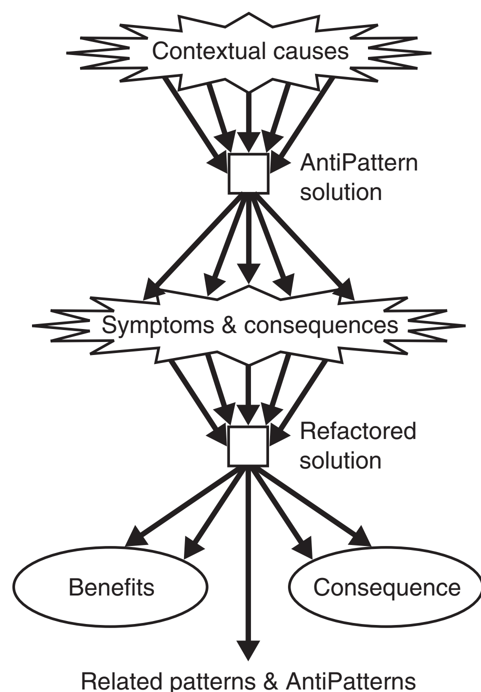
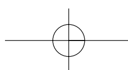
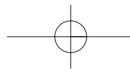


FIGURE 1.19 AntiPatterns





define a potential solution for migrating the problematic solution to a re-factored solution providing improved benefits. AntiPatterns are fundamentally about software refactoring. Refactoring is modification to software to improve its structure or quality. Common examples of AntiPatterns include stovepipe systems, spaghetti code, and analysis paralysis. AntiPatterns are further explained in the book *AntiPatterns* published by John Wiley & Sons in 1998 [Brown 98].

1.4 CONCLUSIONS

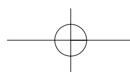
In order to realize the benefits of software components and object technology, much more effective guidance is needed than the naive application of objects which characterized the first generation of these technologies. Design patterns are a highly effective and academically based guidance approach that is now being practically applied in many software development shops. The technology and skills transfer available through design patterns can lead to some important benefits, including reducing software risks, enhancing the effectiveness and productivity of the software developer, and making successful practices repeatable.

In particular, the reference model for open distributed processing is the formal standard for object-oriented architecture. This reference model is widely used because it is effective for defining distributed systems. The model is used in many industries where mission-critical systems must be successful. RM-ODP separates complex concerns for the specification of distributed system. RM-ODP enables future proofing because it defines an approach for specifying architectural plans which are independent of distribution and technology choices. We believe that RM-ODP is a key architectural guideline for object-oriented systems and should be applied in your organizational practices.

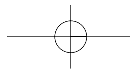
1.5 EXERCISES

EXERCISE 1.1 Define your career plan for the next two years. As your career progresses to higher levels of seniority, you will be expected to require redirection on a less frequent basis, with the maximum being about once a year. We believe that planning is essential, so making a career plan at this early stage of your reading would be a positive step. Identify your goals, and then identify what you need to know in order to achieve your goals (i.e., knowledge gaps). Be brutally honest.

EXAMPLE SOLUTION: Three years ago, my goal was to continue in technical architecture roles and increase my knowledge in several areas, so that I



ch01.qxd 9/20/00 12:46 PM Page 39



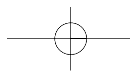
could be a more complete contributor. In particular, I wanted to gain extensive experience in UML modeling, design patterns, and software process and to reconnect with programming fundamentals. I also wanted to gain some management experiences for addition to my resume. I wanted to give the research and development cycle one more go, for both personal and professional reasons. After all, I joined this industry because I loved programming. At the back of my mind was a desire to help some friends in small commercial businesses, but I sorely lacked experience in this area, having worked mostly for large defense contractors and think tanks. Having a list of what I wanted to learn, I next consulted the Internet, the world's most extensive collection of free resources. I located several books, training courses, and other information that helped me identify specific learning targets.

Being a relatively independent middle manager in the technical ladder, I adjusted my workload to align with my goals. I prepared a tutorial on UML and defined an architecture using UML notation, that was within the scope of our research. I downloaded the latest version of the Java programming language from SunSoft and began programming the first phase of the architecture prototype. I was having fun and achieving my goals while performing useful architecture research and evangelism for my firm, which was in the midst of UML adoption. Reviewing my results with co-workers enriched my learning experience and helped my firm to move forward on UML-related initiatives. Also I pursued directed readings and attended a patterns workshop, which greatly enriched my knowledge of the field.

Having achieved a modicum of success on this path, I was ready for the next phase. Time to replan. In the pre-Y2K days, the software industry was very profitable. Opportunities abounded. I lacked much of the essential knowledge to help my friends in small commercial businesses (my ultimate goal). In addition, I wanted to do more technical architecture work, on a faster cycle. Defining a new architecture every month would be ideal, but that kind of opportunity was not available at my current firm. Also, business in my firm was in a cooling-off period.

A career change was in order. I took a job at a very stable, highly reputable small commercial firm; e.g., their paychecks came regularly and they always cleared the bank. This new firm knew everything that I wanted to learn—a perfect match. At the library I discovered the book resources to learn the requisite areas of knowledge that I was lacking, a bit of business training, and so forth. I was able to read about these matters and apply them on the job daily. I was able to complete several interesting architecture projects, including a financial system specification, a middleware architecture specification for a large telecommunications firm, and a high-level architecture for a real-time system. In addition, I was able to do a great deal of UML modeling, learn

ch01.qxd 9/20/00 12:46 PM Page 40



Visual Basic and C++, and do some CORBA programming. I was also teaching courses on the topics that I wanted to master—excellent progress, by any standards of performance. At this point, I had achieved the technical goals that I had set two years earlier. Time to re-plan, as this exercise continues in real life.

EXERCISE 1.2 Select an architecture framework for use in your current firm (or customer's organization)—for example, RM-ODP, Zachman Framework, or 4+1 Model View. Write a brief profile description about how the framework should be applied in your organization.

BACKGROUND FOR SOLUTION: We believe that having a framework is far superior to working without one. Whatever framework you choose, certain conventions and guidelines for applying it in your organization will need to be managed. The need for these profile conventions is most obvious in the selection of the Zachman Framework. Since you have 30 candidate specifications to write, you must address two issues. First, 30 specifications is too much work, and you should compress and simplify the amount of effort required to plan a system. Focus on the useful, practical elements for your domain of application. Combine elements as appropriate to assure coverage without elevating the document-driven aspects to an unreasonable level. Second, if there is no profile, you can't possibly expect any two architectures to be comparable. You should select essential and optional viewpoints to be specified, and define what they mean in your organization's terminology. You can also propose conventions for how these viewpoints will be documented, such as a template for each viewpoint, and notational conventions. We believe that these steps are required for any responsible application of these powerful frameworks.

EXERCISE 1.3 Create a pattern system for use in your organization. Select patterns from among the available pattern catalogs to cover the areas of greatest concern and need in your organization.

BACKGROUND FOR SOLUTION: A "pattern system" is documented in a simple tabular form. Use page 380 of [Buschman 96] as your starting point. The pattern-system table contains a listing of the names of each pattern, along with their book page reference, for quick retrieval. Implicit in this exercise is the selection of the key patterns catalogs (i.e., books) that would be readily available to every developer. Remember: Patterns are lessons learned. The purpose of this exercise is to create a job aid so that your developer can more effectively apply lessons learned. We suggest that you consider including sources such as [Gamma 95], [Fowler 96], [Mowbray 97], and [Hays 97] to your list of candidate catalogs.