Chapter 4: Introduction to Software Design

Overview

Higher-quality design, not just higher-quality development processes, is necessary in order to achieve a high-quality product. But what characterizes a higher-quality design, and how do we achieve it? And, more fundamentally, what characterizes an activity as design, and what is the product of design? This chapter presents the fundamental methods of the software design process and establishes the scope of the architectural level of design. The same fundamental design principles and methods apply whether a software architect is analyzing the problem domain, specifying the function of an application to address the problem domain, or designing the structure or form of the software. They also apply when a software engineer is creating a detailed design (including coding) or testing an implementation or designing a development environment in which to create the code. A generalized method of design is presented. The main topics in this chapter are:

- **Problems in software architectural design.** This section addresses why design is so difficult and sets the context for the remainder of the chapter.
- **Function, form, and fabrication (the Vitruvian triad).** This section introduces the three aspects of software architecture, which can also be described as purpose, structure, and quality. These aspects of architecture help form a conceptual framework for architectural design.
- **The scope of design.** This section defines what activities are considered to be design activities.
- **The psychology and philosophy of design.** This section presents some concepts around the subject of how people design things and what characterizes the mental activity of designing.
- **General methodology of design.** The last section presents fundamental design methods that apply to software architectural design and other aspects of software design.

Problems in Software Architectural Design

Experience shows us that as the size and complexity of applications and their development teams grow, so does the need for more control of the design of the application. An ad hoc approach to application design and implementation does not scale with respect to the size of the application in terms of functions and other quality attributes. This is often manifested in what is called the *software crisis*. We need better control of the design process in order to improve the quality of the product and make more accurate predictions of the amount of effort required to develop a system.

The obstacles to achieving high-quality architectural design in software development are:

- A lack of awareness of the *importance of architectural design* to software development
- A lack of understanding of the *role of the software architect*

- A widespread view that designing is an *art form*, not a *technical activity*
- A lack of understanding of the *design process*
- A lack of *design experience* in a development organization
- Insufficient *software architecture design methods and tools*
- A lack of understanding of how to *evaluate designs*
- *Poor communication* among stakeholders

Most of these obstacles stem from ignorance around what software architecture and design are and why they are critical to the success of an application or other system development project. This ignorance results in an inability to effectively communicate what problems an application is really addressing and how a technical solution solves those problems.

The solution to the problem of inadequate design must address the obstacles to achieving adequate design. The solution involves the following:

- Evangelizing the importance of software architecture
- Improving software architecture education
- Using architecture methods and tools

There are many authors evangelizing software architecture today; for example, Sewell and Sewell make a case for software architecture as a separate profession from software engineering (Sewell, 2002). There is a strong need for more education in the field of software architecture. Some authors believe that it is a discipline of its own, separate from (though related to) software engineering and requiring its own colleges. At the core of the solution is the need for architecture methods and tools. By tools I mean the methods, techniques, principles, heuristics, and catalogues of reusable design elements (for example, design patterns).

The solution to the communication obstacle is not just to create more communication channels, but to improve the quality of the information being communicated. Weekly design review meetings may improve the transfer of information, but they won't improve the transfer of knowledge. Information may (and will) be interpreted differently, and so we need to make what we communicate more meaningful and rely less on individual interpretation. Standardized (or at least mutually agreed-upon) architectural terminology, design processes, and description languages are extremely important to effective communication among a project's stakeholders.

At the core of design is the notion of problems, obstacles, and solutions (interestingly, this corresponds with the design of this and other chapters). A solution to the problem of software design, in a nutshell, is the careful planning and systematic execution of a design process and the production of design artifacts (models and specifications). Fundamental to understanding design is the understanding of what design is and what differentiates the act of designing from other activities. In the next section, we look at design in a broad architectural context.

Function, Form, and Fabrication: The Vitruvian Triad

Vitruvius wrote in his *Ten Books on Architecture* that an artifact (the result of architecting) should exhibit the principles of *firmitas*, *utilitas*, and *venustas*. These three principles are known as the *Vitruvian triad*. The principle of firmitas, or *soundness*, "will be observed if, whatever the building materials may be, they have been chosen with care but not with excessive frugality." The principle of utilitas, or *utility*, "will be observed if the design allows faultless, unimpeded use through the disposition of the spaces and the allocation of each type of space is properly oriented, appropriate, and comfortable." The principle of venustas, or *attractiveness*, "will be upheld when the appearance of the work is pleasing and elegant, and the proportions of its elements have properly developed principles of symmetry." A translation of Vitruvius can be found in Rowland (Rowland, 2001).

The Vitruvian triad is a useful anecdotal device for thinking about software architectures and the process of architecting. Utilitas includes the analysis of the purpose and need of the application (function). Venustas includes the application of design methods to balance many competing forces to produce a useful system that serves some application (form). Firmitas includes the principles of engineering and construction (fabrication). A sound artifact requires sound engineering. Thus we can think of the Vitruvian triad as the principles of *function*, *form*, and *fabrication*.

An application or software system is constructed based on the specification of its form, which satisfies a function. The form includes the specification of distinct quality attributes that, through a realization of components in a system, satisfies the function. Architectural design is what brings function and form together. In software architecture, the terms function and form may be confused with functionality (for example, functional requirements or features) and internal code structure (for example, architecture). Function in the architectural sense really means the *need, purpose, utility,* or *intended use* of the system or application. One may ask "What is the function of this application?" or "What is it used for?" or "What purpose does it serve?" In other words, what *problems* does the application or system *solve*? Architecting begins with a specification of an application or system in terms of the functions, capabilities, and other qualities that it must possess.

Function and Product Planning

Part of the product-planning phase, as we will see later in this chapter, involves the analysis of the problem domain. The product of this analysis is a formulation of the requirements that an application or system must satisfy. The requirements establish the function of the application or system, not to be confused with the functional specification of the system (that is, part of the form).

For the architectural metaphor to make sense, an application's architecture must address end-user needs. The architecture of a building, for example, is (partially) perceivable to the inhabitants of the building (it's something they see in the exterior and interior of the building and something they experience when they go about their daily activities inside and in the proximity of the building). The architecture of an application likewise must be (partially) perceivable to the users (inhabitants) of the application (which is different from the user being exposed to the internals of the application). This cyber-reality or *virtuality* of the application architecture goes beyond human computer interaction or human factors or even user interface design. The functional architecture may not be obvious from the structural architecture and vice versa, but sometimes one does follow the other.

Form and Interaction Design

Alan Cooper draws a distinction between design that directly affects the ultimate end user (*interaction design*) of the application and all other design (*program design*). Poor interaction design contributes to what Cooper calls *cognitive friction*. Cognitive friction is "the resistance encountered by a human intellect when it engages with a complex system of rules that change as the problem permutes" (Cooper, 1999).

The interaction with software applications is high in cognitive friction because of the large number

of states and modes that an application can have. Mechanical systems are low in cognitive friction because they tend to have relatively few states and relatively few modes. In software applications, the mode the application is in affects the behavior of a given function or operation. Selecting a menu item or clicking on a button may have different effects, depending on the mode of the application. Cognitive friction is not necessarily bad, just as having a large number of states and modes in an application is not inherently bad. What we judge as good or bad is how we manage this complexity in the interaction design of an application.

Terry Winograd (Winograd, 1996) refers to interaction design as simply *software design*. "Software design is the act of determining the user's experience with a piece of software. It has nothing to do with how the code works inside, or how big or small the code is." The product of interaction design is the *user's conceptual model* or *virtuality*. The user's conceptual model is based on his or her experience with the software. Other terms that convey the same idea are *cognitive model*, *interface metaphor*, and *ontology*. Cognitive friction occurs when a virtuality is poorly designed, that is, designed without the user in mind. Cooper says, "Almost all interaction design refers to the selection of behavior, function, and information and their presentation to users." This is an aspect of the field of research called human/computer interaction (HCI). HCI is composed of four threads of research (Carroll, 2002):

- Software engineering methodologies of prototyping and iterative development
- Software psychology and human factors of computing systems
- Computer graphics and software user interfaces
- Models, theories, and frameworks of cognitive science

I extend the definition of end users with respect to virtuality to include programmers. If you are architecting an enterprise application platform or framework that doesn't have an application interface (other than required administration and installation interfaces), you still need to consider the virtuality of the system in terms of its application program interface (API). An API is a type of language; it has words (the functions, methods, classes, and other data structures) and grammars (the constraints on how objects are created and combined, the order in which methods are called). A well-crafted API is easy to learn and apply, unless, of course, the implementation is defective. For example, many Java 2 Enterprise Edition (J2EE) APIs are well designed; they solve very specific problems, the class designs are logical and intuitive, and their constraints are fairly simple. Different APIs can be combined or *synthesized* to solve larger problems. An API that is a kitchen sink of functionality that does not have a well-designed conceptual model is difficult to learn.

A related concept is the *application domain*. The application domain is the part of the world in which an application's effects will be felt, evaluated, and approved by users (Jackson, 1995). This is a different connotation than the idea of a generic domain or class of applications. The term *environment* is sometimes used to mean the same thing. The correct identification of the application domain is necessary in order to focus on requirements. The virtuality of a software application must be congruent with the application domain. The problem being solved is shaped by the definition of the application domain. Use cases are an example of an application domain modeling technique. The Reference Model for Open Distributed Processing (RM-ODP) enterprise viewpoint language also addresses the definition of the application domain (the enterprise in which various systems interact to achieve some objective).

You can see already that there are at least two views or aspects of an application's architecture: the *function* and the *form*. In this book we are concerned with both aspects of an application's architecture.

Cognitive Friction and Architectural Design

Let's consider an example in a different design domain. Donald Norman in *The Design of Everyday Things* (Norman, 2002) gives an example of a consumer refrigerator's control panel for adjusting the temperature of the freezer and the fresh food compartment. There are two controls, one marked with letters A through E for adjusting the freezer and another marked with numbers 1 through 9 for adjusting the fresh food compartment. But the instructions for modifying the temperature of one compartment or the other require adjusting both the freezer and fresh food controls. The initial settings are C and 5. To make the fresh food colder, the numeric control must be moved up to a higher number and the lettered control must be moved to a prior letter, for example, the settings B and 8. To make only the freezer colder requires a setting such as D and 8. Getting the temperature right in both compartments requires experimenting with the two controls.

The controls are nonintuitive, and the instructions provide no hint as to how to effectively change the control settings, requiring the user to guess and wait for the temperatures to change and settle to see if they reach the intended values (usually a 24-hour period). The problem is that the controls were directly affecting internal structures: the numeric control adjusting the cooling unit and the lettered control adjusting the proportion of cold air being directed toward the refrigeration compartment or the freezer compartment.

What this illustrates is that the user interface didn't map naturally to a user's conceptual (mental) model of how the refrigerator should be adjusted, such as one control for adjusting the temperature of each compartment individually. In other words, the virtuality of the refrigerator as embodied in the controls did not abstract away the internal structure (not to mention that the written instructions made its use even more confusing). A good user interface can hide local complexities in an application, but if the overall functional metaphor is unnecessarily complicated, then the application itself will appear incongruent and hard to use and will frustrate users.

Another example is found in Bass and Coutaz's *Developing Software for the User Interface* (Bass, 1991). They describe an interactive system for operating a mobile robot. The purpose of the system was to allow an operator to specify a mission for a mobile robot within a physical environment. The mission specification would direct the robot through an environment that included obstacles. There were two approaches they could have taken with respect to the interaction design of the system. The first approach would be to require the user to specify the mission in a way that is comprehensible to the physical robot, since this low-level specification was necessary to motivate the robot. The preferred approach, however, was to hide this complex implementation from the operator and instead allow the authors to define a mission in terms of higher-level descriptions that could be transformed into a low-level mission specification.

The behavioral and information models of the system commonly represent the virtuality of a system. The requirements of a system can map to many behavioral and information models; therefore, creating a good virtuality requires attention to these fundamental models. The internal structure of the refrigerator was based on engineering design. It was an optimal solution for the cooling of two individual compartments using a single refrigeration coil. The user would not be aware of this internal optimization and would rather work with more natural concepts. Likewise, the mobile robot mission specification was also based on engineering design. It was an optimal solution for motivating a physical robot through a physical environment. The user would not be exposed to this structural detail but rather to a more qualitative environment for specifying the robot's mission in terms of obstacles to avoid and routes to take, allowing the computer to generate a physical quantitative path.

Fabrication

Vitruvius's *firmitas* is the principle of quality (soundness, durability). The product of architecture and

construction must be a high-quality system. The quality of a system is based not only on the design of the system but also on the selection of technologies used in the implementation of the system. I chose the term *fabrication* to connote both the product of construction as well as the construction itself (but certainly not to mean a *falsehood*, which is one connotation of the term).

An architecture must be realizable; it must be possible to build the system as described by the architectural description. That may sound obvious, but it is often not even considered as an architectural attribute. It must be possible to apply current software engineering practices and technologies toward the implementation of the application or system that is described and to satisfy the specified quality attributes. It must be possible to build the system given the available resources such as time, staff, budget, existing (legacy) systems, and components. If existing technologies are insufficient for the problem at hand, will the team be able to invent what is necessary?

For example, several years ago there were no standard commodity application servers so many in-house development projects and commercial enterprise vendors had to build application servers into their products or original equipment manufacturer (OEM)-specific vendor application servers. The architect had to take this into account: Were the staff skills, project budget, and schedule sufficient to build the application server? Today, systems that require application servers tend to be easier to build because there is no need to invent an application server. Commercial application servers are ubiquitous and relatively inexpensive. A good architecture identifies those factors of the solution domain that can be satisfied using commercial off-the-shelf (COTS) components, taking into consideration the cost of licensing the technology versus the cost of building it.

Application Architecture

Interaction design that addresses concerns other than structural design is key to understanding the difference between application architecture (see Chapter 1) and more general software architecture (such as the architecture of a compiler). Although the design principles in software development are universally applicable to high-level and detail-level design as well as different software domains (compilers or enterprise applications), it is important to distinguish between the types of objectives that drive the design activities. Interaction design is usually not a quality of compiler architecture. A compiler does have a virtuality: Its problem domain is that of textual information conforming to formal grammars and having some target language to which it can be compiled.

Example

Traditionally, part of the architecture role was assumed by *systems analysts*, who analyzed the problem and specified a functional model of the system that would address the problems. For example, consider a system where a customer order entry is written by hand and later keyed into a system. This is a time-consuming and error-prone process. An organization may realize that this system cannot scale and therefore brings in a team of Information Technology (IT) specialists to analyze the problem and propose a solution. The analysts determine that the following problem statements capture the essence of the existing system:

- Writing an order by hand is slow (reduces the number of customers a given sales associate may service and may negatively affect sales figures).
- A lot of the form data may already exist in some fashion if the customer has placed an order before (for example, delivery address, payment information).
- Writing an order by hand is error prone (the wrong customer information may be written, such as account numbers or product codes).
- Keying information into a software application later in the process is another opportunity to

introduce errors in the data.

In addition to the requirements, the systems analysts may discover some additional features that were not envisioned by the acquirers but which support the overall objective of the system. Some examples of these suggested features are:

- When an existing customer profile is accessed, the sales associate may also have access to previous orders for that customer. This information may help the sales associate to make suggestions to the customer or even simplify the order creation process if the customer is reordering. This can potentially speed up the order entry process and possibly increase the amount of the individual sale.

- The system can keep track of the products previously ordered by the customer and provide a list of suggestions for other products. The sales associate can use this suggested product list to make suggestions to the customer, potentially increasing the amount of the individual sale.

These statements make up the problem and define the function (utilitas) of the application. After understanding the crux of the problem, the analysts or application designers begin identifying the functional requirements (features and capabilities) and other quality attributes that make up part of the architecture of the application. Suppose our team proposes the following requirements:

- A PC-based order entry system at each sales associate's desk.

- A graphical forms-based order entry system that is linked to a customer database and products database.

- Auto-completion of some form fields to make entry faster: for example, the user begins typing in customer's last name and a list of possible customers to choose from appears on the screen. The user may stop typing the name and then scroll or use a mouse cursor to select the customer; continuing to type will reduce the field of potential customers.

- Selecting the customer from the auto-completion view causes the form to be partially completed, including everything from the customer database that is relevant, for example, full name, billing address and information, shipping address and instructions.

- The auto-completion view will provide enough information on screen to allow the sales associate to determine that it is the correct customer before selecting it (for example, there may be a lot of Smith, Steve entries).

- If the user accidentally selects the wrong entry, it must be easy to undo the action and clear the form quickly (such as pressing a single key on the keyboard or clicking on a single button on the screen).

Some of these requirements have already begun to suggest the form of the internal structure of the solution: We can see that there is a customer database and product database that all workstations can access, and there is an order entry program or application accessible from the workstation. Other than these three functional components, there is not much that has been revealed about the internal structure of the system.

Each of the functional requirements above addresses the problems in some combination. For example, the auto-completion feature is intended to address the slow order entry problem. Typing into a form on a screen may not be significantly faster than handwriting fields on a paper form, especially if the user is not a proficient typist. However, automatically filling in several fields can significantly improve the speed with which a user can complete a form by reducing how much he or

she must type. The other requirements related to the auto-completion of the form support the main capability and are intended to ensure that the feature is indeed useful in speeding up form entry and reducing errors. The analysts didn't have to specify these features and could have instead relied on the designers and engineers to come up with features that support the objective of speeding up form entry. We can assume that our analysts have some knowledge of solution patterns such as these and therefore have included them in the specification. They could annotate these requirements as preferences, knowing that they will satisfy the objective, but allow the engineering team to present other solutions that also address the same objective.

In addition to the intended function of the system, the designers must consider business-related quality attributes. Some examples that may affect the architecture are:

- Cost of implementation
- Cost of deployment, administration, and technical support
- Cost of training
- Time to deliver

The return on investment (ROI) should be the overriding concern of the system designers because the *utilitas* of the system is not to make order entry faster and less error prone, rather it is to increase revenue. A system that increases productivity by a factor of 4 but which costs more to implement than the company will realize in several years can probably be considered a failure. The cost of implementation includes development staff salaries and development tools such as workstations and software. Deployment and implementation costs include not only staff salaries for installation and administration but also the cost of purchasing workstations for sales associates, the time and cost of training, and the time lost mastering the necessary skills to use the system. Will the new system actually be faster than handwriting orders or will it prove to be a hindrance to its users? Will it be usable?

PREVIOUS NEXT
PREVIOUS NEXT

The Scope of Design

At this point you should have an idea of what the scope of architecture is with respect to software development and with respect to software engineering. In this section we look at the scope of design in general with respect to application development.

Tasks and Activities of Design

Any design activity can be seen from many points of view:

- Psychological
- Systematic
- Organizational

From the *psychological* view, design is a creative process that requires knowledge in the appropriate disciplines such as software engineering, computer science, logic, cognitive science, linguistics, programming languages, and software design methodologies as well as application domain-specific

knowledge. HCI approaches software development and use from the psychological aspect.

From the *systematic* view, design is seen as an architecting or engineering activity that involves finding optimized solutions to a set of objectives or problems while balancing competing obstacles or forces.

The *organizational* view considers essential elements of the application or system life cycle. Application development begins with a market need or new product idea. Application design (or, more generally, product design) starts with planning and ends when the product reaches its end of life (in software this means executing an end-of-life strategy and implies that a product is no longer maintained or supported). There may be some recycling of software elements to be reused in other products.

Design tasks may be classified by a variety of characteristics. The following characteristics are based partly on Pahl and Beitz (Pahl, 1996):

- Origin of the task
- Organization
- Novelty
- Production
- Technology
- Horizontal domain
- Quality attributes

Origin of the Task

There are many possible origins for a design task such as:

- Product planning (especially for commercial software)
- In-house software (custom systems)
- Systems integration
- Production and field testing

The genesis of a software application may involve a group responsible for product planning, also known as product management. The product-planning group identifies a real or perceived need and establishes the requirements of an application or system. The need for the system may be internally generated to solve some in-house business problem, or it may be market driven with the intention of selling the software. Sometimes systems that are originally designed for internal use are commercialized and sold externally, after executives discover their potential resale value.

Some software vendors do not create and sell complete applications, but rather domain-specific application frameworks. This is common with enterprise software vendors. The vendor cannot anticipate every potential customer need so instead focuses on building a platform or framework that provides general application domain-level services along with a software development kit (SDK) for

building applications on the platform. The assumption is that it is ultimately more cost effective for the customer to license the platform and build on it than to build the system from scratch. A customer may make a special order for a system, which is handled by the vendor's own consulting organization (sometimes called *professional services*). A company may start the process by hiring an outside software architecture consultant to assess its needs and make recommendations and proposals. Sometimes an enterprise vendor sells a partial solution to the executives of a company without input from a software architecture team or the engineering organization (much to the dismay of the engineers).

In-house software may be built by a company's own IT or engineering organization or may be contracted out to an outside software development firm. This type of design usually starts with some internal organization identifying a business need for an application or system.

Some tasks start out as specific systems integration problems in which case the requirements are much more rigid, and there is more need to interact with other engineering or design groups. Systems integration may involve third party to third party integration, third party to internally developed system integration, and even integration between internally developed systems such as is common when integrating different departments of the same organization. This last form of integration is commonly the result of a merger or acquisition or even an integration effort between two different enterprises (Business to Business Integration [B2BI]).

In the context of engineering machines or devices that must be manufactured, there are design tasks related to the creation of production machines, jigs and fixtures, and inspection equipment. Likewise, in software development the production-level work requires development and testing environments, configuration and release management mechanisms, and bug-tracking systems. Unless you are using a completely integrated off-the-shelf system for developing software, you will most likely have to assemble a system from various vendors, open source projects, and internally developed tools. The creation of a development (environment) system is a design task much like the development of an application, but is often not treated in such a manner.

Organization

The design and development process is commonly organized around the structure of a given company or development organization. Some common organizations are product-oriented; some are problem-oriented.

In *product-oriented* companies, product development and production responsibilities are divided among different divisions based on product type. For example, it is common in enterprise software companies to have a platform division and one or more application divisions based on general application domains. Sometimes the application divisions are the only users of the platform, and sometimes the platform is sold as a product (such as a relational database or an application server).

Problem-oriented companies organize work according to a division of labor along domain boundaries such as database administration and user interface design. An organization based on design phases divides specific design phases among divisions. For example, product planning, architectural design, and implementation (detail design) phases may be assigned to individual organizations. I have seen examples in industry of each of these approaches as well as hybrids of them. Each organization type affects the choice of design activities and how they are performed as well as affecting the role architects and engineers assume.

Novelty

Some design tasks require much more invention and creative problem solving than others do. The novelty of the problem can be characterized as follows:

- Original design
- Adaptive design
- Variant design

Original design starts with a clean slate. The designers start with a set of problems and objectives to which no known solution principles exist. Original designs may be created through the synthesis of known solution principles and existing technology or it may require inventing new technology. For example, several years ago I worked on a project where one of the primary problems to solve was to create a generalized mechanism for transforming structured (hierarchical) data to other hierarchical form. At the time the XML 1.0 specification had just been published and technologies like Extensible Stylesheet Language Transformations (XSLT) were not available. The problem was novel at the time and required inventing a technology for tree transformations because such a technology did not exist.

Adaptive design starts with known or established solution principles and adapts the principles to fit the current problems or requirements. Adaptive design may involve some original design as new components are added to meet the new requirements. In a software development system or application, modifications made during the maintenance cycle may involve adaptive design unless the modifications are so localized and isolated in the source code as to not require any conceptual or embodiment design work.

Variant design starts with an existing design that must be changed only with respect to some nonfunctional quality attributes. For example, a system may need to be reworked to handle greater user loads or to perform certain operations faster. Some variant design may not be possible at the detailed design (implementation) level and may require new embodiment or even conceptual design, which is no longer variant design but may be adaptive or original design.

Production

Software may be created for either a one-off custom system or for resale (either as shrink-wrapped applications or enterprise platforms). One-off software system production is similar to one-off or small batch production in established engineering fields. The design activities for one-off system development should emphasize risk reduction. Development of throwaway prototypes is not always economically feasible. Functionality, maintainability, performance, and reliability and/or availability are commonly the paramount qualities.

Commercial enterprise software production is similar to large-batch or mass production where the product is used by customers solving similar problems but with different requirements. Functionality, performance, extensibility, adaptability, and usability are commonly the paramount qualities. Functionality in commercial enterprise software tends to be more general than in specific in-house systems.

Technology

Different technologies require different design methods. The working principles of user interfaces are different from business logic involving distributed transactions and require different design activities. Some technology areas that affect design methods are:

- Information representation
- Data storage

- Data transformation
- Business logic
- User interface design
- Vendor platforms

The above list is by no means exhaustive of the types of software technologies available today, but does demonstrate the varieties that exist. User interface design methods include those proposed by Bass and Coutaz (Bass, 1991). Information representation design methods include object-oriented analysis and entity relationship modeling. Some vendors have suggested design methods for building applications using their proprietary technology.

Horizontal Domain

Systems can be characterized by their relative complexity. A plant (typically highly complex) is composed of machines and instrumentation, which in turn are composed of machine assemblies and components (typically less complex). In software application development we have similar complexities that affect the design tasks. I call these *horizontal domains*.

Each horizontal domain is a layer that incorporates the components of the lower layer. In this model each layer is not necessarily more complex than the lower layer (except in the sense that the higher layers inherit the complexity of the lower layers). An enterprise application may be less complex to design than an operating system, but it is more complex in the sense that it is composed of the operating system and other underlying technologies. The following list is an example of horizontal domains. The domains can be further subdivided; for example, the enterprise applications can be divided into high-level components and objects. The demarcation of a horizontal domain is based on the types of concerns addressed by design and the types of patterns, methods, and tools specific to a domain:

- Integrated enterprises (B2B)
- Enterprise integration: systems of applications (integrated applications)
- Enterprise applications (point or vertical solutions)
- Software libraries, databases, application servers, operating systems (platforms)

Quality Attributes

One of the largest factors that affect the design task is the set of required quality attributes of the system or application. Most design methods focus on functionality. Functional models such as use cases typically drive object-oriented projects. Attributes such as modifiability are not easily expressed in purely object-oriented terms and need supplemental textual descriptions to represent the design. Each attribute must be considered during design and usually requires the architect to make multiple passes at a design. A class design may start by addressing functionality only. Then another pass is made to incorporate the modifiability requirements while making sure that the functionality is not affected (in terms of the functional requirements, not necessarily in the actual interaction design of the application, which may change). Likewise, many passes are taken at a design to incorporate the various requirements and to evaluate the design and design tradeoffs.

There are many competing quality attributes, and the architect must find a suitable design that strikes a balance among them. For example, modifiability and performance are commonly competing

requirements because modifiability design techniques usually incorporate extra levels of indirection (interfaces) and stricter encapsulation of data and services to make modifications more local and less pervasive.

Performance design techniques, however, usually incorporate fewer levels of indirection and optimizations to data structures in order to improve operation execution times. Cost is usually competing with everything else, and it is common that usability is sacrificed first.

Some common quality attribute characterizations are:

- Functionality
- Buildability
- Cost and time to market
- Performance
- Usability
- Security
- Availability and reliability
- Modifiability

Architecture versus Engineering Design

Many design tasks in software development may be considered either strictly architectural or strictly engineering. Sometimes the distinction is fuzzy. Production-based design issues (the design of a software development environment for a particular organization or project) is considered an engineering activity (but is commonly not even regarded as a design activity because it is not a product design-related activity).

One way to characterize the difference between software architecture design and software engineering design is in scope and complexity. While engineering design concentrates on implementing specific quality attributes using technologies, architectural design concentrates on formulating the quality attributes and system characteristics and selecting the working principles that balance many competing quality attributes. One may extend the definition of software engineering to include what I described as architectural design, but the distinction between the two types of design is important (even if the boundary is fuzzy) because it establishes two separate disciplines with different objectives and which use different tools and methods.

PREVIOUS NEXT
PREVIOUS NEXT

The Psychology and Philosophy of Design

Software design is rooted in the fundamentals of language, logic, and knowledge. In this section, we take a look at the psychology of design and the application of logical reasoning as the basic means of problem solving.

Design is a creative problem-solving activity that involves intuitive and discursive thought. The

intuitive thinking process is mostly subconscious and is characterized by flashes of insight and inspiration commonly triggered by some association of ideas. The *discursive thinking* process is conscious and deliberate where facts and relationships are analyzed and combined in a variety of ways, evaluated, and then disposed of. The products of discursive thinking are knowledge structures, also known as *semantic networks*.

Problems, Obstacles, and Solutions

All design methodologies consider design as an activity of finding or creating solutions to problems given a set of obstacles to overcome (see Figure 4.1). Among the types of obstacles that exist between a problem and its solution is a lack of understanding of the problem. Fundamental to good design is a grasp of the problem being solved; without this understanding it is hard to create a good design because it cannot be assessed based on its effectiveness in satisfying the problem. This is not to say that designs created without a complete understanding of the problem are inherently bad, but the likelihood of having stumbled onto the best solution is diminished. Design without context is unpredictable, and in today's compressed software development schedules and increased demand for application capabilities, unpredictability is the enemy.

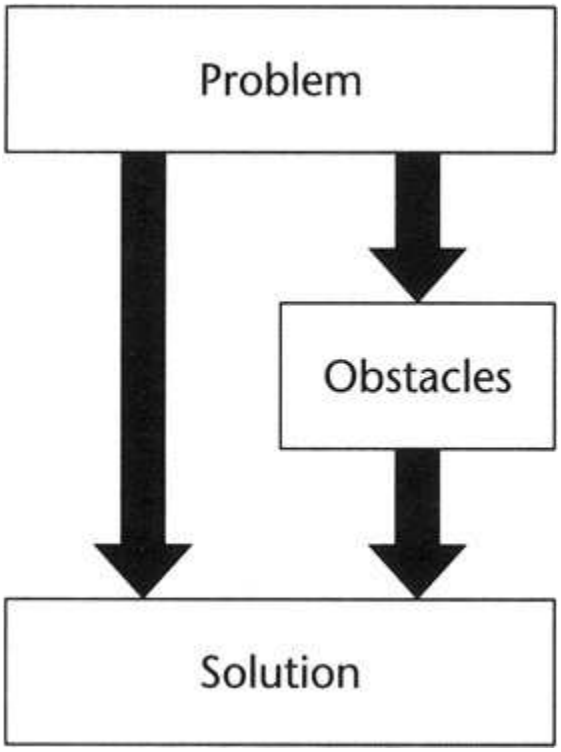

Figure 4.1: Design solution represented as a function of problems and obstacles.

Since design is predominantly creative, it cannot be fully measured and understood. Experience has proved that adopting systematic approaches to design can bring a degree of predictability in terms of the amount of time required to design, as well as the quality of the design. Systematic approaches shouldn't hinder creativity, but rather serve as a catalyst for creative thought.

There is no singular design method that is superior; rather, there are good or poor choices for which method to employ given a unique set of circumstances. The software architect should exercise judgment when using a particular design method for a particular aspect of a project in order to avoid the pitfall of adopting a comprehensive methodology wholesale only later to discover that it wasn't entirely appropriate. Object-oriented analysis and design are popular methodologies but not always the best choice, especially with architectural design, which may not be at the level of objects and classes.

Recall that specifying the function (*utilitas*) of an application is commonly referred to as *analysis* and that specifying the form (*venustas*) is commonly referred to as *design*. Analysis is commonly understood to be the phase or activity where the problem is articulated or structured in order to clarify the problem and to facilitate the application of design methods in order to derive or discover a solution. Software development is characterized as an activity of creating problem-solution pairs. This is why design patterns are so popular, because they capture many known problem-solution pairs. The pattern's context defines some types of obstacles of the problem.

Aristotelian Reasoning

Software design shares some things in common with the philosophical investigations of a subject matter. Whereas logic and philosophy intend to investigate some subject fully, software design stops at practical limits within the context of solving a practical problem. In the article *An Aristotelian Understanding of Object-Oriented Programming*, Rayside and Campbell take the position that achieving conceptual integrity (Brooks, 1975) requires the ability to reason with order, with ease, and without error.

Aristotle's work on logic (the *Organon*) establishes reason as the tool of tools. In the language of software we can consider reason as a meta-tool. Reason can be used as a tool for *understanding* and as a tool for *discoursing*, which is the formation of new truths about something. Thomas Aquinas divided the Organon based on three rational operations used in the formation of knowledge: definition, predication, and inference. Definition and predication are tools of understanding, and inference is a tool of discoursing.

- **Definition.** By applying reason, we can apprehend (literally "reach out and grasp") something. The art of definition is in the grasping of the unity in things, which we express in words. Once we have found the unity of things, we must find what differentiates it from other things. There are two kinds of distinction: accidental and essential. *Essential* distinctions tell us *the kind* of thing it is. *Accidental* distinctions tell us something about the thing other than what it is. In software modeling techniques, it is common to search for the objects or components that comprise the system view and to organize them into classes or types. After these types have been identified, we look for essential distinctions, such as the specific attributes that characterize things in the type, for example, specifying the data members of a class or the set of operations (the interface) of a component type.

  The activity of defining things characterizes analysis (such as object-oriented analysis). Therefore, the goal of analysis is the apprehension of some problem or application domain. Accidental distinctions are not represented in modeling or programming languages. However accidental distinctions are made, sometimes subconsciously, about the classes in a model. For example, statements like "components of type A must not have knowledge of the structure of data in components of type B" are accidental distinctions. Components of type A are not specified or coded with the accidental distinction (there is no way to represent it), but it may be specified in an architectural model using natural language descriptions.

- **Predication.** Predication, the second operation of reason, is also a tool for understanding. This is the operation of creating statements about a subject in which a predicate is either affirmed or denied. A statement either *composes* a predicate with a subject or *divides* a predicate from a subject, and therefore a statement is either true or false. For example, in object-oriented designs the statement "class A is-a class B" is a statement that is either true or false. The predicate (also called a *predicable relation*) is the phrase "is-a class B" and denotes a subclass relationship in this case. The subject class A may be composed with this predicate as above or it may be divided such as in the statement "class A is-not-a class B." In this example, I modified the predicate in the context of the statement to be grammatically correct. However, to be a little more mathematical, I could have said "*not* class A is-a class B," where *not* is an

operator that negates the predicate, thus dividing it from the subject class A. Whereas definition helps us find universal things, predication helps us discover the relationships between those things. If we don't do a very good job at defining our concepts, predication can be difficult.

- **Inference.** The purpose of the third operation of reason is the discovery of new truths about subjects. This operation involves the ordering of statements with the purpose of drawing a conclusion. This ordered set of statements and conclusion is called an *argument*. An argument is either *valid* or *invalid*. The conclusion of an argument is also a statement and is therefore either true or false. In Aristotelian logic, this argument is called a *syllogism*. In architecture, one type of inference operation is the assessment of an architectural design (addressed in Chapter 14).

Designers usually apply these operations in an iterative fashion and at varying levels of abstraction, although they probably are not consciously aware of which operation they are applying. Definition and predication are the basis of abstract thinking. The notion of design as a problem-solving activity is an application of the rational operators.

Recall that one of the most common obstacles to good design is lack of understanding of the problem. By applying the rational operations of definition and predication, the designer is better able to apprehend or grasp the true nature of the problem. However, there is no design operation that can be easily applied, so we rely on our existing body of knowledge contained in our own memories and in design catalogues. Even if we understand the problem, it may not be represented in a way that facilitates the discovery of existing solutions. So we divide and conquer in order to refine our understanding of the problem, looking for more common subproblems (abstractions) that, when synthesized, describe the original problem. Again, we search for solutions to our new set of problems.

For experienced designers, this process may be intuitive and proceed in the subconscious. Less experienced designers may need to apply these steps consciously and systematically until it becomes second nature. In order to master the art of software architecture, the architect needs to practice these techniques until they are part of his or her subconscious.

General Methodology of Design

In this section, we look at the general elements of a design method based on Pahl and Beitz (Pahl, 1996). Not all design methods include all of these elements, but these elements are present in all design methods:

- Purposeful thinking
- Analysis
- Abstraction
- Synthesis
- General heuristics

Purposeful Thinking

Design requires systematic thinking, which separates design from routine tasks. Systematic, or *discursive*, techniques are not the opposite of creative techniques nor are they the opposite of subconscious, or *intuitive*, thought. Relying on intuition alone, however, is not a good design practice so we need to be more deliberate in our approach to design. A purely intuitive approach to design has several disadvantages: The right solution rarely comes at the right time, the results depend on the architect's skills and talents, and the solutions may be negatively influenced by preconceived ideas (also called *fixation*). Intuition can lead to good solutions but requires conscious and deliberate involvement with the problem. Discursive thought can stimulate intuitive thinking. Programmers experience this frequently. A programmer may spend hours or days struggling with some problem only to have the solution come to her while she is away from the computer, not even consciously thinking about the problem.

Design should be primarily discursive; that is, it should proceed deliberately in a stepwise fashion. Unconscious procedures can be transformed into deliberate and systematic methods through systematic rules, clear task formulation, and structured design procedures.

Errors are unavoidable so our processes must allow for this. Therefore the architect should analyze the design for errors or weak points in the early stages of development. Clearly defined requirements and problem statements help reduce the risk of design errors by minimizing the amount of guesswork that goes into the design process as well as minimizing the amount of rework that would otherwise be necessary. A discursive design approach helps reduce errors by enforcing systematic testing of design ideas and assumptions early in the development process. Fixed design ideas (assumptions) should be avoided and existing methods and tools should be adapted as necessary (don't let a given methodology artificially constrain you). The deliberate use of specific methods for failure or error identification should be used.

Creativity is inhibited or encouraged by different influences. Some techniques for encouraging creativity are:

- Interrupt the activity to create incubation periods (but be careful; too many interruptions can be disruptive).
- Apply different solution-finding methods.
- Move from abstract to concrete ideas.
- Find and collect information from design catalogues (such as design patterns).
- Divide work among architecture team members.
- Make realistic plans: Realistic planning is found to encourage motivation and creativity, while unrealistic planning is inhibiting.

Analysis

Analysis is the decomposition of complex systems into elements and their interrelationships, identifying essential distinctions, and discarding accidental distinctions. The activities of analysis include identification, definition, and structuring with the purpose of acquiring information about a subject that can be transformed into knowledge. Analytic methods are useful in all stages of software design, from requirements gathering to evaluating designs and technologies to implementation.

Formulating problems clearly and unambiguously, via analysis of an application domain, helps avoid many expensive design and implementation errors. Careful analysis and formulation of problems are two of the most important and effective tools a software architect has. All design methodologies

have some analysis aspect. However, I find that many modern design methodologies tend to blur the distinction between analysis and design. Many object-oriented methodologies promote a seamless development concept where the problem or application domain is continuously transformed into a solution, for example, Object Modeling Technique (OMT). In practice, the analysis is rarely done or is not done adequately.

Structured analyses in software, such as the techniques of Gane and Sarson, Yourdon, DeMarco, Ward and Mellor, and the Structured Systems Analysis and Design Methodology (SSADM), are methods for formulating and structuring the problem. Structured analysis is a process where the problem is formulated in terms of data flows, as represented in data flow diagrams (DFDs) and in system state transitions as represented in state transition diagrams. These models emphasize a hierarchical structuring of the functions of a system.

Similarly, object-oriented analysis, such as the techniques of Jacobson, Booch, Rumbaugh (OMT), and Yourdon and Coad, are techniques for structuring the problem domain in terms of objects, their interrelationships, and their class hierarchies. Whereas structured analysis emphasizes a functional hierarchy, object orientation emphasizes a data hierarchy.

All systems have weak areas. In established engineering disciplines, weak-spot analysis is performed to discover weak spots in a design. Similarly, in software architecture we can perform weak-spot analysis on a design by evaluating a design against known metamodels. For example, an application user interface design may be evaluated against the arch/slinky metamodel to determine its weaknesses (see Chapter 11). After gaining knowledge about the weakness of a particular design, the architect may make revisions to the design. Weaknesses that are discovered may not be relevant to a given application or system, especially if the weakness doesn't negatively affect the known quality attribute requirements.

Analysis aids in the search for solutions by decomposing a problem into individual, more manageable subproblems. When a problem seems difficult, sometimes a new formulation of the problem may prove to be a better place to start. Most software design problems are the result of a lack of understanding of the problem being solved.

Analysis is the technique whereby we create definitions and identify predicable relations between defined things. A problem domain model will typically consist of definitions or classes (universals) of objects or functions and their relations.

Abstraction

Through abstraction, we can infer more general and comprehensive relationships among the elements of our problem. Abstraction reduces the complexity of a problem while emphasizing the essential characteristics of it and aids in the discovery of solutions. Abstractions are not always invented; sometimes they are discovered by using existing abstraction models and attempting to fit the problem into them.

This is an example of predication: A statement is created about the problem abstraction and an existing model such as "problem A (new) is a form of problem B (existing)." An otherwise seemingly novel problem may turn out to be a variation of an existing problem to which some known solutions already exist. By formulating problem A in terms of problem B, the architect then only needs to find suitable abstractions for the remaining part of problem A. A problem can have many abstractions, and, using a discursive approach, the architect can discover several suitable abstractions and apply many existing problem models. Metamodels or reference models are examples of such reusable abstractions.

Analysis aids in the discovery and evaluation of abstractions. Not all analysis results in abstractions,

but most abstractions are discovered via analysis.

Synthesis

Synthesis is the combining of individual elements or parts to produce a new effect. It is the integration of solutions to subproblems and the evaluation of the resulting system. When we decompose a problem via analysis into sub-problems and discover solutions to those subproblems, there is still the risk that synthesizing those solutions will not solve the larger problem entirely or, worse, the solutions will be incompatible. Existing abstraction models may also be synthesized to form larger, more specialized abstractions. Synthesis is not only applied to the models of software architecture, it is also applied to the design methods.

General Heuristics

In this section, we look at some heuristic design methods. Heuristics are techniques that are characterized as the searching for suitable solutions. These methods apply to many design tasks:

- Persistent questions
- Negation
- Forward steps
- Backward steps
- Factorization
- Systematic variation
- Division of labor and collaboration

The Method of Persistent Questions

This method is useful when applying any systematic procedure such as requirements engineering (systems analysis), architectural design, and architectural evaluation. Requirements are almost never complete or at the right level of abstraction and the architect must help the acquirer to discover the essential requirements of the application they are envisioning during the analysis of the requirements. Persistent questioning stimulates ideas and intuition and helps to eliminate assumptions and preconceived notions. The method of persistent questions is a tool that the architect may use when analyzing the problem to discover useful abstractions, as well as a tool for evaluating solutions. An architect or architecture team may even keep a database of standard sets of questions for various purposes. Such a database of questions helps extend the memory of the architect and transfer useful design methods and knowledge to others. Asking questions is considered one of the most important methodological tools, and the technique is found in many existing product development methodologies.

This method is fundamental to the discursive method, and it seems so obvious that we may not even realize we're using it. This makes it easy to overlook, even though it is the cornerstone of any analytic method.

The Method of Negation

The method of negation is also known as the *method of deliberate negation* and *systematic doubting*.

The method starts with a known solution that is divided into statements (truthful predicable relations) about its individual parts. These statements are then negated. For example, the statement "class A is a class B" is negated. This deliberate inversion of the solution, that class A is not a class B, may lead to new solution possibilities. Consider another example of a system that uses a two-phase commit for transactions. By negating this statement and assuming that the system won't use a two-phase commit approach, we are forced to consider the consequences or some alternatives.

An alternative may be the use of asynchronous message passing and the use of compensating transactions. A variation of this method is the deliberate omission of elements, such as removing the notion of class A altogether and seeing what other solution possibilities arise. This may lead to nothing, but the potential of discovering a different solution is improved.

The Method of Forward Steps

The method of forward steps is also known as *the method of divergent thought*. It starts with a first solution attempt and proceeds to follow as many solution paths as possible, yielding other solutions. The method of forward steps is not necessarily systematic and usually starts with an unsystematic divergence of ideas. For example, the first solution attempt at an application problem may be to apply a client/server architectural style. By applying forward steps, the architect follows several paths such as browser-based client, Web client with browser plug-ins, fat client with embedded browser, or fat client that speaks HTTP with the server.

This method can be applied to any model at any level of abstraction and is typically followed recursively as far down a path as possible. For example, starting with the fat client that speaks HTTP solution, additional application of the method of forward steps could yield solutions like fat client with local storage, fat client with server-side storage only, and fat client with heterogeneous local and server-side storage. The process can follow any path. This is a good way to brainstorm over an object model of the application domain as well as the choice of technologies in the solution domain. This method is similar to the method of systematic variation.

The Method of Backward Steps

The method of backward steps is also known as the method of convergent thought. In this approach, the architect starts with a goal in mind rather than the initial problem. This technique is most useful for setting up an engineering design process for a product and a product development plan after the architecture has been determined. Starting with the final objective of the development effort, all (or a reasonable subset of) possible paths that could have led up to the objective are retraced. This method is useful not only in preparing an engineering process but also for organizing an engineering department around the architecture of a product (known as Conway's law).

The Method of Factorization

This method is the basis of the refactoring technique, popular among software engineers when reworking existing source code to make it more readable, manageable, maintainable, and reusable. The method involves breaking a complex system into less complex elements or *factors*. Factorization is a type of analysis method. In architecting, this technique is used to find the essential problems being solved (for example, factoring the objectives into subobjectives). Just as in mathematics, the factorization may not be obvious and requires some insight, skill, and lots of practice. In algebra, you may factor an equation by introducing zero as a factor. Similarly, finding that one of the factors of a publishing system is a file-versioning repository may not be obvious (although the experienced architect may see the factorization immediately). Another example is taking what may appear to be an indivisible component and finding a way to divide it into two components and an interface.

The Method of Systematic Variation

The method of systematic variation of solutions may have originated with Leonardo DaVinci. DaVinci kept meticulous notes on his ideas and inventions and applied this technique to derive multiple variations of ideas, each varying only by a single characteristic from the former. The method starts with a generalized classification structure, such as a class hierarchy, that represents the various problem characteristics and possible solutions. By systematically varying a single characteristic, the architect may discover more optimized solutions. The approach yields a *solution field*. Evaluation techniques can be applied to each solution to determine its suitability.

Division of Labor and Collaboration

The study of human factors has found that implementing large and complex tasks requires a division of labor. The more specialized the task the more important the division of labor. Software development tasks are commonly based not only on functional areas of the system but also on the technology used. We typically have user interface designers, user interface engineers, middle-tier engineers, back-end or database engineers, test engineers, configuration management engineers, and so on. A complex application or system also benefits from a division of labor with respect to the architecture. For example, it seems natural to separate the tasks of requirements engineering, functional specification, interaction design, and structural design, all of which are in the scope of architecture. However, whenever there is a division of labor there is an information exchange problem. Systematic methods and the creation of models can help overcome this problem.

PREVIOUS NEXT
PREVIOUS NEXT

Summary

Design is a creative problem-solving activity that involves finding or creating solutions to problems given a set of obstacles. Architecting typically starts by formulating the problems that need to be solved. The three Aristotelian operations of reason can be used in formulating problems and rationalizing about their solutions. The operations of definition and predication are used to gain understanding of the problem. Definition is the creation of universals and predication is the creation of Boolean statements about the universals. The last operation of reason, inference, is used to discover new truths about universals and their relationships. Inferences are arguments, which are ordered statements and a conclusion. The conclusion is also a Boolean statement.

There are three principles of architecture: *utilitas* (*function*), *firmitas* (fabrication, quality), and *venustas* (form). These principles form an interdependent triad. The function of an application is a different, though related, concept than the functional specification of an application. The intended function of the application is expressed as problems and, through the application of analysis, is divided into more manageable, less complex subproblems. Analysis involves the rational operations of definition and predication. The first phase of design, product planning, is to transform these subproblems into a functional specification. The functional specification or behavioral model of an application is one aspect of the architectural design that is created in order to better understand the intended function or need of the application. During conceptual design the functional design and structural design are elaborated. Functional design corresponds to the externally facing design or interaction design of an application or system. This is the aspect of the architecture that is perceived by users. Interaction design focuses on the creation of a virtuality that reduces cognitive friction. Structural design is concerned with the internal physical design of the software itself as embodied in source code and other electronic artifacts that are compiled into an executable system. Heuristic design methods, such as the method of forward steps and the method of systematic variation, are applied to produce a solution field consisting of several candidate structural architectural designs called solution principles, or concepts. Conceptual design artifacts are not purely logical models; they also specify working principles, which may include the identification of technologies or design patterns that can be used to solve the problem. The methods applied during conceptual design

involve the application of inference to discover new solutions and to evaluate existing solutions.

The architecture design or design candidates are evaluated, disposed of, or further refined during embodiment design. The same rational operations and design methods are applied but now the focus is more on elaboration of the solution and less on formulation of the problem. Detail design involves the specification of layout and in software is the activity of implementation and testing.

In many software development methodologies, a distinction between the conceptual phase and the embodiment phase is not explicit and is usually referred to simply as design, as in analysis and design. Many methodologies, such as OMT, promote a seamless development model where the models of analysis and design are combined into one evolving model that starts as a formulation of the problem and evolves into a solution model.

In Chapter 5, we take a detailed look at the concept of complexity and modularity and how it affects the architect's understanding of the problem and solution. The modular operators are presented, which are design operations that can be applied, together with the design methods. In Chapters 6 and 7, we see how design decisions are represented as models and architectural views of a system.

PREVIOUS NEXT