UNIT III

Design Engineering:

Design Engineering is a critical phase in the software development process that focuses on **how the system will be built**. It bridges the gap between **requirements specification** and **coding**, by defining the software architecture, data structures, interfaces, and algorithms. Design is a **multi-step process** in which representations of data and program structure, interface characteristics, and procedural detail are developed to guide coding."

Goals of Design Engineering:

- Transform the **requirements model** into a **design model**
- Ensure that the software meets **functional and quality requirements**
- Establish a basis for **software construction (coding)** and **testing**

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 behavioural 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.

1. Software Quality Guidelines and Attributes:

Throughout the design process, the quality of the evolving design is assessed with a series of technical reviews. McGlaughlin [McG91] suggests three characteristics that serve as a guide for the evaluation of a good design:

- The design should 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 should 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, you and other members of the software team must establish technical criteria for good design. Design concepts that also serve as software quality criteria. For the time being, consider the following guidelines:

1. A design should exhibit an architecture that,

(a) Has been created using recognizable architectural styles or patterns,

(b) Is composed of components that exhibit good design characteristics (these are discussed later in this chapter), and

(c) 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 [Gra87] 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 using processing speed, response time, resource consumption, throughput, and efficiency.
- Supportability combines extensibility, adaptability, and serviceability. These three attributes represent a more common term, maintainability and in addition, testability, compatibility, configurability (the ability to organize and control elements of the software configuration), 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.

2. The Evolution of Software Design:

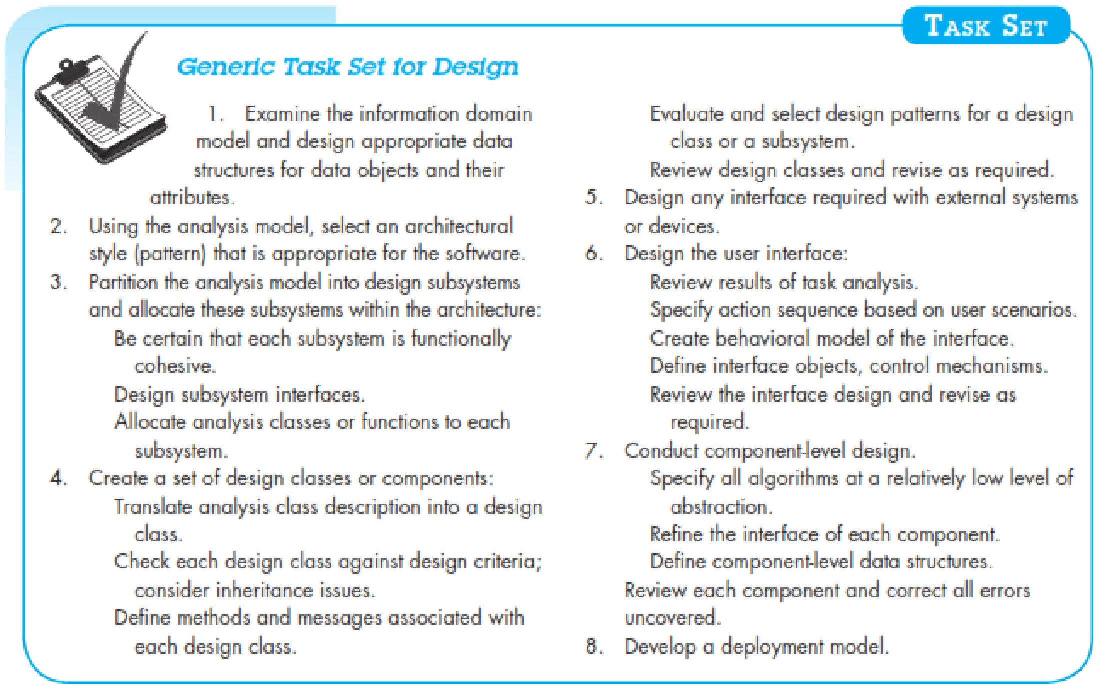
The evolution of software design is a continuing process that has now spanned more than six decades. Early design work concentrated on criteria for the development of modular programs and methods for refining software structures in a top-down “structured” manner. Newer design approaches proposed an object-oriented approach to design derivation. More recent emphasis in software design has been on soft- ware 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. Like the analysis methods, each software design method introduces unique heuristics and notation, as well as a somewhat parochial view of what characterizes de- sign quality. Yet, all of these 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.



TASK SET

Generic Task Set for Design

1. Examine the information domain model and design appropriate data structures for data objects and their attributes.
2. Using the analysis model, select an architectural style (pattern) that is appropriate for the software.
3. Partition the analysis model into design subsystems and allocate these subsystems within the architecture:
   Be certain that each subsystem is functionally cohesive.
   Design subsystem interfaces.
   Allocate analysis classes or functions to each subsystem.
4. Create a set of design classes or components:
   Translate analysis class description into a design class.
   Check each design class against design criteria; consider inheritance issues.
   Define methods and messages associated with each design class.
   Evaluate and select design patterns for a design class or a subsystem.
   Review design classes and revise as required.
5. Design any interface required with external systems or devices.
6. Design the user interface:
   Review results of task analysis.
   Specify action sequence based on user scenarios.
   Create behavioral model of the interface.
   Define interface objects, control mechanisms.
   Review the interface design and revise as required.
7. Conduct component-level design.
   Specify all algorithms at a relatively low level of abstraction.
   Refine the interface of each component.
   Define component-level data structures.
   Review each component and correct all errors uncovered.
8. Develop a deployment model.

Design concepts:

A set of fundamental software design concepts has evolved over the history of software engineering. Although the degree of interest in these concepts has var- ied 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 define criteria that can be used to partition software into individual components, separate or data structure detail from a conceptual representation of the software, and establish uniform criteria that define the technical quality of a software design.

M. A. Jackson [Jac75] once said: “The beginning of wisdom for a [software engineer] is to recognize the difference between getting a program to work, and getting it right.” In the sections that follow, we present an overview of fundamental software design concepts that provide the necessary framework for "getting it right."

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" [Sha95a]. 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 reuse design-level concepts.

Shaw and Garlan [Sha95a] describe a set of properties that should be specified as part of an architectural design. Structural properties define “the components of a system (e.g., modules, objects, filters) and the manner in which those components are packaged and interact with one another.” Extra-functional properties address “how the design architecture achieves requirements for performance, capacity, reliability, security, adaptability, and other system characteristics. Families of related systems “draw upon repeatable patterns that are commonly en- countered in the design of families of similar systems.”

Given the specification of these properties, the architectural design can be represented using one or more of a number of different models [Gar95]. 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 (patterns) that are encountered in similar types of applications. Dynamic models address the behavioural 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 [Sha95b]. 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.

You should note that there is some debate about the role of architecture in design. Some researchers argue that the derivation of software architecture should be separated from design and occurs between requirements engineering actions and more conventional design actions. Others believe that the derivation of architecture is an integral part of the design process.

Patterns:

Brad Appleton defines a design pattern in the following manner: “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 data:

Separation of concerns is a design concept [Dij82] 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.

It 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.

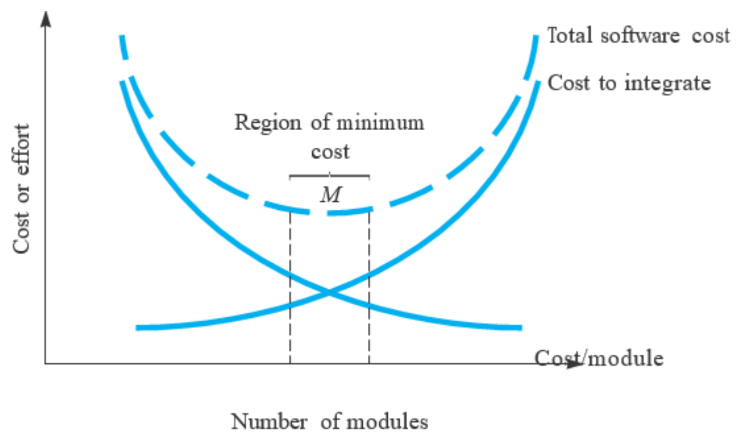
Modularity:

Modularity is the most common manifestation of separation of concerns. Soft- ware 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 al- lows a program to be intellectually manageable" [Mye78]. 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. In almost all instances, you should break the design into many modules, hoping to make understanding easier and, as a consequence, reduce the cost required to build the software.

Recalling our discussion of separation of concerns, it is possible to conclude that if you subdivide software indefinitely the effort required to develop it will become negligibly small! Unfortunately, other forces come into play, causing this conclusion to be (sadly) invalid. Referring to Figure, 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.

Modularity
and software
cost



The curves shown in above 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. But how do you know the vicinity of M? How modular should you make software? The answers to these questions require an understanding of other design concepts. 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 con- ducted more efficiently, and long-term maintenance can be conducted without serious side effects.

Data 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 [Par72] 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 [Ros75].

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. In landmark papers on software design Wirth [Wir71] and Parnas [Par72] allude to refinement techniques that enhance module independence. Later work by Stevens, Myers, and Constantine [Ste74] solidified the concept.

Functional independence is achieved by developing modules with “single- minded” 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 inter- action 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" [Ste74], caused when errors occur at one location and propagate throughout a system

Refactoring:

An important design activity suggested for many agile methods (Chapter 5), re- factoring is a reorganization technique that simplifies the design (or code) of a component without changing its function or behavior. Fowler [Fow00] defines re- factoring in the following manner: "Refactoring is the process of changing a soft- ware system in such a way that it does not alter the external behavior of the code [design] yet improves its internal structure."

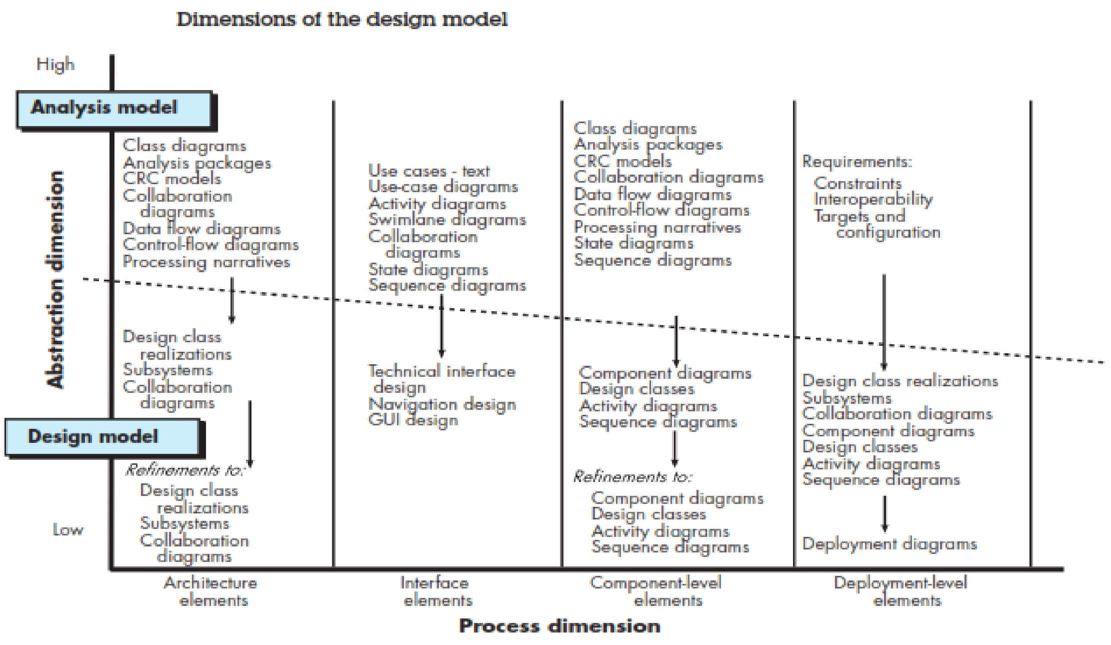
When software is refactored, the existing design is examined for redundancy, unused design elements, inefficient or unnecessary algorithms, poorly constructed or inappropriate data structures, or any other design failure that can be corrected to yield a better design. For example, a first design iteration might yield a component that exhibits low cohesion (i.e., it performs three functions that have only limited relationship to one another). After careful consideration, you may decide that the component should be refactored into three separate components, each exhibiting high cohesion. The result will be software that is easier to integrate, easier to test, and easier to maintain.

Although the intent of refactoring is to modify the code in a manner that does not alter its external behavior, inadvertent side effects can and do occur. As a consequence, refactoring tools [Soa10] are used to analyze changes automatically and to "generate a test suite suitable for detecting behavioral changes."

The Design model:

The design model can be viewed in two different dimensions as illustrated in the below Figure. The process dimension indicates the evolution of the design model as design tasks are executed as part of the software process. The abstraction dimension represents the level of detail as each element of the analysis model is transformed into a design equivalent and then refined iteratively. Referring to the figure, the dashed line indicates the boundary between the analysis and de- sign models. In some cases, a clear distinction between the analysis and design models is possible. In other cases, the analysis model slowly blends into the de- sign and a clear distinction is less obvious.

The elements of the design model use many of the same UML diagrams that were used in the analysis model. The difference is that these diagrams are re- fined and elaborated as part of design; more implementation-specific detail is provided, and architectural structure and style, components that reside within the architecture, and interfaces between the components and with the outside world are all emphasized.



1. Data design elements :

Like other software engineering activities, data design (sometimes referred to as data architecting) creates a model of data and/or information that is represented at a high level of abstraction (the customer/user's view of data). This data model is then refined into progressively more implementation-specific representations that can be processed by the computer-based system. In many soft- ware applications, the architecture of the data will have a profound influence on the architecture of the software that must process it.

The structure of data has always been an important part of software design. At the program-component level, the design of data structures and the associated algorithms required to manipulate them is essential to the creation of high-quality applications. At the application level, the translation of a data model (derived as part of requirements engineering) into a database is pivotal to achieving the business objectives of a system. At the business level, the collection of information stored in disparate databases and reorganized into a "data ware- house" enables data mining or knowledge discovery that can have an impact on the success of the business itself. In every case, data design plays an important role.

2. *Architectural design elements:*

The architectural design for software is the equivalent to the floor plan of a house. The floor plan depicts the overall layout of the rooms; their size, shape, and relationship to one another; and the doors and windows that allow movement into and out of the rooms. The floor plan gives us an overall view of the house. Architectural design elements give us an overall view of the software.

The architectural model is derived from three sources:

(1) Information about the application domain for the software to be built;

(2) Specific requirements model elements such as use cases or analysis classes, their relationships and collaborations for the problem at hand; and

(3) The availability of architectural styles and patterns.

The architectural design element is usually depicted as a set of interconnected subsystems, often derived from analysis packages within the requirements model. Each subsystem may have its own architecture (e.g., a graphical user interface might be structured according to a pre-existing architectural style for user interfaces).

3. *Interface design elements*

The interface design for software is analogous to a set of detailed drawings (and specifications) for the doors, windows, and external utilities of a house. In essence, the detailed drawings (and specifications) for the doors, windows, and external utilities tell us how things and information flow into and out of the house and within the rooms that are part of the floor plan. The interface design elements for software depict information flows into and out of a system and how it is communicated among the components defined as part of the architecture.

There are three important elements of interface design:

(1) The user inter- face (UI),

(2) External interfaces to other systems, devices, networks, or other producers or consumers of information, and

(3) Internal interfaces between various design components. These interface design elements allow the software to communicate externally and enable internal communication and collaboration among the components that populate the software architecture.

UI design (increasingly called usability design) is a major software engineering action and is considered in detail in Chapter 15. Usability design incorporates aesthetic elements (e.g., layout, color, graphics, interaction mechanisms), ergonomic elements (e.g., information layout and placement, metaphors, UI navigation), and technical elements (e.g., UI patterns, reusable components). In general, the UI is a unique subsystem within the overall application architecture.

The design of external interfaces requires definitive information about the entity to which information is sent or received. In every case, this information should be collected during requirements engineering and verified once the interface design commences. the design of external interfaces should incorporate error checking and appropriate security features.

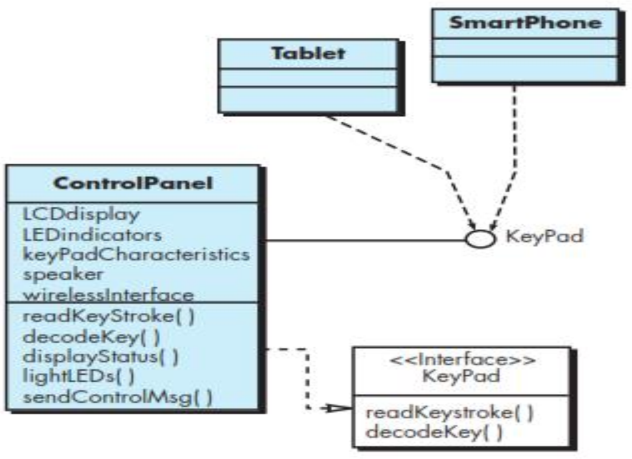
The design of internal interfaces is closely aligned with component-level design. Design realizations of analysis classes represent all operations and the messaging schemes required to enable communication and collaboration between operations in various classes. Each message must be designed to accommodate the requisite information transfer and the specific functional requirements of the operation that has been requested.

In some cases, an interface is modeled in much the same way as a class. In UML, an interface is defined in the following manner: “An interface is a specifier for the externally-visible [public] operations of a class, component, or other classifier (including subsystems) without specification of internal structure.” Stated more simply, an interface is a set of operations that describes some part of the behavior of a class and provides access to these operations.

For example, the **SafeHome** security function makes use of a control panel that allows a homeowner to control certain aspects of the security function. In an advanced version of the system, control panel functions may be implemented via a mobile platform (e.g., smartphone or tablet).

The Control Panel class (Below Figure) provides the behavior associated with a keypad, and therefore, it must implement the operations readKeyStroke () and decodeKey (). If these operations are to be provided to other classes (in this case, Tablet and SmartPhone), it is useful to define an interface as shown in the figure. The interface, named KeyPad, is shown as an <<interface>> stereotype or as a small, labeled circle connected to the class with a line. The interface is defined with no attributes and the set of operations that are necessary to achieve the behavior of a keypad.

Interface representation for ControlPanel



The dashed line with an open triangle at its end (See the above Figure) indicates that the Control Panel class provides KeyPad operations as part of its behavior. In UML, this is characterized as a realization. That is, part of the behavior of Control Panel will be implemented by realizing KeyPad operations. These operations will be provided to other classes that access the interface.

4. *Component level diagram elements:*

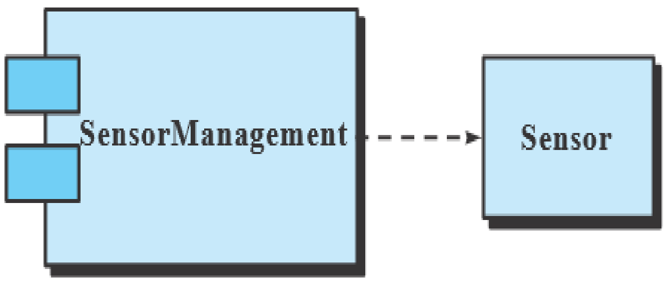
The component-level design for software is the equivalent to a set of detailed drawings (and specifications) for each room in a house. These drawings depict wiring and plumbing within each room, the location of electrical receptacles and wall switches, faucets, sinks, showers, tubs, drains, cabinets, and closets, and every other detail associated with a room.

The component-level design for software fully describes the internal detail of each software component. To accomplish this, the component-level design de- fines data structures for all local data objects and algorithmic detail for all processing that occurs within a component and an interface that allows access to all component operations (behaviors).

Within the context of object-oriented software engineering, a component is represented in UML diagrammatic form as shown in the **below Figure**. In this figure, a component named **SensorManagement** (part of the SafeHome security function) is represented. A dashed arrow connects the component to a class named Sensor that is assigned to it. The **SensorManagement** component performs all functions associated with SafeHome sensors including monitoring and configuring them.

The design details of a component can be modeled at many different levels of abstraction. A UML activity diagram can be used to represent processing logic. Detailed procedural flow for a component can be represented using either pseudocode (a programming language like representation) or some other diagrammatic form (e.g., flowchart or box diagram). Algorithmic structure follows the rules established for structured programming (i.e., a set of constrained procedural constructs). Data structures, selected based on the nature of the data objects to be processed, are usually modeled using pseudocode or the programming language to be used for implementation.

A UML compo-
nent diagram



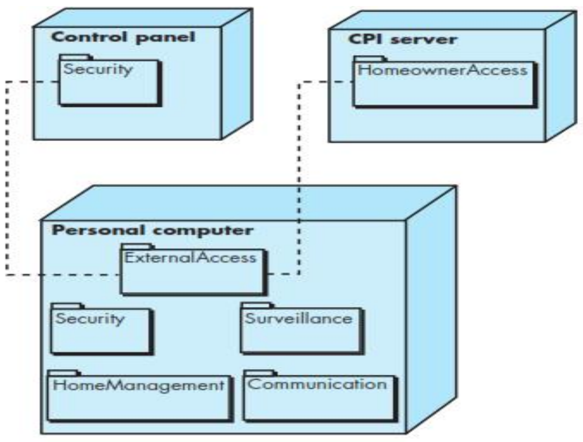
5.Deployment level design elements:

Deployment-level design elements indicate how software functionality and sub-systems will be allocated within the physical computing environment that will support the software. For example, the elements of the SafeHome product are configured to operate within three primary computing environments—a home- based PC, the SafeHome control panel, and a server housed at CPI Corp. (providing Internet-based access to the system). In addition, limited functionality may be provided with mobile platforms.

During design, a UML deployment diagram is developed and then refined as shown in the below Figure. In the figure, three computing environments are shown (in actuality, there would be more including sensors, cameras, and functionality delivered by mobile platforms). The subsystems (functionality) housed within each computing element are indicated. For example, the personal computer houses subsystems that implement security, surveillance, home management, and communications features. In addition, an external access subsystem has been de- signed to manage all attempts to access the SafeHome system from an external source. Each subsystem would be elaborated to indicate the components that it implements.

The diagram shown in below Figure is in descriptor form. This means that the deployment diagram shows the computing environment but does not explicitly indicate configuration details. For example, the "personal computer" is not further identified.

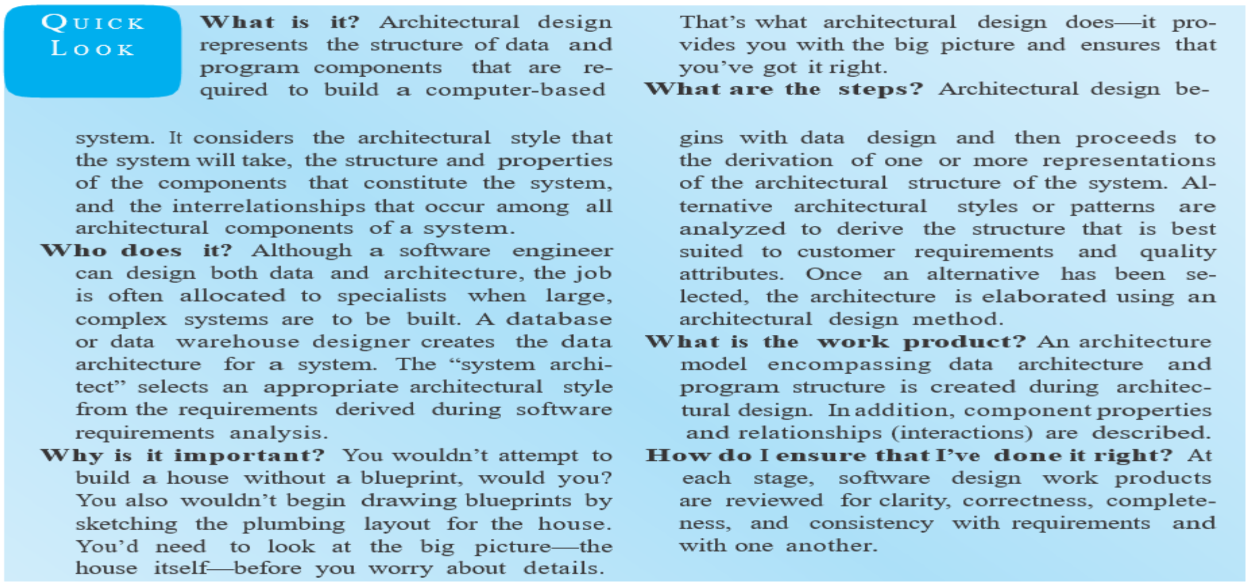
A UML deploy-
ment diagram



It could be a Mac, a Windows-based PC, a Linux-box or a mobile plat- form with its associated operating system. These details are provided when the deployment diagram is revisited in instance form during the latter stages of design or as construction begins. Each instance of the deployment (a specific, named hardware configuration) is identified.

Creating an architectural design:

Design has been described as a multistep process in which representations of data and program structure, interface characteristics, and procedural detail are synthesized from information requirements.


QUICK LOOK

What is it? Architectural design represents the structure of data and program components that are required to build a computer-based system. It considers the architectural style that the system will take, the structure and properties of the components that constitute the system, and the interrelationships that occur among all architectural components of a system.

Who does it? Although a software engineer can design both data and architecture, the job is often allocated to specialists when large, complex systems are to be built. A database or data warehouse designer creates the data architecture for a system. The "system architect" selects an appropriate architectural style from the requirements derived during software requirements analysis.

Why is it important? You wouldn't attempt to build a house without a blueprint, would you? You also wouldn't begin drawing blueprints by sketching the plumbing layout for the house. You'd need to look at the big picture—the house itself—before you worry about details. That's what architectural design does—it provides you with the big picture and ensures that you've got it right.

What are the steps? Architectural design begins with data design and then proceeds to the derivation of one or more representations of the architectural structure of the system. Alternative architectural styles or patterns are analyzed to derive the structure that is best suited to customer requirements and quality attributes. Once an alternative has been selected, the architecture is elaborated using an architectural design method.

What is the work product? An architecture model encompassing data architecture and program structure is created during architectural design. In addition, component properties and relationships (interactions) are described.

How do I ensure that I've done it right? At each stage, software design work products are reviewed for clarity, correctness, completeness, and consistency with requirements and with one another.

Software architecture in the following manner:

Ever since the first program was divided into modules, software systems have had architectures, and programmers have been responsible for the interactions among the modules and the global properties of the assemblage. Historically, architectures have been implicit—accidents of implementation, or legacy systems of the past. Good software developers have often adopted one or several architectural patterns as strategies for system organization, but they use these patterns informally and have no means to make them explicit in the resulting system.

Today, effective software architecture and its explicit representation and design have become dominant themes in software engineering.

What Is Architecture?

When you consider the architecture of a building, many different attributes come to mind. At the most simplistic level, you think about the overall shape of the physical structure. But in reality, architecture is much more. It is the manner in which the various components of the building are integrated to form a cohesive whole. It is the way in which the building fits into its environment and meshes with other buildings in its vicinity. It is the degree to which the building meets its stated purpose and satisfies the needs of its owner. It is the aesthetic feel of the structure—the visual impact of the building—and the way textures, colors, and materials are combined to create the external facade and the internal "living environment." It is small details—the design of lighting fixtures, the type of flooring, the placement of wall hangings, and the list is almost endless. And finally, it is art.

Architecture is also something else. It is "thousands of decisions, both big and small" [Tyr05]. Some of these decisions are made early in design and can have a profound impact on all other design actions. Others are delayed until later, thereby eliminating overly restrictive constraints that would lead to a poor implementation of the architectural style.

But what about software architecture? Bass, Clements, and Kazman [Bas03] define this elusive term in the following way:

The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them.

The architecture is not the operational software. Rather, it is a representation that enables you to

(1) Analyse the effectiveness of the design in meeting its stated requirements,

(2) Consider architectural alternatives at a stage when making design changes is still relatively easy, and

(3) Reduce the risks associated with the construction of the software.

This definition emphasizes the role of "software components" in any architectural representation. In the context of architectural design, a software component can be something as simple as a program module or an object-oriented class, but it can also be extended to include databases and "middleware" that enable the configuration of a network of clients and servers. The properties of components are those characteristics that are necessary to an understanding of how the components interact with other components. At the architectural level, internal properties (e.g., details of an algorithm) are not specified. The relation- ships between components can be as simple as a procedure call from one module to another or as complex as a database access protocol.

Some members of the software engineering community (e.g., [Kaz03]) make a distinction between the actions associated with the derivation of a software architecture (what we call "architectural design") and the actions that are applied to derive the software design.

As one reviewer of a past edition noted:

There is a distinct difference between the terms architecture and design. A design is an instance of an architecture similar to an object being an instance of a class.

For example, consider the client-server architecture. I can design a network-centric software system in many different ways from this architecture using either the Java platform (Java EE) or Microsoft platform (.NET framework). So, there is one architecture, but many designs can be created based on that architecture. Therefore, you cannot mix "architecture" and "design" with each other.

Although we agree that a software design is an instance of a specific software architecture, the elements and structures that are defined as part of an architecture are the root of every design. Design begins with a consideration of architecture.

Why Is Architecture Important?

Three key reasons that software architecture is important:

- Software architecture provides a representation that facilitates communication among all stakeholders.
- The architecture highlights early design decisions that will have a pro- found impact on all software engineering work that follows.
- Architecture "constitutes a relatively small, intellectually graspable model of how the system is structured and how its components work together".

The architectural design model and the architectural patterns contained within it are transferable. That is, architecture genres, styles, and patterns can be applied to the design of other systems and represent a set of abstractions that enable software engineers to describe architecture in predictable ways.

Architectural Descriptions:

Each of us has a mental image of what the word architecture means. The implication is that different stakeholders will see an architecture from different viewpoints that are driven by different sets of concerns. This implies that an architectural description is actually a set of work products that reflect different views of the system.

Smolander, Rossi, and Purao [Smo08] have identified multiple metaphors, representing different views of the same architecture, that stakeholders use to understand the term software architecture. The blueprint metaphor seems to be most familiar to the stakeholders who write programs to implement a system. Developers regard architecture descriptions as a means of transferring explicit information from architects to designers to software engineers charged with producing the system components. The language metaphor views architecture as a facilitator of communication across stakeholder groups. This view is preferred by stakeholders with a high customer focus (e.g., managers or marketing experts). The architectural description needs to be concise and easy to under- stand since it forms the basis for negotiation particularly in determining system boundaries.

The decision metaphor represents architecture as the product of decisions involving trade-offs among properties such as cost, usability, maintainability, and performance. Each of these properties can have a significant impact on the system design. Stakeholders (e.g., project managers) view architectural decisions as the basis for allocating project resources and work tasks. These decisions may affect the sequence of tasks and the structure of the software team. The literature metaphor is used to document architectural solutions constructed in the past. This view supports the construction of artifacts and the transfer of knowledge between designers and software maintenance staff. It also supports stakeholders whose concern is reuse of components and designs.

An architectural description of a software-based system must exhibit characteristics that combine these metaphors. Tyree and Akerman [Tyr05] note this when they write:

Developers want clear, decisive guidance on how to proceed with design. Customers want a clear understanding of the environmental changes that must occur and assurances that the architecture will meet their business needs. Other architects want a clear, salient understanding of the architecture’s key aspects.

Each of these “wants” is reflected in a different metaphor represented using a different viewpoint.

The IEEE Computer Society has proposed IEEE-Std-1471-2000, Recommended Practice for Architectural Description of Software-Intensive Systems, [IEE00], with the following objectives:

(1) To establish a conceptual framework and vocabulary for use during the design of software architecture,

(2) To provide detailed guide- lines for representing an architectural description, and

(3) To encourage sound architectural design practices. An architectural description (AD) represents multiple views, where each view is "a representation of a whole system from the perspective of a related set of [stakeholder] concerns."

Architectural Decisions:

Each view developed as part of an architectural description addresses a specific stakeholder concern. To develop each view (and the architectural description as a whole) the system architect considers a variety of alternatives and ultimately decides on the specific architectural features that best meet the concern. Therefore, architectural decisions themselves can be considered to be one view of the architecture. The reasons that decisions were made provide insight into the structure of a system and its conformance to stakeholder concerns.

As a system architect, you can use the template suggested in the sidebar to document each major decision. By doing this, you provide a rationale for your work and establish a historical record that can be useful when design modifications must be made.

Grady Booch [Boo11a] writes that when setting out to build an innovative product, software engineers often feel compelled to plunge right in, build stuff, fix what doesn't work, improve what does work, and then repeat the process. After doing this a few times, they begin to recognize that an architecture should be defined and decisions associated with architectural choices must be stated explicitly. It may not be possible to predict the right choices before building a new product. However, if innovators find that architectural decisions are worth repeating after testing their prototypes in the field, then a dominant design for this type of product may begin to emerge. Without documenting what worked and what did not, it is hard for software engineers to decide when to innovate and when to use previously created architecture.

Architectural styles and patterns.

When a builder uses the phrase “center hall colonial” to describe a house, most people familiar with houses in the United States will be able to conjure a general image of what the house will look like and what the floor plan is likely to be. The builder has used an architectural style as a descriptive mechanism to differentiate the house from other styles (e.g., A-frame, raised ranch, Cape Cod). But more important, the architectural style is also a template for construction. Further details of the house must be defined, its final dimensions must be specified, customized features may be added, building materials are to be determined, but the style—a “center hall colonial”—guides the builder in his work.

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),

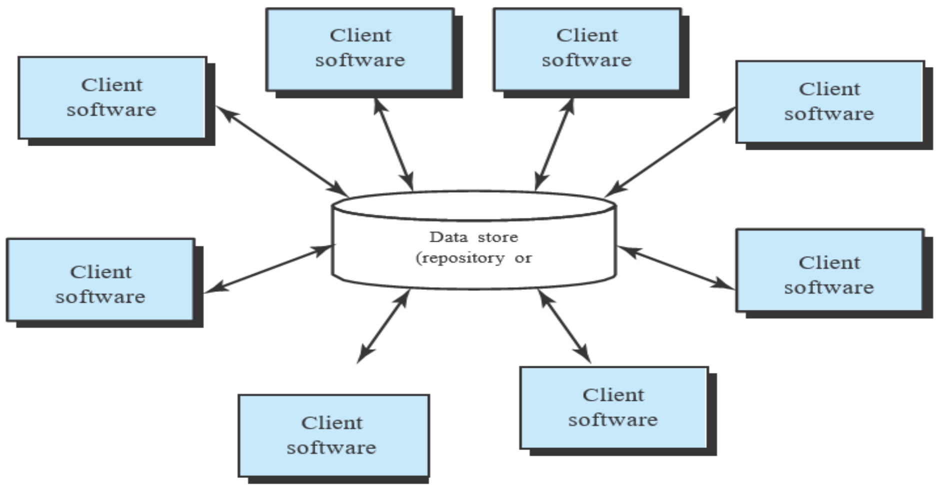
(3) Architectural patterns tend to address specific behavioral issues within the context of the architecture (e.g., how real-time applications handle synchronization or interrupts). Patterns

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,



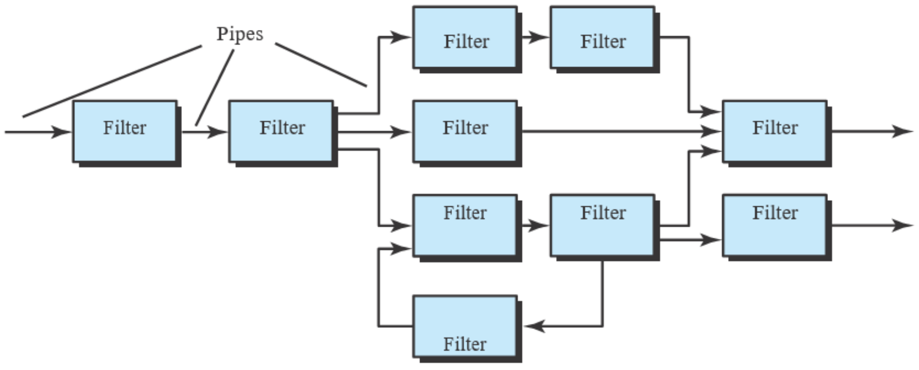
delete, or otherwise modify data within the store. Above Figure. Illustrates a typical data-centered style. Client software accesses a central repository. In some cases the data repository is passive. That is, client software accesses the data independent of any changes to the data or the actions of 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.

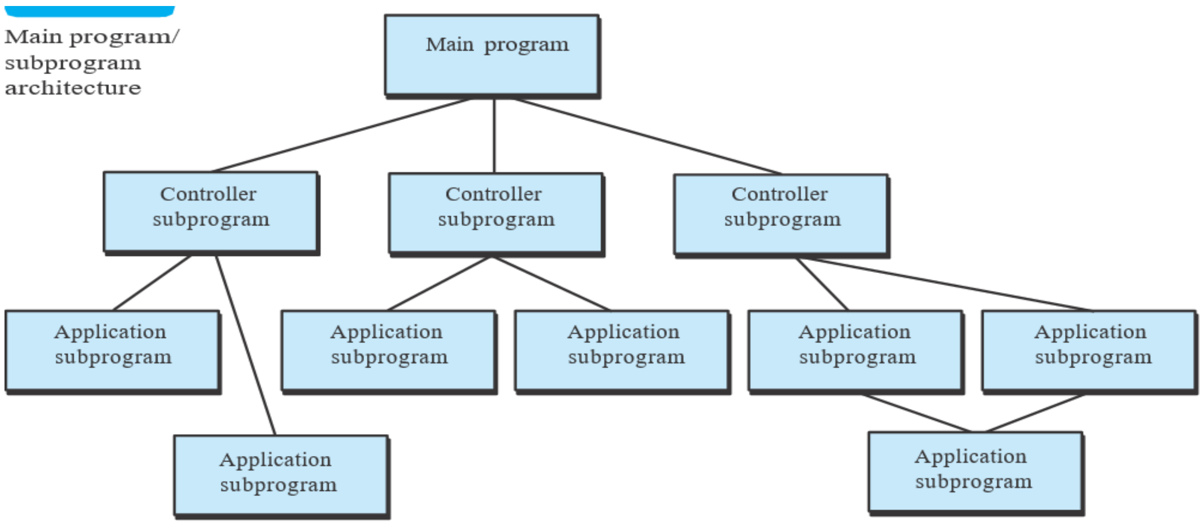
A pipe-and-filter pattern (Below Figure) has a set of components, called filters, connected by pipes that transmit data from one component to the next. Each filter works independently of those components upstream and down- stream, is designed to expect data input of a certain form, and produces data 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

Fig: Data Flow Architecture



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 [Bas03] 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, which in turn may invoke still other components. Above 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.

Architectural Patterns

As the requirements model is developed, you'll notice that the software must ad- dress 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 many different 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 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.

Previously in this chapter, we noted that most applications fit within a specific domain or genre and that one or more architectural styles may be appropriate for that genre. For example, the overall architectural style for an application might be call-and-return or object-oriented. But within that style, you will encounter a set of common problems that might best be addressed with specific architectural patterns.

Architectural Design:

As architectural design begins, context must be established. To accomplish this, the external entities (e.g., other systems, devices, and people) that interact with the software and the nature of their interaction are described. This information can generally be acquired from the requirements model. Once context is modeled and all external software interfaces have been described, you can identify a set of architectural archetypes.

An archetype is an abstraction (similar to a class) that represents one element of system behavior. The set of archetypes provides a collection of abstractions that must be modeled architecturally if the system is to be constructed, but the archetypes themselves do not provide enough implementation detail. Therefore, the designer specifies the structure of the system by defining and refining soft- ware components that implement each archetype. This process continues iteratively until a complete architectural structure has been derived.

A number of questions must be asked and answered as a software engineer creates meaningful architectural diagrams. Does the diagram show how the system responds to inputs or events? What visualizations might there be to help emphasize areas of risk? How can hidden system design patterns be made more obvious to other developers? Can multiple viewpoints show the best way to refactor specific parts of the system? Can design trade-offs be represented in a meaningful way? If a diagrammatic representation of software architecture answers these questions, it will have value to software engineers that use it.

1. Representing the system in context.

At the architectural design level, a software architect uses an architectural con- text diagram (ACD) to model the manner in which software interacts with enti- ties external to its boundaries. The generic structure of the architectural context diagram is illustrated in the below Figure.

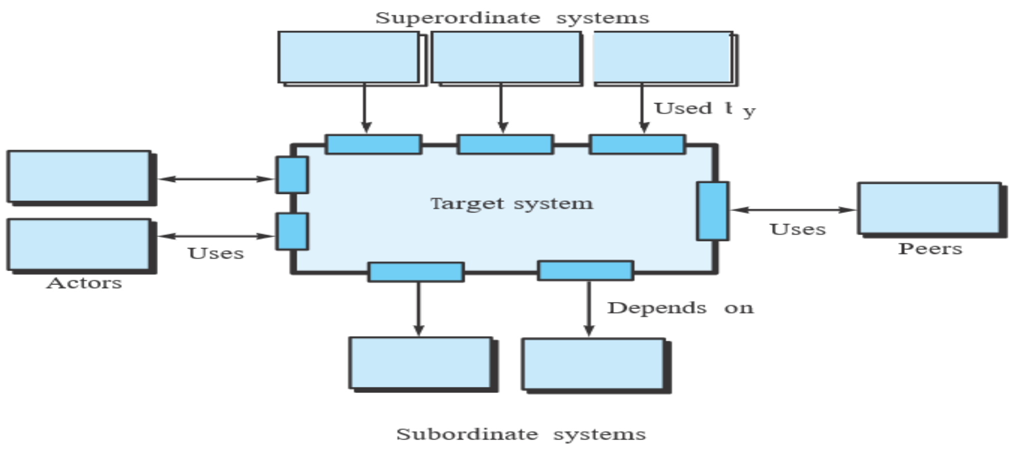


Fig: Architectural context diagram

Referring to the figure, systems that interoperate with the target system (the system for which an architectural design is to be developed) are represented as:

- Superordinate systems—those systems that use the target system as part of some higher-level processing scheme.

- Subordinate systems—those systems that are used by the target system and provide data or processing that are necessary to complete target sys- tem functionality.
- Peer-level systems—those systems that interact on a peer-to-peer basis

(i.e., information is either produced or consumed by the peers and the tar- get system.

- Actors—entities (people, devices) that interact with the target system by producing or consuming information that is necessary for requisite processing

Each of these external entities communicates with the target system through an interface (the small shaded rectangles).

To illustrate the use of the ACD, consider the home security function of the Safe-Home product. The overall **SafeHome** product controller and the Internet-based system are both superordinate to the security function and are shown above the function in Figure 13.6. The surveillance function is a peer system and uses (is used by) the home security function in later versions of the product. The home- owner and control panels are actors that producer and consume information used/produced by the security software. Finally, sensors are used by the security software and are shown as subordinate to it.

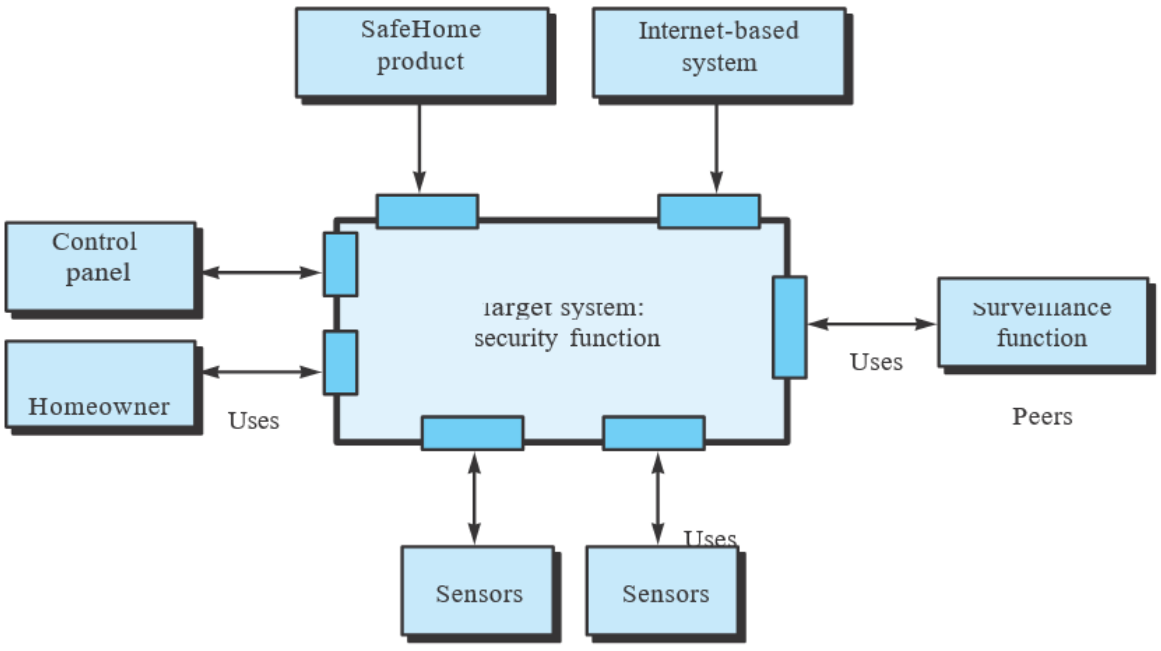


Fig: Architectural context diagram for the SafeHome security function

As part of the architectural design, the details of each interface shown in the above Figure would have to be specified. All data that flow into and out of the target system must be identified at this stage.

2 Defining Archetypes.

An archetype is a class or pattern that represents a core abstraction that is critical to the design of an architecture for the target system. In general, a relatively small set of archetypes is required to design even relatively complex systems. The target system architecture is composed of these archetypes, which represent stable elements of the architecture but may be instantiated many different ways based on the behavior of the system.

In many cases, archetypes can be derived by examining the analysis class's defined as part of the requirements model. Continuing the discussion of the **SafeHome** security function, you might define the following archetypes:

- Node. Represents a cohesive collection of input and output elements of the home security function. For example, a node might be composed of
  (1) Various sensors and
  (2) A variety of alarm (output) indicators.
- Detector. An abstraction that encompasses all sensing equipment that feeds information into the target system.
- Indicator. An abstraction that represents all mechanisms (e.g., alarm siren, flashing lights, and bell) for indicating that an alarm condition is occurring.
- Controller. An abstraction that depicts the mechanism that allows the arming or disarming of a node. If controllers reside on a network, they have the ability to communicate with one another.

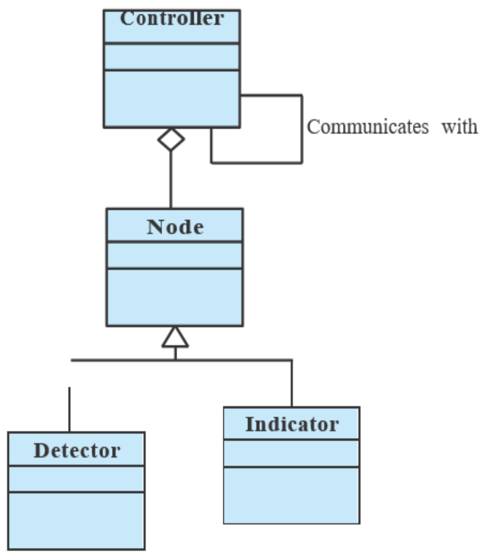


Fig: Relationtionships for SafeHome security function archetypes

Agility: Definition in Software Design and Architecture

Agility is the capability of a development process (including design and architecture) to accommodate change **rapidly**, **efficiently**, and **with minimal cost and disruption**.

- It emphasizes **iterative**, **incremental**, and **customer-focused** design.
- Agile design encourages **flexibility**, **early feedback**, and **continuous refinement** of architecture.
- Designs are kept **modular**, **loosely coupled**, and **evolutionary**, which makes them easy to modify.

Agility and Cost of Change

In traditional (prescriptive) models:

- The **cost of change increases dramatically** as the software progresses through the lifecycle—especially after design or coding stages.

In agile approaches:

- The **cost of change is kept low** throughout the lifecycle by:
  - **Continuous integration** and **frequent refactoring**
  - **Just-in-time architectural decisions**
  - **Simple and extensible design principles** (like SOLID)

Key Agility Principles in Design Engineering:

- **Design for change**: anticipating future changes (e.g., using design patterns or abstraction).
- **Evolve architecture**: architecture grows iteratively along with system functionality.
- **Feedback loops**: use of prototypes, stakeholder feedback, and test-driven design.

Performing User interface design: Golden rules:

Theo Mandel [Man97] coins three golden rules:

1. Place the user in control.
2. Reduce the user's memory load.
3. Make the interface consistent.

These golden rules actually form the basis for a set of user interface design principles that guide this important aspect of software design.

Place the User in Control:

During a requirements-gathering session for a major new information system, a key user was asked about the attributes of the window-oriented graphical interface.

"What I really would like," said the user solemnly, "is a system that reads my mind. It knows what I want to do before I need to do it and makes it very easy for me to get it done. That's all, just that."

Your first reaction might be to shake your head and smile, but pause for a moment. There was absolutely nothing wrong with the user's request. She wanted a system that reacted to her needs and helped her get things done. She wanted to control the computer, not have the computer control her.

Most interface constraints and restrictions that are imposed by a designer are intended to simplify the mode of interaction. But for whom?

As a designer, you may be tempted to introduce constraints and limitations to simplify the implementation of the interface. The result may be an interface that is easy to build, but frustrating to use. Mandel [Man97] defines a number of design principles that allow the user to maintain control:

Define interaction modes in a way that does not force a user into unnecessary or undesired actions. An interaction mode is the current state of the interface. For example, if spell check is selected in a word-processor menu, the software moves to a spell-checking mode. There is no reason to force the user to remain in spell-checking mode if the user desires to make a small text edit along the way. The user should be able to enter and exit the mode with little or no effort.

Provide for flexible interaction. Because different users have different interaction preferences, choices should be provided. For example, software might allow a user to interact via keyboard commands, mouse movement, a digitizer pen, a multi touch screen, or voice recognition commands. But every action is not amenable to every interaction mechanism. Consider, for example, the difficulty of using keyboard command (or voice input) to draw a complex shape.

Allow user interaction to be interruptible and undoable. Even when involved in a sequence of actions, the user should be able to interrupt the sequence to do something else (without losing the work that had been done). The user should also be able to "undo" any action.

Streamline interaction as skill levels advance and allow the interaction to be customized. Users often find that they perform the same sequence of interactions repeatedly. It is worthwhile to design a "macro" mechanism that enables an advanced user to customize the interface to facilitate interaction.

Hide technical internals from the casual user. The user interface should move the user into the virtual world of the application. The user should not be aware of the operating system, file management functions, or other arcane computing technology.

Design for direct interaction with objects that appear on the screen. The user feels a sense of control when able to manipulate the objects that are necessary to perform a task in a manner similar to what would occur if the object were a physical thing. For example, an application interface that allows a user to drag a document into the "trash" is an implementation of direct manipulation.

Reduce the User's Memory Load:

A well-designed user interface does not tax a user's memory because the more a user has to remember, the more error-prone the interaction will be When- ever possible, the system should "remember" pertinent information and assist the user with an interaction scenario that assists recall. Mandel [Man97] defines design principles that enable an interface to reduce the user's memory load:

Reduce demand on short-term memory. When users are involved in complex tasks, the demand on short-term memory can be significant. The interface should be designed to reduce the requirement to remember past actions, inputs, and results. This can be accomplished by providing visual cues that enable a user to recognize past actions, rather than having to recall them.

Establish meaningful defaults. The initial set of defaults should make sense for the average user, but a user should be able to specify individual preferences. However, a "reset" option should be available, enabling the redefinition of original default values.

Define shortcuts that are intuitive. When mnemonics are used to accomplish a system function (e.g., alt-P to invoke the print function), the mnemonic should be tied to the action in a way that is easy to remember (e.g., first letter of the task to be invoked).

The visual layout of the interface should be based on a real-world metaphor. For example, a bill payment system should use a checkbook and check register metaphor to guide the user through the bill paying process. This enables the user to rely on well-understood visual cues, rather than memorizing an arcane interaction sequence.

Disclose information in a progressive fashion. The interface should be organized hierarchically. That is, information about a task, an object, or some behavior should be presented first at a high level of abstraction. More detail should be presented after the user indicates interest.

Make the Interface Consistent:

The interface should present and acquire information in a consistent fashion. This implies that

(1) All visual information is organized according to design rules that are maintained throughout all screen displays,

(2) Input mechanisms are constrained to a limited set that is used consistently throughout the application, and

(3) Mechanisms for navigating from task to task are consistently defined and implemented. Mandel [Man97] defines a set of design principles that help make the interface consistent:

Allow the user to put the current task into a meaningful context.

Many interfaces implement complex layers of interactions with dozens of screen images. It is important to provide indicators (e.g., window titles, graphical icons, consistent color coding) that enable the user to know the context of the work at hand. In addition, the user should be able to determine where he has come from and what alternatives exist for a transition to a new task.

Maintain consistency across a complete product line.

A family of applications (i.e., a product line) should implement the same design rules so that consistency is maintained for all interaction.

If past interactive models have created user expectations, do not make changes unless there is a compelling reason to do so.

Once a particular interactive sequence has become a de facto standard (e.g., the use of alt-S to save a file), the user expects this in every application encountered. A change (e.g., using alt-S to invoke scaling) will cause confusion.

The interface design principles discussed in this and the preceding sections provide you with basic guidance. In the sections that follow, you'll learn about the interface design process itself.