UNIT III SOFTWARE DESIGN

SOFTWARE DESIGN

Software design encompasses the set of principles, concepts, and practices that lead to the development of a high-quality system or product. Design principles establish an overriding philosophy that guides you in the design work you must perform. Design concepts must be understood before the mechanics of design practice are applied, and design practice itself leads to the creation of various representations of the software that serve as a guide for the construction activity that follows. Design is pivotal to successful software engineering.

Design Concepts

What is it? Design is what almost every engineer wants to do. It is the place where creativity rules—where stakeholder requirements, business needs, and technical considerations all come together in the formulation of a product or system. Design creates a representation or model of the software, but unlike the requirements model (that focuses on describing required data, function, and behavior), the design model provides detail about software architecture, data structures, interfaces, and components that are necessary to implement the system.

Who does it? Software engineers conduct each of the design tasks.

Why is it important? Design allows you to model the system or product that is to be built. This model can be assessed for quality and improved before code is generated, tests are conducted, and end users become involved in large numbers. Design is the place where software quality is established.

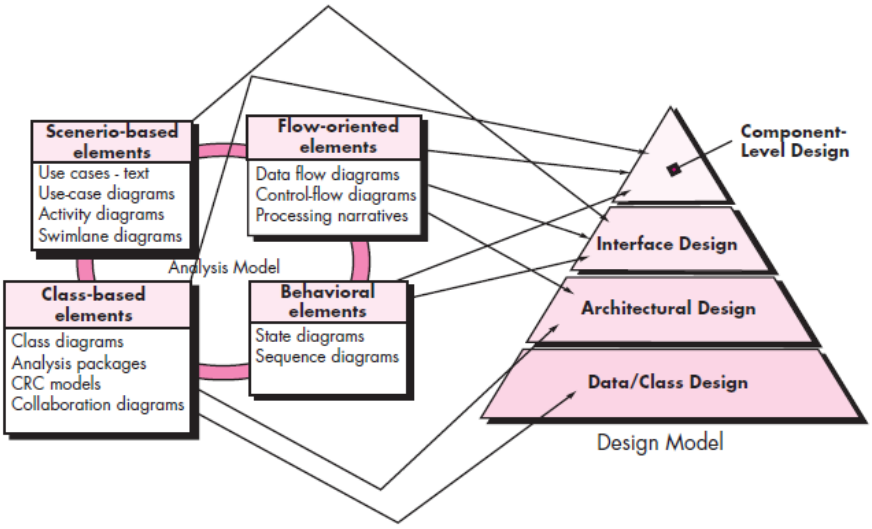
What are the steps? Design depicts the software in a number of different ways. First, the architecture of the system or product must be represented. Then, the interfaces that connect the software to end users, to other systems and devices, and to its own constituent components are modeled. Finally, the software components that are used to construct the system are designed. Each of these views represents a different design action, but all must conform to a set of basic design concepts that guide software design work.

What is the work product? A design model that encompasses architectural, interface, component level, and deployment representations is the primary work product that is produced during software design.

How do one can ensure that they done it right? The design model is assessed by the software team in an effort to determine whether it contains errors, inconsistencies, or omissions; whether better alternatives exist; and whether the model can be implemented within the constraints, schedule, and cost that have been established. The goal of design is to produce a model or representation that exhibits firmness, commodity, and delight. To accomplish this, you must practice diversification and then convergence.

Design within the Context of Software Engineering

Software design sits at the technical kernel of software engineering and is applied regardless of the software process model that is used. Beginning once software requirements have been analyzed and modeled, software design is the last software engineering action within the modeling activity and sets the stage for **construction** (code generation and testing).



Each of the elements of the requirements model provides information that is necessary to create the four design models required for a complete specification of design. The flow of information during software design is illustrated in the above Figure .The requirements model, manifested by scenario-based, class-based, flow-oriented, and behavioral elements, feed the design task. Using design notation and design methods discussed in later chapters, design produces a data/class design, an architectural design, an interface design, and a component design.

The data/class design transforms class models into design class realizations and the requisite data structures required to implement the software. The objects and relationships defined in the CRC diagram and the detailed data content depicted by class attributes and other notation provide the basis for the data design action. Part of class design may occur in conjunction with the design of software architecture. More detailed class design occurs as each software component is designed.

The architectural design defines the relationship between major structural elements of the software, the architectural styles and design patterns that can be used to achieve the requirements defined for the system, and the constraints that affect the way in which architecture can be implemented. The architectural design representation—the framework of a computer-based system—is derived from the requirements model.

The interface design describes how the software communicates with systems that interoperate with it, and with humans who use it. An interface implies a flow of information (e.g., data and/or control) and a specific type of behavior. Therefore, usage scenarios and behavioral models provide much of the information required for interface design.The component-level design transforms structural elements of the software architecture into a

procedural description of software components. Information obtained from the class-based models, flow models, and behavioral models serve as the basis for component design.

During design you make decisions that will ultimately affect the success of software construction and, as important, the ease with which software can be maintained. But why is design so important? The importance of software design can be stated with a single word—*quality*.

Design is the place where quality is fostered in software engineering. Design provides you with representations of software that can be assessed for quality. Design is the only way that you can accurately translate stakeholder's requirements into a finished software product or system. Software design serves as the foundation for all the software engineering and software support activities that follow. Without design, you risk building an unstable system—one that will fail when small changes are made; one that may be difficult to test; one whose quality cannot be assessed until late in the software process, when time is short and many dollars have already been spent.

THE DESIGN PROCESS

Software design is an iterative process through which requirements are translated into a "blueprint" for constructing the software. Initially, the blueprint depicts a holistic view of software. That is, the design is represented at a high level of abstraction a level that can be directly traced to the specific system objective and more detailed data, functional, and behavioral requirements. As design iterations occur, subsequent refinement leads to design representations at much lower levels of abstraction. These can still be traced to requirements, but the connection is more subtle.

Software Quality Guidelines and Attributes

Throughout the design process, the quality of the evolving design is assessed with a series of technical reviews that suggests three characteristics that serve as a guide for the evaluation of a good design:

- The design must implement all of the explicit requirements contained in the requirements model, and it must accommodate all of the implicit requirements desired by stakeholders.
- The design must be a readable, understandable guide for those who generate code and for those who test and subsequently support the software.
- The design should provide a complete picture of the software, addressing the data, functional, and behavioral domains from an implementation perspective.
- Each of these characteristics is actually a goal of the design process. But how is each of these goals achieved?

Quality Guidelines:

In order to evaluate the quality of a design representation, The other members of the software team must establish technical criteria for good design. For the time being, consider the following guidelines:

1.A design should exhibit an architecture that has been created using recognizable architectural styles or patterns, and is composed of components that exhibit good design .It can be implemented in an evolutionary fashion, thereby facilitating implementation and testing.

2.A design should be modular; that is, the software should be logically partitioned into elements or subsystems.

3.A design should contain distinct representations of data, architecture, interfaces, and components.

4.A design should lead to data structures that are appropriate for the classes to be implemented and are drawn from recognizable data patterns.

5.A design should lead to components that exhibit independent functional characteristics.

6. A design should lead to interfaces that reduce the complexity of connections between components and with the external environment.

7. A design should be derived using a repeatable method that is driven by information

obtained during software requirements analysis.

8. A design should be represented using a notation that effectively communicates its meaning.

These design guidelines are not achieved by chance. They are achieved through the application of fundamental design principles, systematic methodology, and thorough review.

Quality Attributes.

Hewlett-Packard developed a set of software quality attributes that has been given the acronym FURPS—functionality, usability, reliability, performance, and supportability. The FURPS quality attributes represent a target for all software design:

Functionality is assessed by evaluating the feature set and capabilities of the program, the generality of the functions that are delivered, and the security of the overall system.

Usability is assessed by considering human factors , overall aesthetics, consistency, and documentation.

Reliability is evaluated by measuring the frequency and severity of failure, the accuracy of output results, the mean-time-to-failure (MTTF), the ability to recover from failure, and the predictability of the program.

Performance is measured by considering processing speed, response time,

resource consumption, throughput, and efficiency.

Supportability combines the ability to extend the program (extensibility), adaptability, serviceability—these three attributes represent a more common term,

maintainability—and in addition, testability, compatibility, configurability the ease with which a system can be installed, and the ease with which problems can be localized.

Not every software quality attribute is weighted equally as the software design is developed. One application may stress functionality with a special emphasis on security. Another may demand performance with particular emphasis on processing speed. A third might focus on reliability. Regardless of the weighting, it is important to note that these quality attributes must be considered as design commences, *not* after the design is complete and construction has begun.

The Evolution of Software Design

The evolution of software design is a continuing process that has now spanned almost six decades. Early design work concentrated on criteria for the development of modular programs and methods for refining software structures in a top down manner. Procedural aspects of design definition evolved into a philosophy called *structured programming.*

Later work proposed methods for the translation of data flow or data structure, into a design definition. Newer design approaches proposed an object-oriented approach to design derivation. More recent emphasis in software design has been on software architecture and the design patterns that can be used to implement software architectures and lower levels of design abstractions. Growing emphasis on aspect-oriented methods, model-driven development, and test-driven development emphasize techniques for achieving more effective modularity and architectural structure in the designs that are created.

A number of design methods, growing out of the work just noted, are being applied throughout the industry, each software design method introduces unique heuristics and notation, as well as a somewhat parochial view of what characterizes design quality. Yet, all of the methods have a number of common characteristics:

(1) a mechanism for the translation of the requirements model into a design representation,

(2) a notation for representing functional components and their interfaces,

(3) heuristics for refinement and partitioning, and

(4) guidelines for quality assessment.

Regardless of the design method that is used, you should apply a set of basic concepts to data, architectural, interface, and component-level design.

Design Concepts

A set of fundamental software design concepts has evolved over the history of software

engineering. Although the degree of interest in each concept has varied over the years, each has stood the test of time. Each provides the software designer with a foundation from which more sophisticated design methods can be applied. Each helps you answer the following questions:

- What criteria can be used to partition software into individual components?
- How is function or data structure detail separated from a conceptual representation of the software?
- What uniform criteria define the technical quality of a software design?

Abstraction

When you consider a modular solution to any problem, many levels of abstraction can be posed. At the highest level of abstraction, a solution is stated in broad terms using the language of the problem environment. At lower levels of abstraction, a more detailed description of the solution is provided. Problem-oriented terminology is coupled with implementation-oriented terminology in an effort to state a solution. Finally, at the lowest level of abstraction, the solution is stated in a manner that can be directly implemented. As different levels of abstraction are developed, you work to create both procedural and data abstractions. A procedural abstraction refers to a sequence of instructions that have a specific and limited function. The name of a procedural abstraction implies these functions, but specific details are suppressed. An example of a procedural abstraction would be the word open for a door. Open implies a long sequence of procedural steps (e.g., walk to the door, reach out and grasp knob, turn knob and pull door, step away from moving door, etc.).

A data abstraction is a named collection of data that describes a data object. In the context of the procedural abstraction open, we can define a data abstraction called door. Like any data object, the data abstraction for door would encompass a set of attributes that describe the door (e.g., door type, swing direction, opening mechanism, weight, dimensions). It follows that the procedural abstraction open would make use of information contained in the attributes of the data abstraction door.

Architecture

Software architecture alludes to “the overall structure of the software and the ways in which that structure provides conceptual integrity for a system” .In its simplest form, architecture is the structure or organization of program components (modules), the manner in which these components interact, and the structure of data that are used by the components. In a broader sense, however, components can be generalized to represent major system elements and their interactions. One goal of software design is to derive an architectural rendering of a system. This rendering serves as a framework from which more detailed design activities are conducted. A set of architectural patterns enables a software engineer to solve common design problems.

A set of properties that should be specified as part of an architectural design:

- Structural properties.

This aspect of the architectural design representation defines the components of a system (e.g., modules, objects, filters) and the manner in which those components are packaged and interact with one another. For example, objects are packaged to encapsulate both data and the processing that manipulates the data and interact via the invocation of methods.

- Extra-functional properties.

The architectural design description should address how the design architecture achieves requirements for performance, capacity, reliability, security, adaptability, and other system characteristics.

- Families of related systems.

The architectural design should draw upon repeatable patterns that are commonly encountered in the design of families of similar systems. In essence, the design should have the ability to reuse architectural building blocks.

Given the specification of these properties, the architectural design can be represented using one or more of a number of different models. Structural models represent architecture as an organized collection of program components.

Framework models increase the level of design abstraction by attempting to identify repeatable architectural design frameworks that are encountered in similar types of applications.

Dynamic models address the behavioral aspects of the program architecture, indicating how the structure or system configuration may change as a function of external events. Process models focus on the design of the business or technical process that the system must accommodate. Finally, functional models can be used to represent the functional hierarchy of a system.

A number of different architectural description languages (ADLs) have been developed to represent these models. Although many different ADLs have been proposed, the majority provide mechanisms for describing system components and the manner in which they are connected to one another.

Patterns

A pattern is a named nugget of insight which conveys the essence of a proven solution to a recurring problem within a certain context amidst competing concerns. Stated in another way, a design pattern describes a design structure that solves a particular design problem within a specific context and amid “forces” that may have an impact on the manner in which the pattern is applied and used.

The intent of each design pattern is to provide a description that enables a designer to determine :

(1) whether the pattern is applicable to the current work,

(2) whether the pattern can be reused (hence, saving design time), and

(3) whether the pattern can serve as a guide for developing a similar, but functionally or structurally different pattern.

Separation of Concerns

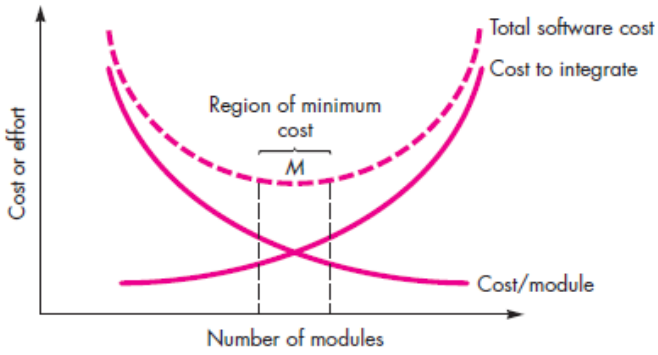
Separation of concerns is a design concept that suggests that any complex problem can be more easily handled if it is subdivided into pieces that can each be solved and/or optimized independently. A concern is a feature or behavior that is specified as part of the requirements model for the software. By separating concerns into smaller, and therefore more manageable pieces, a problem takes less effort and time to solve.

For two problems, p1 and p2, if the perceived complexity of p1 is greater than the perceived complexity of p2, it follows that the effort required to solve p1 is greater than the effort required to solve p2. As a general case, this result is intuitively obvious. It does take more time to solve a difficult problem.

It also follows that the perceived complexity of two problems when they are combined is often greater than the sum of the perceived complexity when each is taken separately. This leads to a divide-and-conquer strategy—it's easier to solve a complex problem when you break it into manageable pieces. This has important implications with regard to software modularity. Separation of concerns is manifested in other related design concepts: modularity, aspects, functional independence, and refinement. Each will be discussed in the subsections that follow.

Modularity

Modularity is the most common manifestation of separation of concerns. Software is divided into separately named and addressable components, sometimes called modules, that are integrated to satisfy problem requirements. It has been stated that "modularity is the single attribute of software that allows a program to be intellectually manageable". Monolithic software (i.e., a large program composed of a single module) cannot be easily grasped by a software engineer. The number of control paths, span of reference, number of variables, and overall complexity would make understanding close to impossible. Referring to the below diagram, the effort (cost) to develop an individual



software module does decrease as the total number of modules increases. Given the same set of requirements, more modules means smaller individual size. However, as the number of modules grows, the effort (cost) associated with integrating the modules also grows. These characteristics lead to a total cost or effort curve shown in the figure. There is a number, M, of modules that would result in minimum development cost, but we do not have the necessary sophistication to predict M with assurance. The curves shown in the figure do provide useful qualitative guidance when modularity is considered. You should modularize, but care should be taken to stay in the vicinity of M. Under modularity or over modularity should be avoided.

You modularize a design (and the resulting program) so that development can be more easily planned; software increments can be defined and delivered; changes can be more easily accommodated; testing and debugging can be conducted more efficiently, and long-term maintenance can be conducted without serious side effects.

Information Hiding

The concept of modularity leads you to a fundamental question: "How do I decompose a software solution to obtain the best set of modules?" The principle of information hiding suggests that modules be "characterized by design decisions that (each) hides from all others." In other words, modules should be specified and designed so that information (algorithms and data) contained within a module is inaccessible to other modules that have no need for such information. Hiding implies that effective modularity can be achieved by defining a set of independent modules that communicate with one another only that information necessary to achieve software function. Abstraction helps to define the procedural (or informational) entities that make up the software. Hiding defines and enforces access constraints to both procedural detail within a module and any local data structure used by the module.

The use of information hiding as a design criterion for modular systems provides the greatest benefits when modifications are required during testing and later during software maintenance. Because most data and procedural detail are hidden from other parts of the software, inadvertent errors introduced during modification are less likely to propagate to other locations within the software.

Functional Independence

The concept of functional independence is a direct outgrowth of separation of concerns, modularity, and the concepts of abstraction and information hiding.

Functional independence is achieved by developing modules with “singleminded” function and an “aversion” to excessive interaction with other modules. Stated another way, you should design software so that each module addresses a specific subset of requirements and has a simple interface when viewed from other parts of the program structure. It is fair to ask why independence is important. Software with effective modularity, that is, independent modules, is easier to develop because function can be compartmentalized and interfaces are simplified (consider the ramifications when development is conducted by a team). Independent modules are easier to maintain (and test) because secondary effects caused by design or code modification are limited, error propagation is reduced, and reusable modules are possible. To summarize, functional independence is a key to good design, and design is the key to software quality.

Independence is assessed using two qualitative criteria:

- cohesion and
- coupling.
    - Cohesion is an indication of the relative functional strength of a module.
    - Coupling is an indication of the relative interdependence among modules.

Cohesion is a natural extension of the information-hiding. A cohesive module performs a single task, requiring little interaction with other components in other parts of a program. Stated simply, a cohesive module should (ideally) do just one thing. Although you should always strive for high cohesion (i.e., single-mindedness), it is often necessary and advisable to have a software component perform multiple functions.

However, “schizophrenic” components (modules that perform many unrelated functions) are to be avoided if a good design is to be achieved. Coupling is an indication of interconnection among modules in a software structure. Coupling depends on the interface complexity between modules, the point at which entry or reference is made to a module, and what data pass across the interface. In software design, you should strive for the lowest possible coupling. Simple connectivity among modules results in software that is easier to understand and less prone to a “ripple effect”, caused when errors occur at one location and propagate throughout a system.

Refinement

Stepwise refinement is a top-down design strategy. A program is developed by successively refining levels of procedural detail. A hierarchy is developed by decomposing a macroscopic statement of function (a procedural abstraction) in a stepwise fashion until programming language statements are reached.

Refinement is actually a process of elaboration. You begin with a statement of function (or description of information) that is defined at a high level of abstraction. That is, the statement

describes function or information conceptually but provides no information about the internal workings of the function or the internal structure of the information. You then elaborate on the original statement, providing more and more detail as each successive refinement (elaboration) occurs.

Abstraction and refinement are complementary concepts. Abstraction enables you to specify procedure and data internally but suppress the need for "outsiders" to have knowledge of low-level details. Refinement helps you to reveal low-level details as design progresses. Both concepts allow you to create a complete design model as the design evolves.

Aspects

As requirements analysis occurs, a set of "concerns" is uncovered. These concerns "include requirements, use cases, features, data structures, quality-of-service issues, variants, intellectual property boundaries, collaborations, patterns and contracts".Ideally, a requirements model can be organized in a way that allows you to isolate each concern (requirement) so that it can be considered independently. In practice, however, some of these concerns span the entire system and cannot be easily compartmentalized.

As design begins, requirements are refined into a modular design representation. Consider two requirements, A and B. Requirement A crosscuts requirement B "if a software decomposition [refinement] has been chosen in which B cannot be satisfied without taking A into account".

For example, consider two requirements for the SafeHomeAssured.com WebApp. A design refinement would focus on those modules that would enable a registered user to access video from cameras placed throughout a space. Requirement B is a generic security requirement that states that a registered user must be validated prior to using SafeHomeAssured.com. This requirement is applicable for all functions that are available to registered SafeHome users. As design refinement occurs, A* is a design representation for requirement A and B* is a design representation for requirement B. Therefore, A* and B* are representations of concerns, and B* crosscuts A*. An aspect is a representation of a crosscutting concern. Therefore, the design representation, B*, of the requirement a registered user must be validated prior to using SafeHomeAssured.com, is an aspect of the SafeHome WebApp.

It is important to identify aspects so that the design can properly accommodate them as refinement and modularization occur. In an ideal context, an aspect is implemented as a separate module (component) rather than as software fragments that are "scattered" or "tangled" throughout many components.

To accomplish this, the design architecture should support a mechanism for defining an aspect—a module that enables the concern to be implemented across all other concerns that it crosscuts.

Refactoring

Refactoring is a reorganization technique that simplifies the design (or code) of a component without changing its function or behavior. Fowler defines refactoring in the following manner: “Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code [design] yet improves its internal structure.” When software is refactored, the existing design is examined for redundancy, unused design elements, inefficient or unnecessary algorithms, poorly constructed or inappropriate data structures, or any other design failure that can be corrected to yield a better design.

Object-Oriented Design Concepts

The object-oriented (OO) paradigm is widely used in modern software engineering.(such as classes and objects, inheritance, messages, and polymorphism)

Design Classes

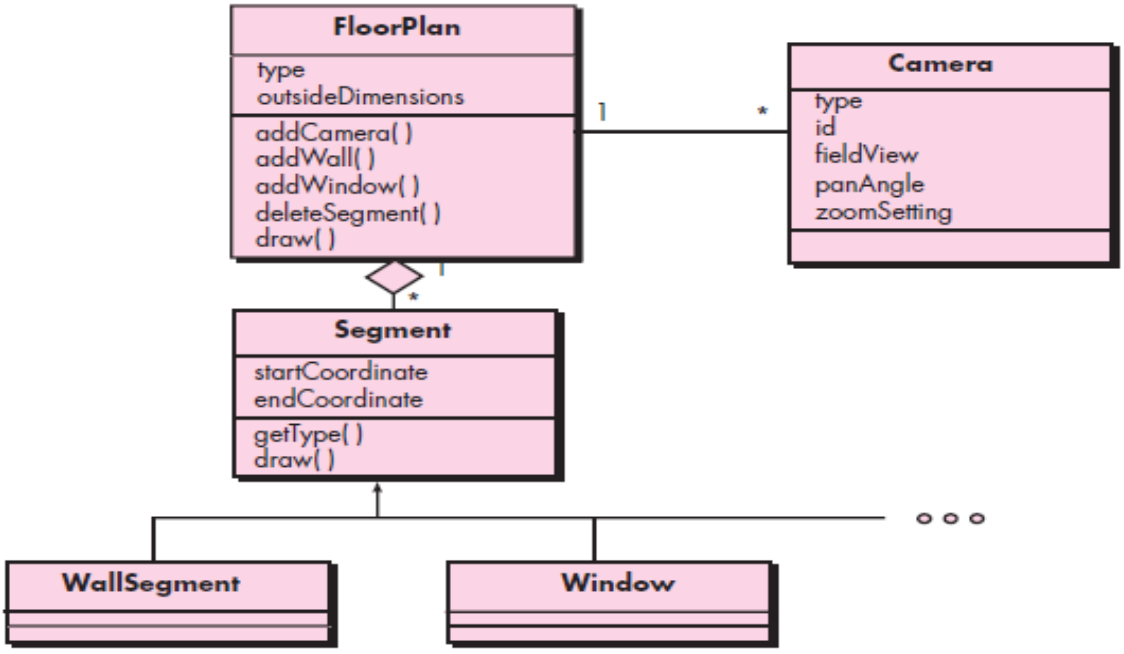
The requirements model defines a set of analysis classes. Each describes some element of the problem domain, focusing on aspects of the problem that are user visible. The level of abstraction of an analysis class is relatively high. As the design model evolves, you will define a set of design classes that refine the analysis classes by providing design detail that will enable the classes to be implemented, and implement a software infrastructure that supports the business solution. Five different types of design classes, each representing a different layer of the design architecture, can be developed:

- User interface classes define all abstractions that are necessary for human computer interaction (HCI). In many cases, HCI occurs within the context of a metaphor (e.g., a checkbook, an order form, a fax machine), and the design classes for the interface may be visual representations of the elements of the metaphor.
- Business domain classes are often refinements of the analysis classes defined earlier. The classes identify the attributes and services (methods) that are required to implement some element of the business domain.
- Process classes implement lower-level business abstractions required to fully manage the business domain classes.
- Persistent classes represent data stores (e.g., a database) that will persist beyond the execution of the software.
- System classes implement software management and control functions that enable the system to operate and communicate within its computing environment and with the outside world.

As the architecture forms, the level of abstraction is reduced as each analysis class is transformed into a design representation. That is, analysis classes represent data objects (and associated services that are applied to them) using the jargon of the business domain. Design classes present significantly more technical detail as a guide for implementation.

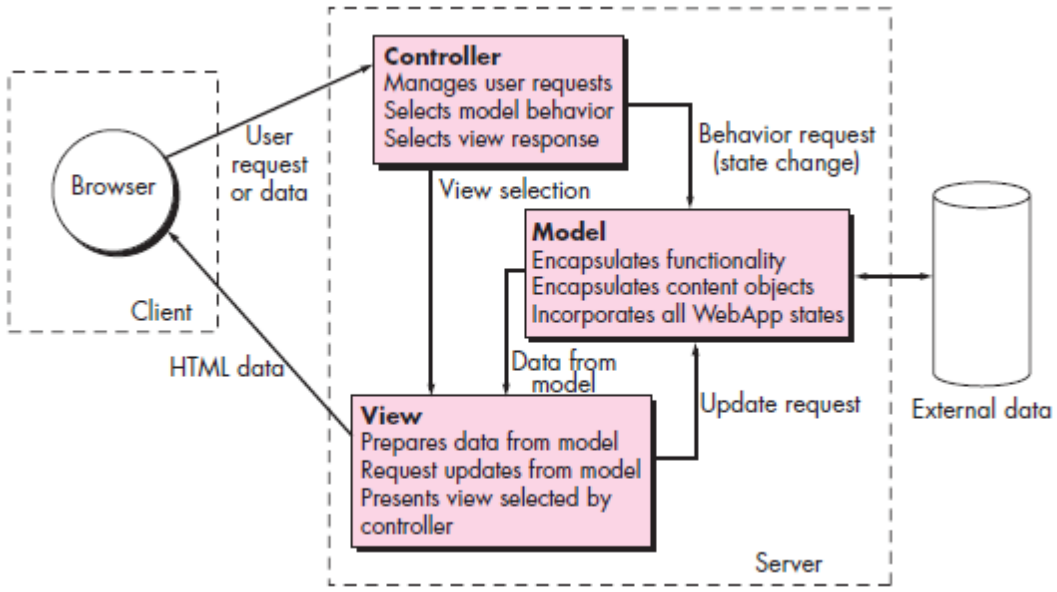
Four characteristics of a well-formed design class:

- Complete and sufficient.
- Primitiveness.
- High cohesion.
- Low coupling.



WebApp Architecture

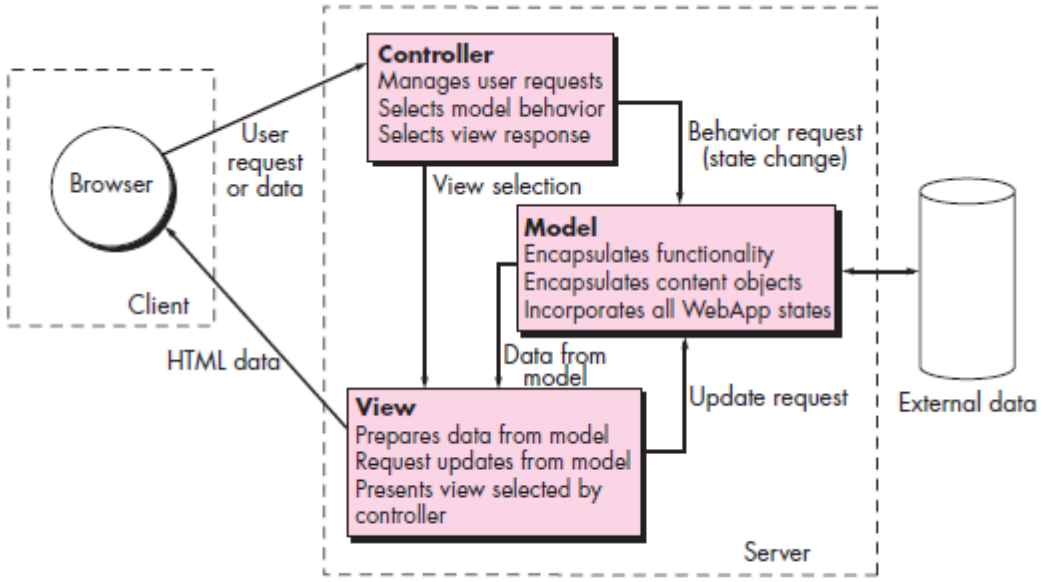
WebApp architecture describes an infrastructure that enables a Web-based system or application to achieve its business objectives. The basic characteristics of this infrastructure in the following manner: Applications should be built using layers in which different concerns are taken into account; in particular, application data should be separated from the page's contents (navigation nodes) and these contents, in turn, should be clearly separated from the interface look-and-feel (pages). The authors suggest a three-layer design architecture that decouples interface from navigation and from application behavior. They argue that keeping interface, application, and navigation separate simplifies implementation and enhances reuse. The Model-View-Controller (MVC) architecture is one of a number of suggested WebApp infrastructure models that decouple the user interface from the WebApp functionality and informational content.



The model (sometimes referred to as the "model object") contains all application-specific content and processing logic, including all content objects, access to external data/information sources, and all processing functionality that is application specific. The view contains all interface specific functions and enables the presentation of content and processing logic, including all content objects, access to external data/information sources, and all processing functionality required by the end user. The controller manages access to the model and the view and coordinates the flow of data between them. In a WebApp, "the view is updated by the controller with data from the model based on user input" [WMT02]. A schematic representation of the MVC architecture is shown in Figure 13.8. Referring to the figure, user requests or data are handled by the controller. The controller also selects the view object that is applicable based on the user request. Once the type of request is determined, a behavior request is transmitted to the model, which implements the functionality or retrieves the content required to accommodate the request. The model object can access data stored in a corporate database, as part of a local data store, or as a collection of independent files. The data developed by the model must be formatted and organized by the appropriate view object and then transmitted from the application server back to the client-based browser for display on the customer's machine. In many cases, WebApp architecture is defined within the context of the development environment in which the application is to be implemented. If you have further interest, see [Fow03] for a discussion of development environments and their role in the design of Web application architectures.

WEBAPP DESIGN

The MVC architecture decouples the user interface from WebApp functionality and information content.



The MVC Architecture

Architectural Styles

The software that is built for computer-based systems also exhibits one of many architectural styles. Each style describes a system category that encompasses

(1) a set of components (e.g., a database, computational modules) that perform a function required by a system;

(2) a set of connectors that enable "communication, coordination and cooperation" among components;

(3) constraints that define how components can be integrated to form the system; and

(4)semantic models that enable a designer to understand the overall properties of a system by analyzing the known properties of its constituent parts.

An architectural style is a transformation that is imposed on the design of an entire system. The intent is to establish a structure for all components of the system. In the case where an existing architecture is to be reengineered, the imposition of an architectural style will result in fundamental changes to the structure of the software including a reassignment of the functionality of components.An architectural pattern, like an architectural style, imposes a transformation on the design of an architecture. However, a pattern differs from a style in a number of fundamental ways:

(1) the scope of a pattern is less broad, focusing on one aspect of the architecture rather than the architecture in its entirety;

(2) a pattern imposes a rule on the architecture, describing how the software will handle some aspect of its functionality at the infrastructure level (e.g., concurrency);

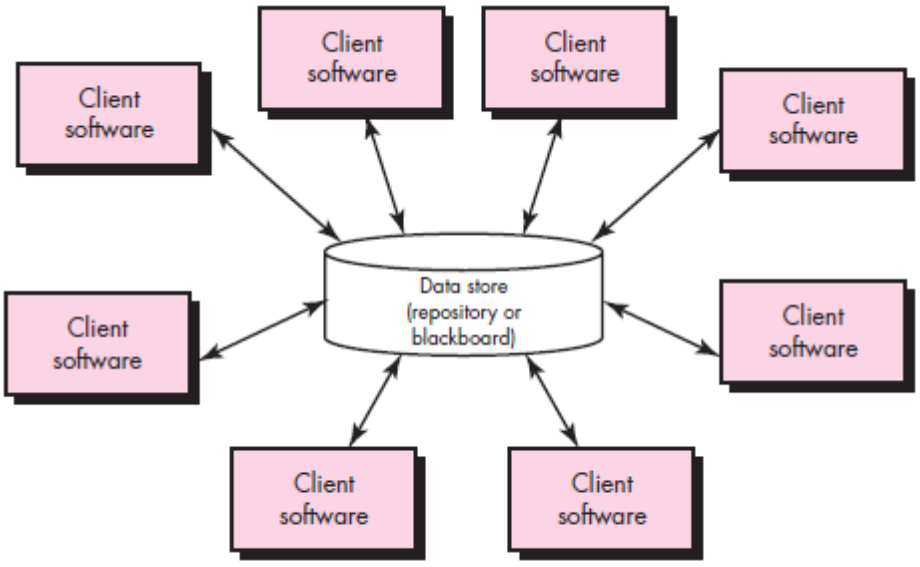
(3) architectural patterns tend to address specific behavioral issues within the context of the architecture (e.g., how real-time applications handle synchronization or interrupts).

Patterns can be used in conjunction with an architectural style to shape the overall structure of a system.

A Brief Taxonomy of Architectural Styles

Although millions of computer-based systems have been created over the past 60 years, the vast majority can be categorized into one of a relatively small number of architectural styles:

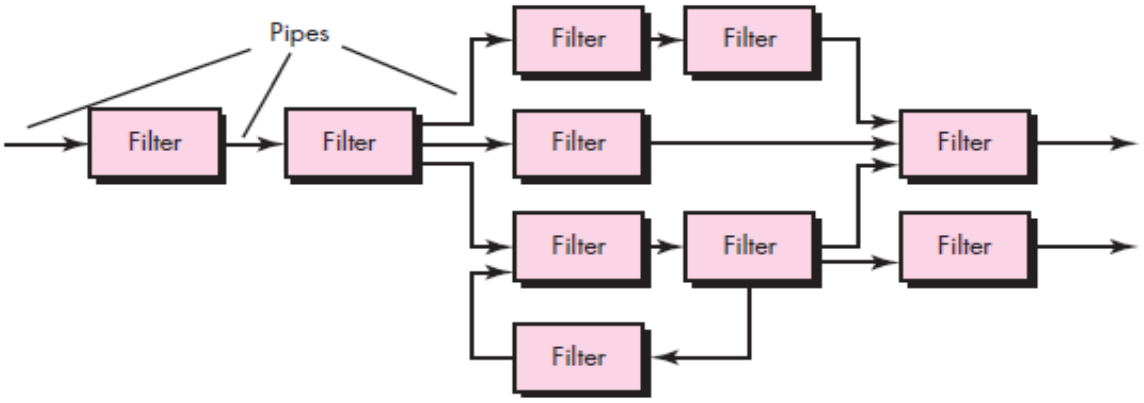
Data-centered architectures: A data store (e.g., a file or database) resides at the center of this architecture and is accessed frequently by other components that update, add, delete, or otherwise modify data within the store. Figuren illustrates a typical data-centered style. Client software accesses a central repository. In some cases the data repository is passive. That is, client software accesses the data independent of any changes to the data or the actions of other client software. A variation on this approach transforms the repository into a "blackboard" that sends notifications to client software when data of interest to the client changes.



Data-centered architectures promote integrability. That is, existing components can be changed and new client components added to the architecture without concern about other clients (because the client components operate independently). In addition, data can be passed

among clients using the blackboard mechanism (i.e., the blackboard component serves to coordinate the transfer of information between clients). Client components independently execute processes.

Data-flow architectures. This architecture is applied when input data are to be transformed through a series of computational or manipulative components into output data. A pipe-and-filter pattern has a set of components, called filters, connected by pipes that transmit data from one component to the next. Each filter works independently of those components upstream and downstream, is designed to expect data input of a certain form, and produces data output (to the next filter) of a specified form. However, the filter does not require knowledge of the workings of its neighboring filters. If the data flow degenerates into a single line of transforms, it is termed batch sequential. This structure accepts a batch of data and then applies a series of sequential components (filters) to transform it.

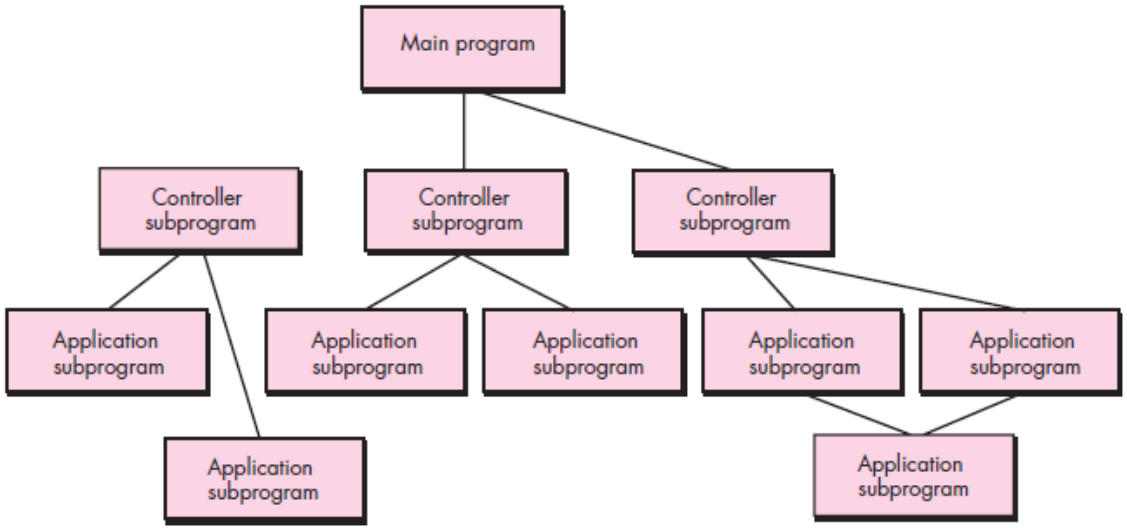


Pipes and filters

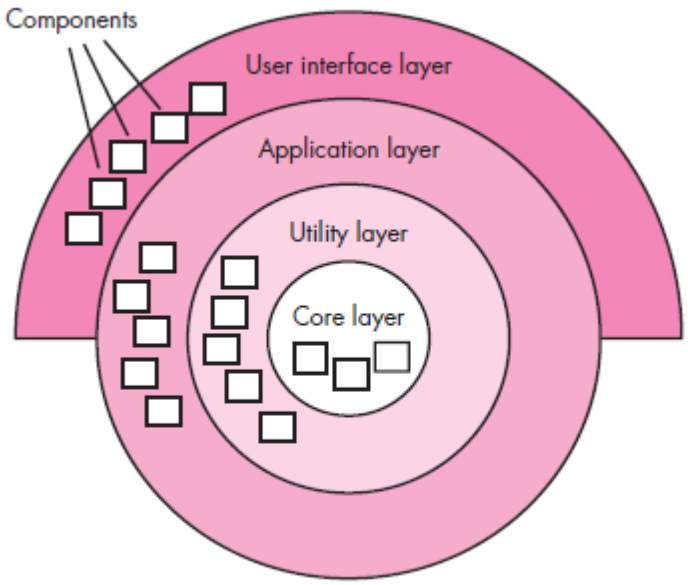
Call and return architectures. This architectural style enables you to achieve a program structure that is relatively easy to modify and scale. A number of sub styles exist within this category:

Main program/subprogram architectures. This classic program structure decomposes function into a control hierarchy where a “main” program invokes a number of program components that in turn may invoke still other components. Figure illustrates an architecture of this type.

Remote procedure call architectures. The components of a main program/subprogram architecture are distributed across multiple computers on a network.



Object-oriented architectures. The components of a system encapsulate data and the operations that must be applied to manipulate the data. Communication and coordination between components are accomplished via message passing.



Layered architectures. The basic structure of a layered architecture is illustrated in Figure . A number of different layers are defined, each accomplishing operations that progressively become closer to the machine instruction set. At the outer layer, components service user interface operations. At the inner layer, components perform operating system interfacing. Intermediate layers provide utility services and application software functions. These architectural styles are only a small subset of those available.

Once requirements engineering uncovers the characteristics and constraints of the system to be built, the architectural style and/or combination of patterns that best fits those characteristics and constraints can be chosen. In many cases, more than one pattern might be

appropriate and alternative architectural styles can be designed and evaluated. For example, a layered style (appropriate for most systems) can be combined with a data-centered architecture in many database applications.

Architectural Patterns

As the requirements model is developed, you'll notice that the software must address a number of broad problems that span the entire application. For example, the requirements model for virtually every e-commerce application is faced with the following problem: How do we offer a broad array of goods to a broad array of customers and allow those customers to purchase our goods online? The requirements model also defines a context in which this question must be answered. For example, an e-commerce business that sells golf equipment to consumers will operate in a different context than an e-commerce business that sells high-priced industrial equipment to medium and large corporations. In addition, a set of limitations and constraints may affect the way in which you address the problem to be solved. Architectural patterns address an application-specific problem within a specific context and under a set of limitations and constraints. The pattern proposes an architectural solution that can serve as the basis for architectural design.

Design Patterns

A design pattern can be characterized as "a three-part rule which expresses a relation between a certain context, a problem, and a solution". For software design, context allows the reader to understand the environment in which the problem resides and what solution might be appropriate within that environment. A set of requirements, including limitations and constraints, acts as a system of forces that Influences how the problem can be interpreted within its context and how the solution can be effectively applied.

An effective design pattern in the following way:

• It solves a problem: Patterns capture solutions, not just abstract principles or strategies.

• It is a proven concept: Patterns capture solutions with a track record, not theories or speculation.

• The solution isn't obvious: Many problem-solving techniques (such as software design paradigms or methods) try to derive solutions from first principles. The best patterns generate a solution to a problem indirectly—a necessary approach for the most difficult problems of design.

• It describes a relationship: Patterns don't just describe modules, but describe deeper system structures and mechanisms.

• The pattern has a significant human component (minimize human intervention).

All software serves human comfort or quality of life; the best patterns explicitly appeal to aesthetics and utility. Stated even more pragmatically, a good design pattern captures hard-earned, pragmatic design knowledge in a way that enables others to reuse that

knowledge “a million times over without ever doing it the same way twice.” A design pattern saves you from “reinventing the wheel,” or worse, inventing a “new wheel” that is slightly out of round, too small for its intended use, and too narrow for the ground it will roll over. Design patterns, if used effectively, will invariably make you a better software designer.

Kinds of Patterns

When software design patterns are considered, we strive to identify and document generative patterns. That is, we identify a pattern that describes an important and repeatable aspect of a system and that provides us with a way to build that aspect within a system of forces that are unique to a given context. In an ideal setting, a collection of generative design patterns could be used to “generate” an application or computer-based system whose architecture enables it to adapt to change. Sometimes called generativity, Design patterns span a broad spectrum of abstraction and application.

- **Architectural patterns** describe broad-based design problems that are solved using a structural approach.
- **Data patterns** describe recurring data-oriented problems and the data modeling solutions that can be used to solve them. Component patterns (also referred to as design patterns) address problems associated with the development of subsystems and components, the manner in which they communicate with one another, and their placement within a larger architecture.
- **Interface design patterns** describe common user interface problems and their solution with a system of forces that includes the specific characteristics of end users.
- **WebApp patterns** address a problem set that is encountered when building WebApps and often incorporates many of the other patterns categories just mentioned. At a lower level of abstraction, idioms describe how to implement all or part of a specific algorithm or data structure for a software component within the context of a specific programming language.
- **Creational patterns** provide mechanisms that make the instantiation of objects easier within a system and enforce “constraints on the type and number of objects that can be created within a system” .
- **Structural patterns** focus on problems and solutions associated with how classes and objects are organized and integrated to build a larger structure. In essence, they help to establish relationships between entities within a system. For example, structural patterns that focus on class-oriented issues might provide inheritance mechanisms that lead to more effective program interfaces. Structural patterns that focus on objects suggest techniques for combining objects within other objects or integrating objects into a larger structure.
- **Behavioral patterns** address problems associated with the assignment of responsibility between objects and the manner in which communication is

effected between objects. A wide variety of design patterns that fit into creational, structural, and behavioral categories have been proposed and can be found on the Web.

Creational Patterns

- Abstract factory pattern: centralize decision of what factory to instantiate.
- Factory method pattern: centralize creation of an object of a specific type choosing one of several implementations.
- Builder pattern: separate the construction of a complex object from its representation so that the same construction process can create different representations.
- Prototype pattern: used when the inherent cost of creating a new object in the standard way (e.g., using the "new" keyword) is prohibitively expensive for a given application.
- Singleton pattern: restrict instantiation of a class to one object.

Structural Patterns

- Adapter pattern: "adapts" one interface for a class into one that a client expects.
- Aggregate pattern: a version of the composite pattern with methods for aggregation of children.
- Bridge pattern: decouple an abstraction from its implementation so that the two can vary independently.
- Composite pattern: a tree structure of objects where every object has the same interface.
- Container pattern: create objects for the sole purpose of holding other objects and managing them.
- Proxy pattern: a class functioning as an interface to another thing.
- Pipes and filters: a chain of processes where the output of each process is the input of the next.

Behavioral Patterns

- Chain of responsibility pattern: Command objects are handled or passed on to other objects by logic-containing processing objects.
- Command pattern: Command objects encapsulate an action and its parameters.
- Event listener: Data are distributed to objects that are registered to receive them.
- Interpreter pattern: Implement a specialized computer language to rapidly solve a specific set of problems.
- Iterator pattern: Iterators are used to access the elements of an aggregate object sequentially without exposing its underlying representation.
- Mediator pattern: Provides a unified interface to a set of interfaces in a subsystem.
- Visitor pattern: A way to separate an algorithm from an object.
- Single-serving visitor pattern: Optimize the implementation of a visitor that is allocated, used only once, and then deleted.

- Hierarchical visitor pattern: Provide a way to visit every node in a hierarchical data structure such as a tree.

Frameworks

Patterns themselves may not be sufficient to develop a complete design. In some cases it may be necessary to provide an implementation-specific skeletal infrastructure, called a framework, for design work. A framework is not an architectural pattern, but rather a skeleton with a collection of "plug points" (also called hooks and slots) that enable it to be adapted to a specific problem domain. The plug points enable you to integrate problem-specific classes or functionality within the skeleton. In an object-oriented context, a framework is a collection of cooperating classes.The differences between design patterns and frameworks in the following manner:

1. Design patterns are more abstract than frameworks. Frameworks can be embodied in code, but only examples of patterns can be embodied in code. A strength of frameworks is that they can be written down in programming languages and not only studied but executed and reused directly.

2. Design patterns are smaller architectural elements than frameworks. A typical framework contains several design patterns but the reverse is never true.

3. Design patterns are less specialized than frameworks. Frameworks always have a particular application domain. In contrast, design patterns can be used in nearly any kind of application. While more specialized design patterns are certainly possible, even these wouldn't dictate an application architecture. In essence, the designer of a framework will argue that one reusable mini architecture is applicable to all software to be developed within a limited domain of application. To be most effective, frameworks are applied with no changes. Additional design elements may be added, but only via the plug points that allow the designer to flesh out the framework skeleton.

Describing a Pattern

Pattern-based design begins with the recognition of patterns within the application you intend to build, continues with a search to determine whether others have addressed the pattern, and concludes with the application of an appropriate pattern to the problem at hand. The second of these three tasks is often the most difficult.

The names of design patterns should be chosen with care. One of the key technical problems in pattern-based design is the inability to find existing patterns when hundreds or thousands of candidate patterns exist. The search for the "right" pattern is aided immeasurably by a meaningful pattern name. A pattern template provides a standardized means for describing a design pattern. Each of the template entries represents characteristics of the design pattern that can be searched (e.g., via a database) so that the appropriate pattern can be found.

Pattern Languages and Repositories

A pattern language encompasses a collection of patterns, each described using a standardized template and interrelated to show how these patterns collaborate to solve problems across an application domain.

In a pattern language, design patterns are organized in a way that provides a "structured method of describing good design practices within a particular domain."

In a way, a pattern language is analogous to a hypertext instruction manual for problem solving in a specific application domain. The problem domain under consideration is first described hierarchically, beginning with broad design problems associated with the domain and then refining each of the broad problems into lower levels of abstraction.

In a software context, broad design problems tend to be architectural in nature and address the overall structure of the application and the data or content that serve it. Architectural problems are refined to lower levels of abstraction, leading to design patterns that solve sub problems and collaborate with one another at the component (or class) level. Rather than a sequential list of patterns, a pattern language represents an interconnected collection in which the user can begin with a broad design problem and "burrow down" to uncover specific problems and their solutions.

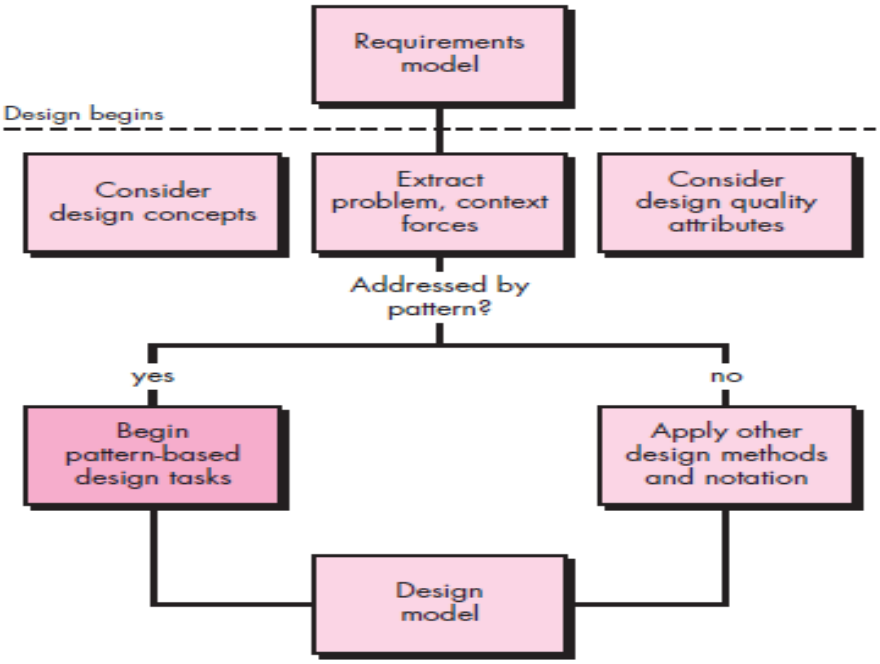
In most cases, the design patterns that are part of pattern language are stored in a Webaccessible patterns repository.The repository provides an index of all design patterns and contains hypermedia links that enable the user to understand the collaborations between patterns.

Pattern-Based Software Design

The best designers in any field have an uncanny ability to see patterns that characterize a problem and corresponding patterns that can be combined to create a solution. The software developers at Microsoft [Mic04] discuss this when they write: While pattern-based design is relatively new in the field of software development, industrial technology has used pattern-based design for decades, perhaps even centuries. Catalogs of mechanisms and standard configurations provide design elements that are used to engineer automobiles, aircraft, machine tools, and robots. Applying pattern based design to software development promises the same benefits to software as it does into industrial technology: predictability, risk mitigation, and increased productivity.Throughout the design process, you should look for every opportunity to apply existing design patterns (when they meet the needs of the design) rather than creating new ones.

Pattern-Based Design in Context

The role of pattern-based design in all of this is Figure. A software designer begins with a requirements model (either explicit or implied) that presents an abstract representation of the system. The requirements model describes the problem set, establishes the context, and identifies the system of forces that hold sway. It may imply the design in an abstract manner, but the requirements model does little to represent the design explicitly.



to keep quality

Thinking in Patterns

1. Be sure you understand the big picture—the context in which the software to

be built resides. The requirements model should communicate this to you.

2. Examining the big picture, extract the patterns that are present at that level of abstraction.

3. Begin your design with “big picture” patterns that establish a context or skeleton for further design work.

4. “Work inward from the context” looking for patterns at lower levels of abstraction that contribute to the design solution.

5. Repeat steps 1 to 4 until the complete design is fleshed out.

6. Refine the design by adapting each pattern to the specifics of the software you’re trying to build.

Design Tasks

The following design tasks are applied when a pattern-based design philosophy isused:

1. Examine the requirements model and develop a problem hierarchy. Describe each problem and subproblem by isolating the problem, the context, and the system of forces that apply. Work from broad problems (high mlevel of abstraction) to smaller subproblems (at lower levels of abstraction).

2. Determine if a reliable pattern language has been developed for the problem domain. problems associated with a specific application domain. The SafeHome software team would look for a pattern language developed specifically for home security products. If that level of pattern language specificity could not be found, the team would partition the SafeHome software problem into a series of generic problem domains (e.g., digital device monitoring problems, user interface problems, digital video management problems) and search for appropriate pattern languages.

3. Beginning with a broad problem, determine whether one or more architectural patterns is available for it. If an architectural pattern is available, be certain to examine all collaborating patterns. If the pattern is appropriate, adapt the design solution proposed and build a design model element that adequately represents it.

4. Using the collaborations provided for the architectural pattern, examine subsystem or component-level problems and search for appropriate patterns to address them. It may be necessary to search through other pattern repositories as well as the list of patterns that corresponds to the architectural solution. If an appropriate pattern is found, adapt the design solution proposed and build a design model element that adequately represents it. Be certain to apply step 7.

5. Repeat steps 2 through 5 until all broad problems have been addressed. The implication is to begin with the big picture and elaborate to solve problems at increasingly more detailed levels.

6. If user interface design problems have been isolated (this is almost always the case), search the many user interface design pattern repositories for appropriate patterns. Proceed in a manner similar to steps 3, 4, and 5.

7. Regardless of its level of abstraction, if a pattern language and/or patterns repository or individual pattern shows promise, compare the problem to be solved against the existing pattern(s) presented. Be certain to examine context and forces to ensure that the pattern does, in fact, provide a solution that is amenable to the problem.

8. Be certain to refine the design as it is derived from patterns using design quality criteria as a guide. Although this design approach is top-down, real-life design solutions are sometimes more complex

Building a Pattern-Organizing Table

As pattern-based design proceeds, you may encounter trouble organizing and categorizing candidate patterns from multiple pattern languages and repositories. To help organize your evaluation of candidate patterns, the creation of a pattern-organizing table that takes the general form

	Database	Application	Implementation	Infrastructure
Data/Content				
Problem statement ...	PatternName(s)		PatternName(s)	
Problem statement ...		PatternName(s)		PatternName(s)
Problem statement ...	PatternName(s)			PatternName(s)
Architecture				
Problem statement ...		PatternName(s)		
Problem statement ...		PatternName(s)		PatternName(s)
Problem statement ...				
Component-level				
Problem statement ...		PatternName(s)	PatternName(s)	
Problem statement ...				PatternName(s)
Problem statement ...		PatternName(s)	PatternName(s)	
User interface				
Problem statement ...		PatternName(s)	PatternName(s)	
Problem statement ...		PatternName(s)	PatternName(s)	
Problem statement ...		PatternName(s)	PatternName(s)	

A pattern-organizing table can be implemented as a spreadsheet model using the form shown in the figure. An abbreviated list of problem statements, organized by data/content, architecture, component-level, and user interface issues, is presented in the left-hand (shaded) column. Four pattern types—database, application, implementation, and infrastructure—are listed across the top row. The names of candidate patterns are noted in the cells of the table.

Common Design Mistakes

Pattern-based design can make you a better software designer, but it is not a panacea. Like all design methods, you must begin with first principles, emphasizing software quality fundamentals and ensuring that the design does, in fact, address the needs expressed by the requirements model. A number of common mistakes occur when pattern-based design is used. In some cases, not enough time has been spent to understand the underlying problem and its context and forces, and as a consequence, you select a pattern that looks right but is inappropriate for the solution required. Once the wrong pattern is selected, you refuse to see your error and force-fit the pattern. In other cases, the problem has forces that are not considered by the pattern you've chosen, resulting in a poor or erroneous fit. Sometimes a pattern is applied too literally and the required adaptations for your problem space are not implemented.

Architectural Patterns

If a house builder decides to construct a center-hall colonial, there is a single architectural style that can be applied. The details of the style (e.g., number of fireplaces, façade of the

house, placement of doors and windows) can vary considerably, but once the decision on the overall architecture of the house is made, the style is imposed on the design. Architectural patterns are a bit different.

Architectural pattern domains.

- Access control.
- Concurrency.
- Distribution.
- Persistence.

Component-Level Design Patterns

Component-level design patterns provide you with proven solutions that address one or more subproblems extracted from the requirements model. In many cases, design patterns of this type focus on some functional element of a system.

User Interface Design Patterns

Hundreds of user interface (UI) patterns have been proposed in recent years. Most fall within one of the following categories of patterns

Whole UI. Provide design guidance for top-level structure and navigation throughout the entire interface.

Page layout. Address the general organization of pages (for websites) or distinct screen displays (for interactive applications).

Tables. Provide design guidance for creating and manipulating tabular data of all kinds.

Direct data manipulation. Address data editing, modification, and transformation. Navigation. Assist the user in navigating through hierarchical menus, Web pages, and interactive display screens.

Searching. Enable content-specific searches through information maintained within a website or contained by persistent data stores that are accessible via an interactive application.

Page elements. Implement specific elements of a Web page or display screen.

E-commerce. Specific to websites, these patterns implement recurring elements of e-commerce applications.

Miscellaneous. Patterns that do not easily fit into one of the preceding categories.

In some cases, these patterns are domain dependent or occur only for specific classes of users.

Webapp Design Patterns

When you consider the design problems that must be solved when a WebApp is to be built, it's worth considering pattern categories by focusing on two dimensions: the design focus of the pattern and its level of granularity.

- Design focus identifies which aspect of the design model is relevant (e.g., information architecture, navigation, interaction).
- Granularity identifies the level of abstraction that is being considered (e.g., does the pattern apply to the entire WebApp, to a single Web page, to a subsystem, or an individual WebApp component?).

Design Focus

WebApp design can be developed for different levels of design focus, so that you can address the unique problems (and related solutions) that are encountered at each level. WebApp patterns can be categorized using the following levels of design focus:

- Information architecture patterns relate to the overall structure of the information space, and the ways in which users will interact with the information.
- **Navigation patterns** define navigation link structures, such as hierarchies, rings, tours, and so on.
- **Interaction patterns** contribute to the design of the user interface. Patterns in this category address how the interface informs the user of the consequences of a specific action, how a user expands content based on usage context and user desires, how to best describe the destination that is implied by a link, how to inform the user about the status of an ongoing interaction, and interface-related issues.
- **Presentation patterns** assist in the presentation of content as it is presented to the user via the interface. Patterns in this category address how to organize user interface control functions for better usability, how to show the relationship between an interface action and the content objects it affects, and how to establish effective content hierarchies.
- **Functional patterns** define the workflows, behaviors, processing, communication, and other algorithmic elements within a WebApp.

In most cases, it would be fruitless to explore the collection of information architecture patterns when a problem in interaction design is encountered. You would examine interaction patterns, because that is the design focus that is relevant to the work being performed.

Design Granularity

When a problem involves "big picture" issues, you should attempt to develop solutions (and use relevant patterns) that focus on the big picture. Conversely, when the focus is very narrow (e.g., uniquely selecting one item from a small set of five or fewer items), the solution (and the corresponding pattern) is targeted quite narrowly.

In terms of the level of granularity, patterns can be described at the following levels:

• **Architectural patterns**. This level of abstraction will typically relate to patterns that define the overall structure of the WebApp, indicate the relationships among different components or increments, and define the rules for specifying relationships among the elements (pages, packages, components, subsystems) of the architecture.

• **Design patterns**. These address a specific element of the design such as an aggregation of components to solve some design problem, relationships among elements on a page, or the mechanisms for effecting component-to component communication. An example might be the Broadsheet pattern for the layout of a WebApp home page.

• **Component patterns.** This level of abstraction relates to individual smallscale elements of a WebApp. Examples include individual interaction elements (e.g., radio buttons), navigation items (e.g., how might you format links?) or functional elements (e.g., specific algorithms). It is also possible to define the relevance of different patterns to different classes of applications or domains. For example, a collection of patterns (at different levels of design focus and granularity) might be particularly relevant to e-business.

User interface design

User interface design creates an effective communication medium between a human and a computer. Following a set of interface design principles, design identifies interface objects

and actions and then creates a screen layout that forms the basis for a user interface prototype.

A software engineer designs the user interface by applying an iterative process that draws on predefined design principles.

Importance of user interface design

If software is difficult to use, if it forces you into mistakes, or if it frustrates your efforts to accomplish your goals, you won't like it, regardless of the computational power it exhibits, the content it delivers, or the functionality it offers.

The interface has to be right because it molds a user's perception of the software.

Steps to be followed are,User interface design begins with the identification of user, task, and environ- mental requirements. Once user tasks have been identified, user scenarios are created and analysed to define a set of interface objects and actions. These form the basis for the creation of screen layout that depicts graphical design and placement of icons, definition of descriptive screen text, specification and titling for windows, and specification of major and minor menu items. Tools are used to prototype and ultimately implement the design model, and the result is evaluated for quality.

What is the work product? User scenarios are created and screen layouts are generated. An interface prototype is developed and modified in an iterative fashion. How do I ensure that I've done it right? An interface prototype is "test driven" by the users, and feedback from the test drive is used for the next iterative modification of the prototype.

Interface Analysis and Design Models

Four different models come into play when a user interface is to be analyzed and designed. A human engineer (or the software engineer) establishes a **user model**,

The software engineer creates a **design model**, the end user develops a mental image that is often called the **user's mental model or the system perception**, and

The implementers of the system create an **implementation model.** Unfortunately, each of these models may differ significantly.

The user's mental model (system perception) is the image of the system that end users carry in their heads.

The implementation model combines the outward manifestation of the computer based system (the look and feel of the interface), coupled with all supporting information(books, manuals, videotapes, help files) that describes interface syntax and semantics.

When the implementation model and the user's mental model are coincident,users generally feel comfortable with the software and use it effectively. To accomplish this "melding" of the models, the design model must have been developed to accommodate the information contained in the user model, and the implementation model must accurately reflect syntactic and semantic information about the interface. In essence, these models enable the interface designer to satisfy a key element of the most important principle of user interface design: "Know the user, know the tasks."

The Process

The analysis and design process for user interfaces is iterative and can be represented using a spiral model. Referring to the below diagram, the user interface analysis and design process begins at the interior of the spiral and encompasses four distinct framework activities (1) interface analysis and modeling,

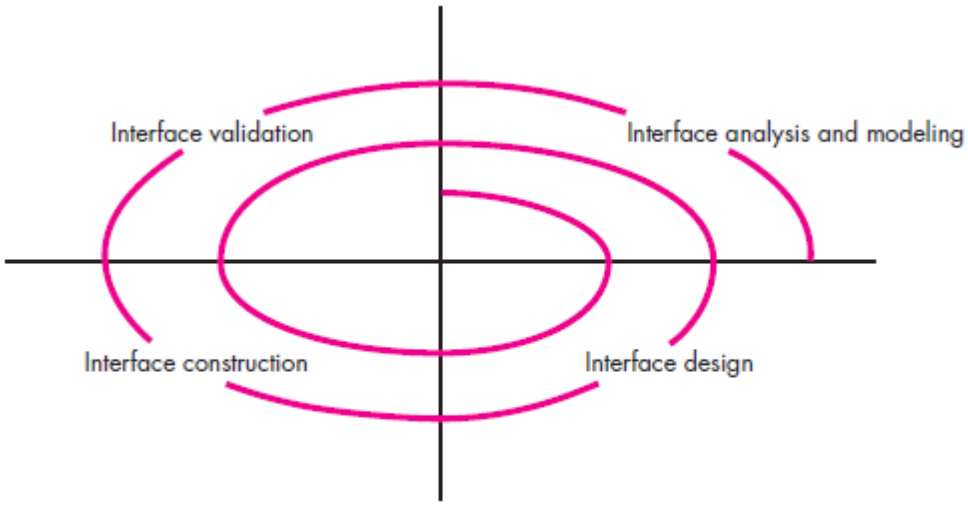
(2) interface design,

(3) interface construction, and

(4) interface validation.

The spiral implies that each of these tasks will occur more than once, with each pass around the spiral representing additional elaboration of requirements and the resultant design. In most cases, the construction activity involves prototyping—the only practical way to validate what has been designed. Interface analysis focuses on the profile of the users who will interact with the system. Skill level, business understanding, and general receptiveness to the new system are recorded; and different user categories are defined. For each user category, requirements are elicited. In essence, you work to understand the system perception for each class of users. Once general requirements have been defined, a more detailed task analysis is conducted. Those tasks that the user performs to accomplish the goals of the

system are identified, described, and elaborated (over a number of iterative passes through the spiral).



Finally, analysis of the user environment focuses on the physical work environment.

Interface Analysis

A key tenet of all software engineering process models is this: understand the problem before you attempt to design a solution. In the case of user interface design, understanding the problem means understanding

(1) the people (end users) who will interact with the system through the interface,

(2) the tasks that end users must perform to do their work,

(3) the content that is presented as part of the interface, and

(4) the environment in which these tasks will be conducted.

User Analysis

The phrase “user interface” is probably all the justification needed to spend some time understanding the user before worrying about technical matters. Earlier I noted that each user has a mental image of the software that may be different from the mental image developed by other users. In addition, the user’s mental image may be vastly different from the software engineer’s design model. The only way that you can get the mental image and the design model to converge is to work to understand the users themselves as well as how these people will use the system. Information from a broad array of sources can be used to accomplish this:

User Interviews. The most direct approach, members of the software team meet with end users to better understand their needs, motivations, work culture and a myriad of other issues. This can be accomplished in one-on-one meetings or through focus groups. Sales input. Sales

people meet with users on a regular basis and can gather information that will help the software team to categorize users and better understand their requirements.

- Marketing input
- Support input.

Task Analysis and Modeling

The goal of task analysis is to answer the following questions:

• What work will the user perform in specific circumstances?

• What tasks and subtasks will be performed as the user does the work?

• What specific problem domain objects will the user manipulate as work is performed?

• What is the sequence of work tasks—the workflow?

• What is the hierarchy of tasks?

Use cases.

This use case provides a basic description of one important work task for the computer-aided design system. From it, you can extract tasks, objects, and the overall flow of the interaction.

Object elaboration. Rather than focusing on the tasks that a user must perform, you can examine the use case and other information obtained from the user and extract the physical objects that are used by the interior designer. These objects can be categorized into classes. Attributes of each class are defined, and an evaluation of the actions applied to each object provide a list of operations.

Workflow analysis. When a number of different users, each playing different roles, makes use of a user interface, it is sometimes necessary to go beyond task analysis and object elaboration and apply workflow analysis. This technique allows you to understand how a work process is completed when several people (and roles) are involved.

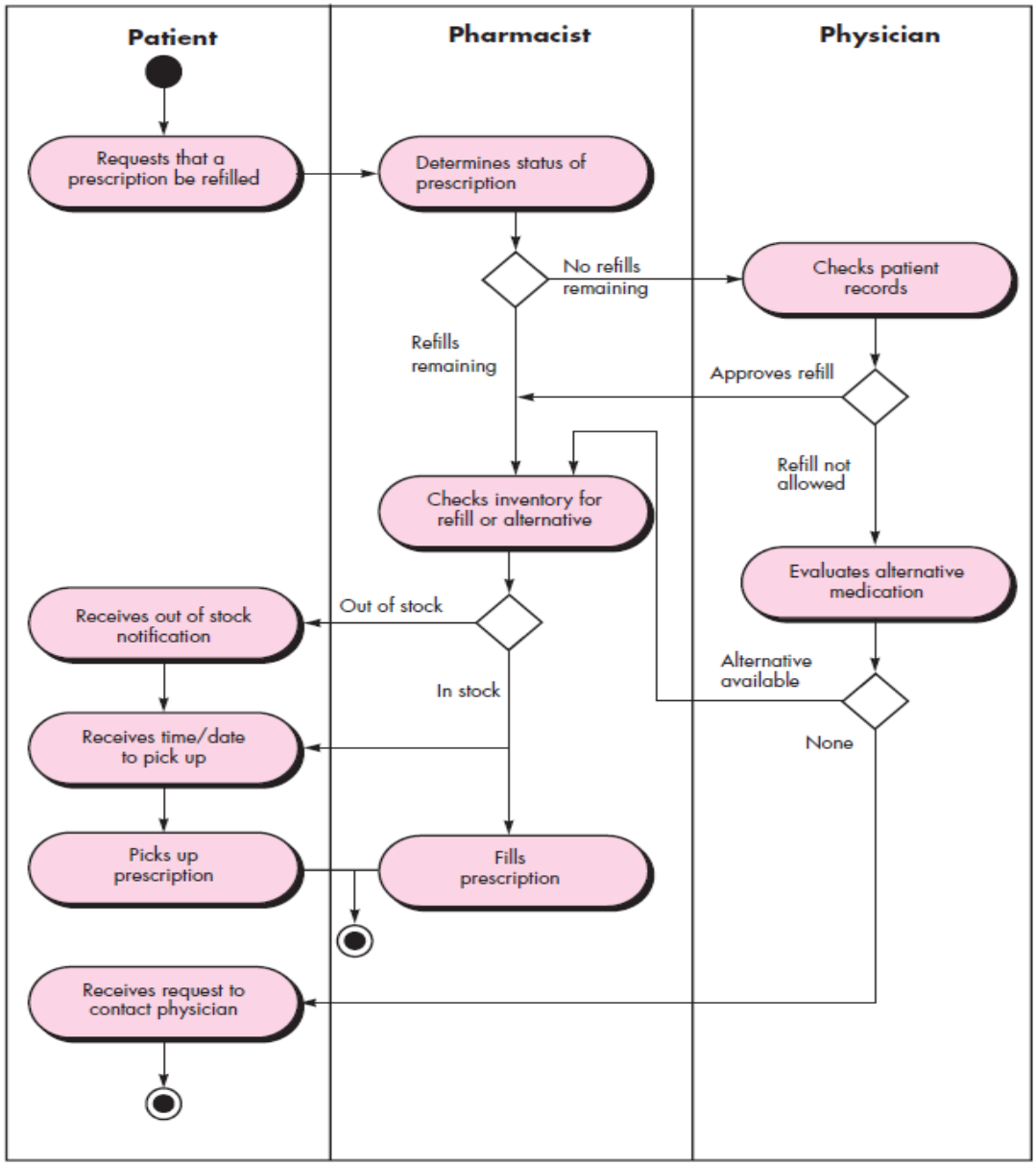
This information may have been elicited via interview or from use cases written by each actor. Regardless, the flow of events (shown in the figure) enables you to recognize a number of key interface characteristics:

1. Each user implements different tasks via the interface; therefore, the look and feel of the interface designed for the patient will be different than the one defined for pharmacists or physicians.

2. The interface design for pharmacists and physicians must accommodate access to and display of information from secondary information sources.

3. Many of the activities can be further elaborated using task analysis and/or object elaboration (e.g., Fills prescription could imply a mail-order delivery, a visit to a pharmacy, or a visit to a special drug distribution center).

\



Hierarchical representation. A process of elaboration occurs as you begin to analyze the interface. Once workflow has been established, a task hierarchy can be defined for each user type. The hierarchy is derived by a stepwise elaboration of User task: Requests that a prescription be refilled

• Provide identifying information.

• Specify name.

• Specify userid.

• Specify PIN and password.

• Specify prescription number.

• Specify date refill is required.

To complete the task, three subtasks are defined. One of these subtasks, provide identifying information, is further elaborated in three additional sub-subtasks.

Analysis of Display Content

The user tasks lead to the presentation of a variety of different types of content. For modern applications, display content can range from character-based reports (e.g., a spreadsheet), graphical displays (e.g., a histogram, a 3-D model, a picture of a person), or specialized information (e.g., audio or video files).These data objects may be

(1) generated by components (unrelated to the interface) in other parts of an application,

(2) acquired from data stored in a database that is accessible from the application, or

(3) transmitted from systems external to the application in question.

APEC