UNIT – 3

SOFTWARE ARCHITECTURE

(Architecture □ Architecture Style □ Architecture Pattern □ Design)

What Is Architecture? : Software architecture must model the structure of a system. 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) analyze the effectiveness of the design in meeting its stated requirements
(2) architectural alternatives at a stage when making design changes is still relatively easy
(3) Reduce the risks associated with the construction of the software.

Architectural design focuses on the representation of the structure of software components, their properties, and interactions.

Why Is Architecture Important?: Three key reasons that software architecture is important:
- Representations of software architecture are an enabler for communication between all parties (stakeholders) interested in the development of a computer-based system.
- The architecture highlights early design decisions that will have a profound impact on all software engineering work that follows and, as important, on the ultimate success of the system as an operational entity.
- Architecture "constitutes a relatively small, intellectually graspable model of how the system is structured and how its components work together".

ARCHITECTURAL GENRES

In the context of architectural design, genre implies a specific category within the overall software domain. Within each category, you encounter a number of subcategories. For example, within the genre of buildings, you would encounter the following general styles: houses, condos, apartment buildings, office buildings, industrial building, warehouses, and so on.

Grady Booch suggests the following architectural genres for software-based systems:
- Artificial intelligence—Systems that simulate or augment human cognition, locomotion, or other organic processes.
- Commercial and nonprofit—Systems that are fundamental to the operation of a business enterprise.
- Communications—Systems that provide the infrastructure for transferring and managing data, for connecting users of that data, or for presenting data at the edge of an infrastructure.
- Content authoring—Systems that are used to create or manipulate textual or multimedia artifacts.
- Devices—Systems that interact with the physical world to provide some point service for an individual.

- Entertainment and sports—Systems that manage public events or that provide a large group entertainment experience.
- Financial—Systems that provide the infrastructure for transferring and managing money and other securities.
- Games—Systems that provide an entertainment experience for individuals or groups.
- Government—Systems that support the conduct and operations of a local, state, federal, global, or other political entity.
- Industrial—Systems that simulate or control physical processes.
- Legal—Systems that support the legal industry.
- Medical—Systems that diagnose or heal or that contribute to medical research.
- Military—Systems for consultation, communications, command, control, and intelligence (C4I) as well as offensive and defensive weapons.
- Operating systems—Systems that sit just above hardware to provide basic software services.
- Platforms—Systems that sit just above operating systems to provide advanced services.
- Scientific—Systems that are used for scientific research and applications.
- Tools—Systems that are used to develop other systems.
- Transportation—Systems that control water, ground, air, or space vehicles.
- Utilities—Systems that interact with other software to provide some point service.

Architectural Design elements

Architectural 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.

Architectural Styles:
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.

An architectural pattern, like an architectural style, imposes a transformation on the design of an architecture.

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. 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. the architecture without concern about other clients (because the client components operate independently).

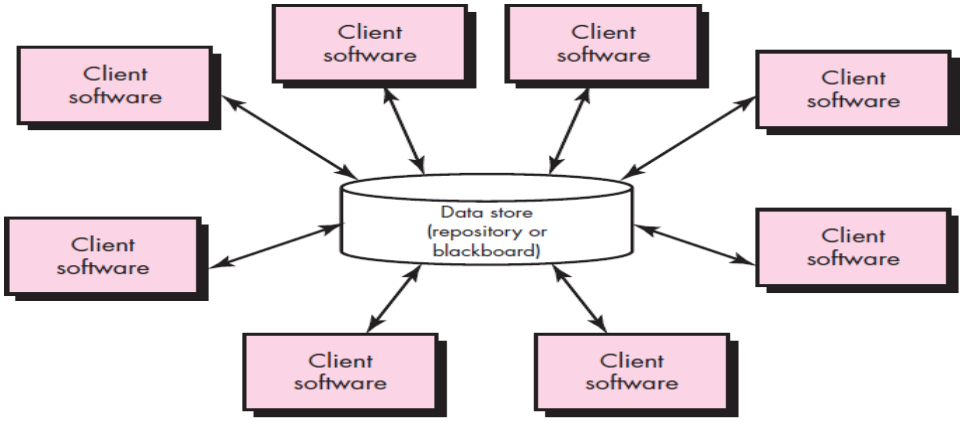


Fig 9.1: Data-centered architecture

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.

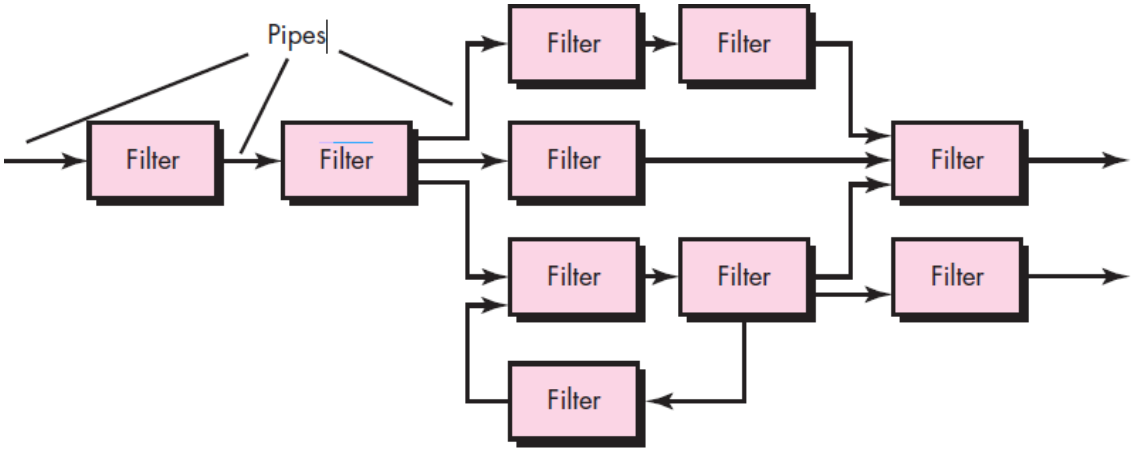


Fig 9.2: Data-flow architecture

Call and return architectures.

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.

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

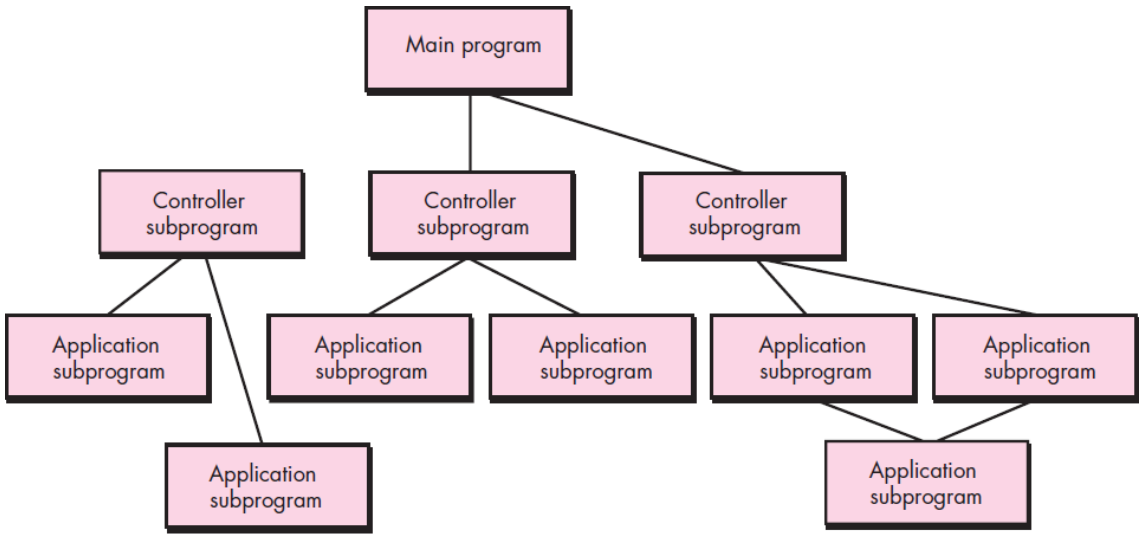


Fig 9.3: Main program / subprogram architecture

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.

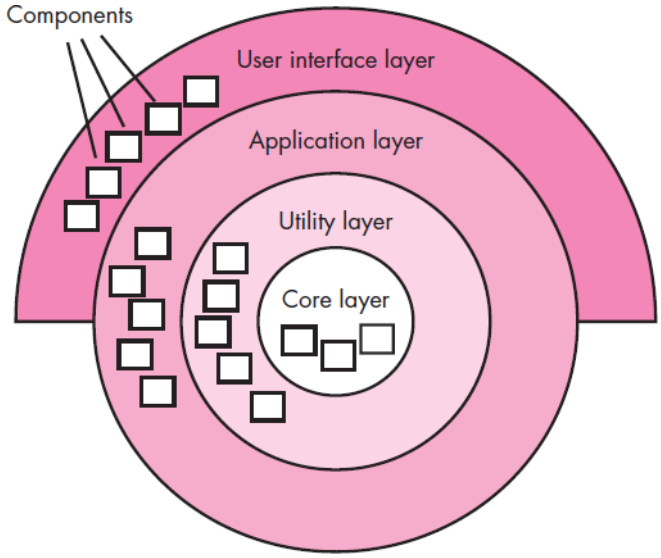


Fig 9.4: Layered architecture

Layered architectures. The basic structure of a layered architecture is illustrated in Figure 9.4. 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.

Architectural Patterns: Architectural patterns address an application-specific problem. The pattern proposes an architectural solution that can serve as the basis for architectural design.

Examples:

- Concurrency—applications must handle multiple tasks in a manner that simulates parallelism
- operating system process management pattern
- task scheduler pattern
    - Persistence—Data persists if it survives past the execution of the process that created it. Two patterns are common:
- a database management system pattern that applies the storage and retrieval capability of a DBMS to the application architecture
- an application level persistence pattern that builds persistence features into the application architecture
- Distribution— the manner in which systems or components within systems communicate with one another in a distributed environment
    - A broker acts as a 'middle-man' between the client component and a server component.

ARCHITECTURAL DESIGN

The design should define the external entities (other systems, devices, people) that the software interacts with and the nature of the interaction. This information can generally be acquired from the requirements model and all other information gathered during requirements engineering. 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 software components that implement each archetype. This process continues iteratively until a complete architectural structure has been derived.

Representing the System in Context: Architectural context represents how the software interacts with entities external to its boundaries. At the architectural design level, a software architect uses an architectural context diagram (ACD) to model the manner in which software interacts with entities external to its boundaries, systems that interoperate with the target system 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 system 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 target 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).

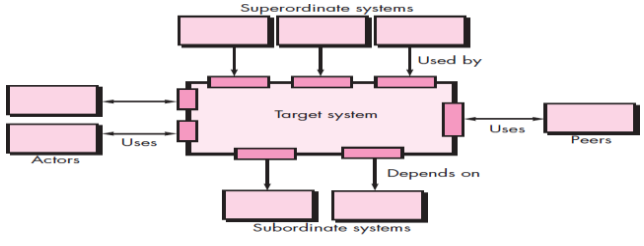


Fig 9.5: Architectural context diagram

Defining Archetypes: Archetypes are the abstract building blocks of an architectural design. 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.

The following are the archetypes for safeHome: Node, Detector, Indicator., Controller.

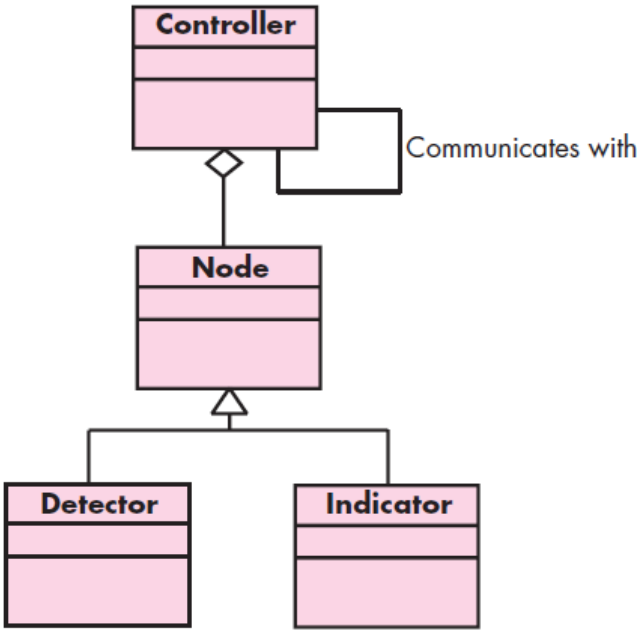


Fig 9.7: UML relationships for safehome function Archetypes

Refining the Architecture into Components: As the software architecture is refined into components, the structure of the system begins to emerge. The architecture must accommodate many infrastructure components that enable application components but have no business connection to the application domain. For example, memory management components, communication components, database components, and task management components are often integrated into the software architecture.

As an example for SafeHome home security, the set of top-level components that address the following functionality:

- External communication management—coordinates communication of the security function with external entities such as other Internet-based systems and external alarm notification.
- Control panel processing—manages all control panel functionality.
- Detector management—coordinates access to all detectors attached to the system.
- Alarm processing—verifies and acts on all alarm conditions.

- Each of these top-level components would have to be elaborated iteratively

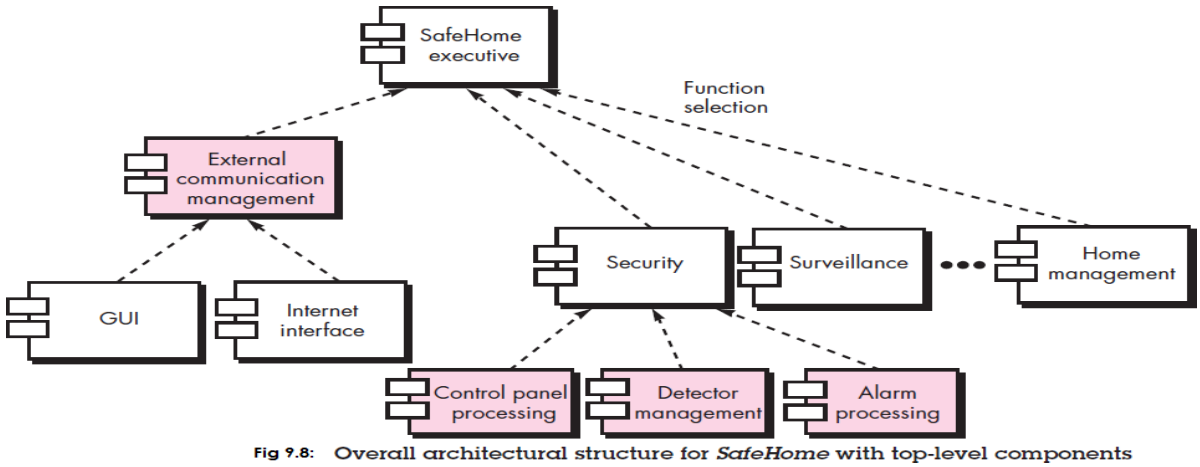


Fig 9.8: Overall architectural structure for *SafeHome* with top-level components

Describing Instantiations of the System: The architectural design that has been modeled to this point is still relatively high level. The context of the system has been represented, archetypes that indicate the important abstractions within the problem domain have been defined, the overall structure of the system is apparent, and the major software components have been identified. However, further refinement is still necessary.

Component-Level Design

WHAT IS A COMPONENT?: A component is a modular building block for computer software. More formally, component is "a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces."

An Object-Oriented View: In the context of object-oriented software engineering, a component contains a set of collaborating classes. Each class within a component has been fully elaborated to include all attributes and operations that are relevant to its implementation.

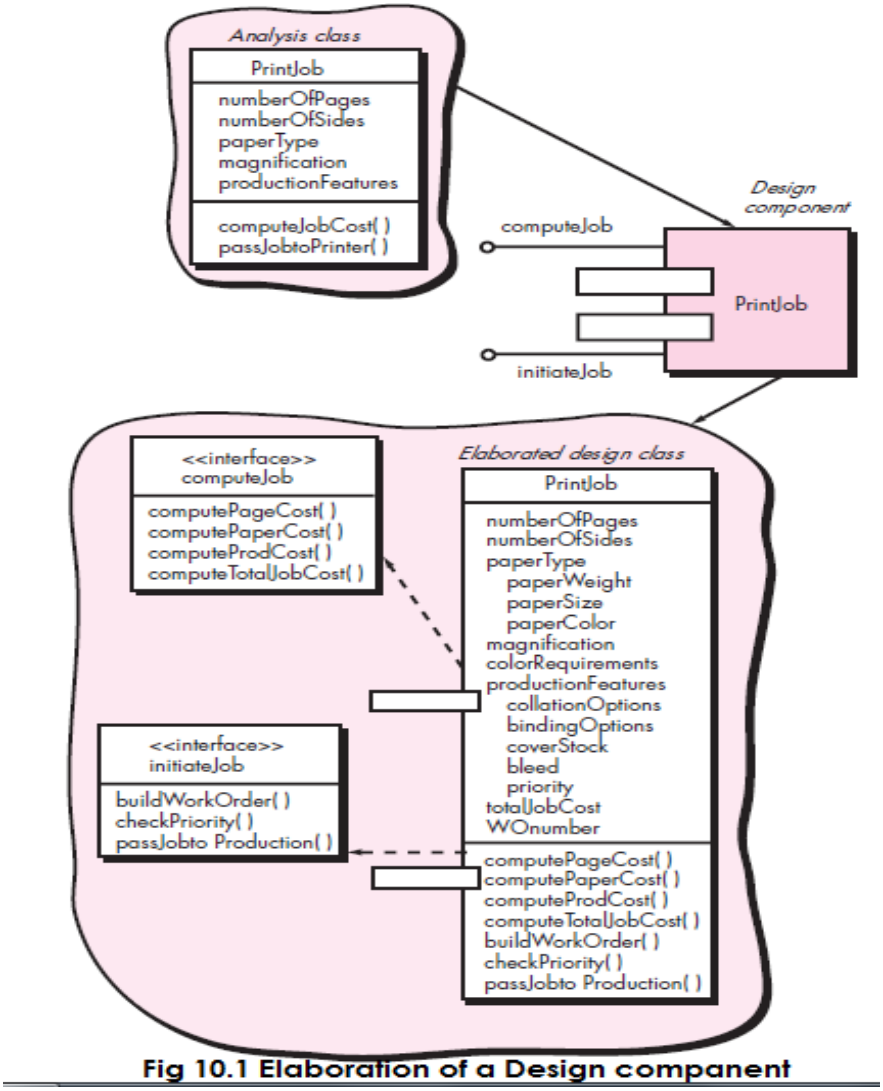


Fig 10.1 Elaboration of a Design component

The Traditional View: In the context of traditional software engineering, a component is a functional element of a program that incorporates processing logic, the internal data structures that are required to implement the processing logic, and an interface that enables the component to be invoked and data to be passed to it. A traditional component, also called a module, resides within the software architecture and serves one of three important roles:
(1) a control component that coordinates the invocation of all other problem domain components,
(2) a problem domain component that implements a complete or partial function that is required by the customer, or
(3) an infrastructure component that is responsible for functions that support the processing required in the problem domain.

Like object-oriented components, traditional software components are derived from the analysis model. To achieve effective modularity, design concepts like functional independence are applied as components are elaborated.

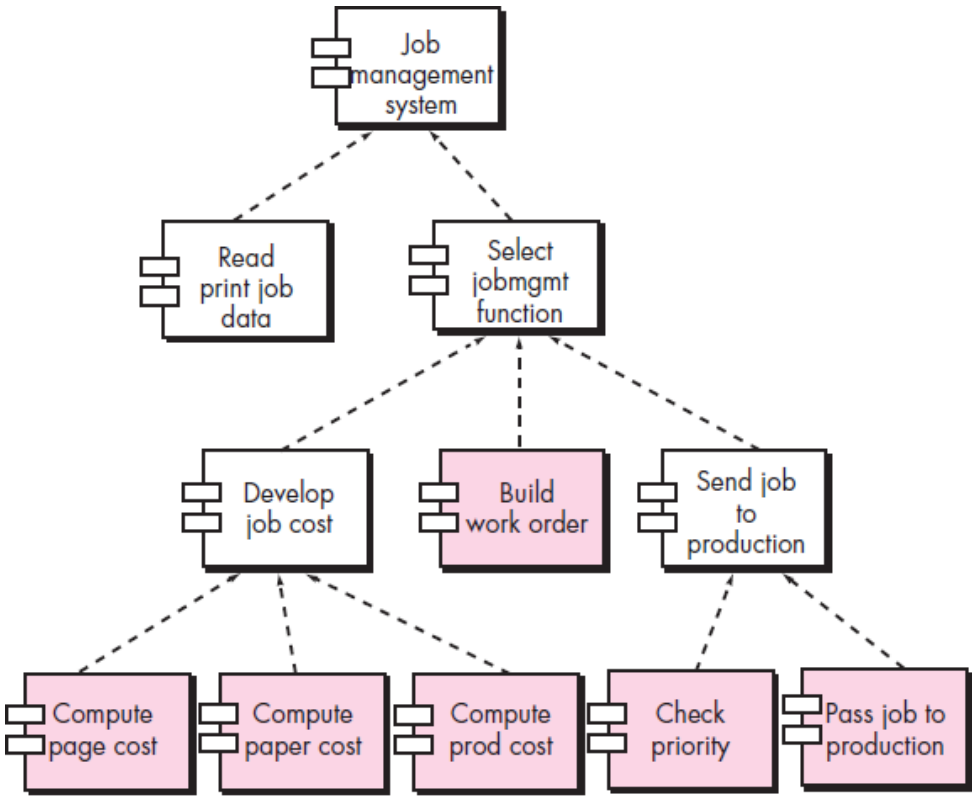


Fig 10.2: Structure chart for a traditional system

During component-level design, each module is elaborated. The module interface is defined explicitly. That is, each data or control object that flows across the interface is represented. The data structures that are used internal to the module are defined. The algorithm that allows the module to accomplish its intended function is designed using the stepwise refinement approach. The behavior of the module is sometimes represented using a state diagram.

A Process-Related View: The object-oriented and traditional views of component-level design assume that the component is being designed from scratch. That is, you have to create a new component based on specifications derived from the requirements model.

Over the past two decades, the software engineering community has emphasized the need to build systems that make use of existing software components or design patterns. In essence, a catalog of proven design or code-level

components is made available to you as design work proceeds. As the software architecture is developed, you choose components or design patterns from the catalog and use them to populate the architecture.

Because these components have been created with reusability in mind, a complete description of their interface, the function(s) they perform, and the communication and collaboration they require are all available.

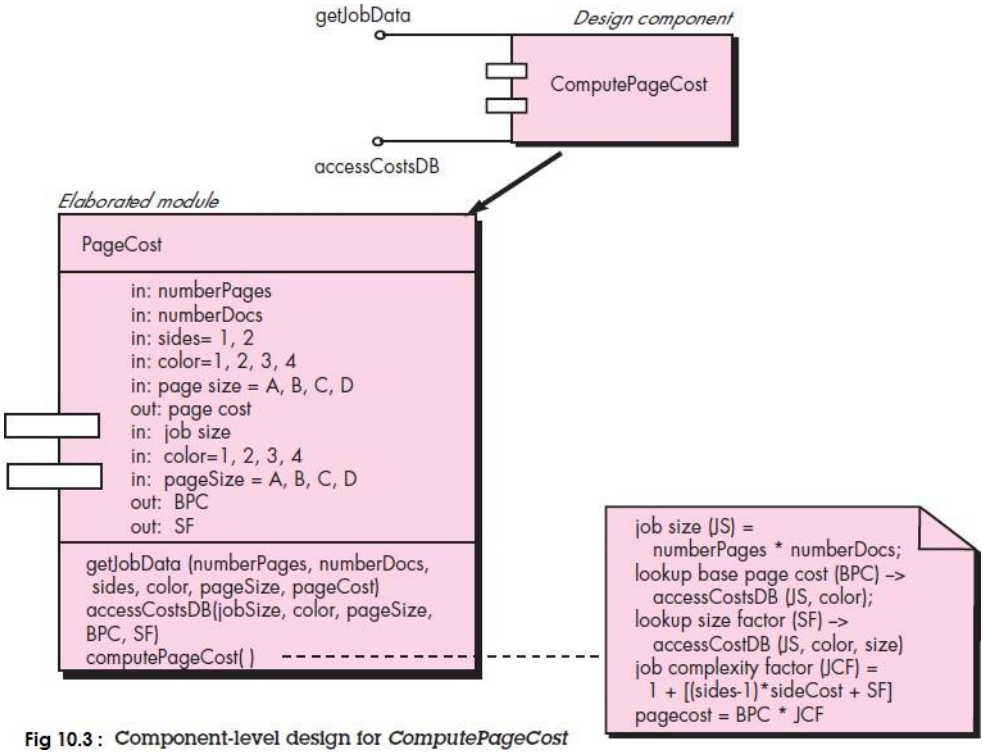


Fig 10.3 : Component-level design for *ComputePageCost*

DESIGNING CLASS-BASED COMPONENTS

The Open-Closed Principle (OCP). "A module [component] should be open for extension but closed for modification". Stated simply, you should specify the component in a way that allows it to be extended without the need to make internal (code or logic-level) modifications to the component itself.

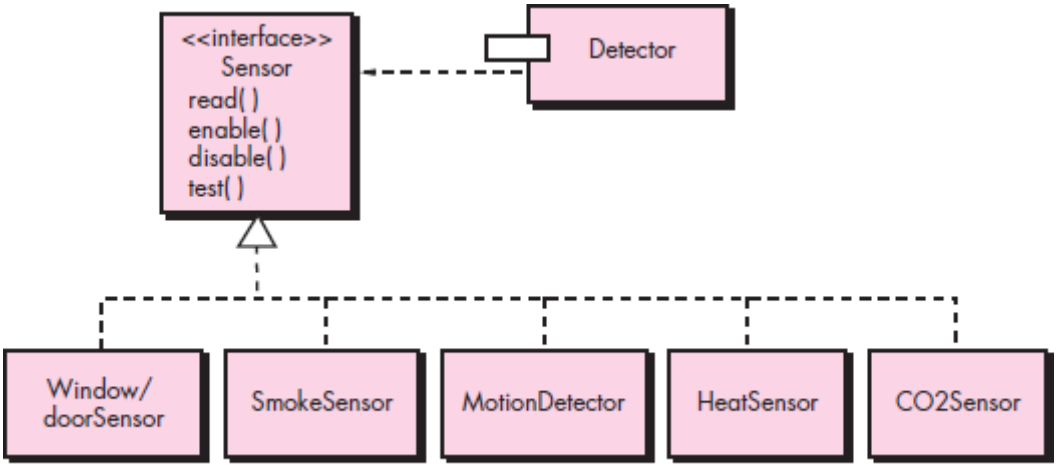


Fig 10.4: Following the OCP

The Liskov Substitution Principle (LSP). "Subclasses should be substitutable for their base classes". This design principle, originally proposed by Barbara Liskov, suggests that a component that uses a base class should continue to function properly if a class derived from the base class is passed to the component instead. LSP demands that any class derived from a base class must honor any implied contract between the base class and the components that use it.

Dependency Inversion Principle (DIP). "Depend on abstractions. Do not depend on concretions". As we have seen in the discussion of the OCP, abstractions are the place where a design can be extended without great complication.

The Interface Segregation Principle (ISP). "ISP suggests that you should create a specialized interface to serve each major category of clients. Only those operations that are relevant to a particular category of clients should be specified in the interface for that client. If multiple clients require the same operations, it should be specified in each of the specialized interfaces.

Martin suggests additional packaging principles that are applicable to component-level design: The Release Reuse Equivalency Principle (REP). "The granule of reuse is the granule of release".,it is often advisable to group reusable classes into packages that can be managed and controlled as newer versions evolve.
The Common Closure Principle (CCP). "Classes that change together belong together." Classes should be packaged cohesively. That is, when classes are packaged as part of a design, they should address the same functional or behavioral area.
The Common Reuse Principle (CRP). "Classes that aren't reused together should not be grouped together" If classes are not grouped cohesively, it is possible that a class with no relationship to other classes within a package is changed.

Component-Level Design Guidelines: Ambler suggests the following guidelines for components, their interfaces, and the dependencies and inheritance characteristics

Components. Naming conventions should be established for components that are specified as part of the architectural model and then refined and elaborated as part of the component-level model.
Interfaces. Interfaces provide important information about communication and collaboration. Dependencies and Inheritance. For improved readability, it is a good to model dependencies from left to right and inheritance from bottom (derived classes) to top (base classes).

Cohesion: Cohesion is the "single-mindedness" of a component. Within the context of component-level design for object-oriented systems.

Functional. Exhibited primarily by operations, this level of cohesion occurs when a component performs a targeted computation and then returns a result.

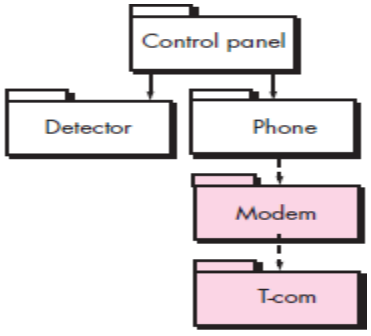


Fig 10.5: Layer cohesion

Layer. Exhibited by packages, components, and classes, this type of cohesion occurs when a higher layer accesses the services of a lower layer, but lower layers do not access higher layers.

Communicational. All operations that access the same data are defined within one class. In general, such classes focus solely on the data in question, accessing and storing it.

Coupling: Coupling is a qualitative measure of the degree to which classes are connected toone another. As classes (and components) become more interdependent, coupling increases. An important objective in component-level design is to keep coupling as low as is possible. following coupling categories:

Content coupling. Occurs when one component "surreptitiously modifies data that is internal to another component". This violates information hiding—a basic design concept.

Common coupling. Occurs when a number of components all make use of a global variable. Although this is sometimes necessary (e.g., for establishing default values that are applicable throughout an application),

Control coupling. Occurs when operation A() invokes operation B() and passes a control flag to B. The control flag then "directs" logical flow within B. The problem with this form of coupling is that an unrelated change in B can result in the necessity to change the meaning of the control flag that A passes. If this is overlooked, an error will result.

Stamp coupling. Occurs when ClassB is declared as a type for an argument of an operation of ClassA. Because ClassB is now a part of the definition of ClassA, modifying the system becomes more complex.

Data coupling. Occurs when operations pass long strings of data arguments. The "bandwidth" of communication between classes and components grows and the complexity of the interface increases. Testing and maintenance are more difficult.

Routine call coupling. Occurs when one operation invokes another. This level of coupling is common and is often quite necessary. However, it does increase the connectedness of a system.

Type use coupling. Occurs when component A uses a data type defined in component B (e.g., this occurs whenever "a class declares an instance variable or a local variable as having another class for its type". If the type definition changes, every component that uses the definition must also change.

Inclusion or import coupling. Occurs when component A imports or includes a package or the content of component B.

External coupling. Occurs when a component communicates or collaborates with infrastructure components (e.g., operating system functions, database capability, telecommunication functions). Although this type of coupling is necessary, it should be limited to a small number of components or classes within a system.

CONDUCTING COMPONENT-LEVEL DESIGN: The following steps represent a typical task set for component-level design, when it is applied for an object-oriented system.
Step 1. Identify all design classes that correspond to the problem domain. Using the requirements and architectural

model, each analysis class and architectural component is elaborated.

Step 2. Identify all design classes that correspond to the infrastructure domain. These classes are not described in the requirements model and are often missing from the architecture model, but they must be described at this point. Classes and components in this category include GUI components, operating system components, and object and data management components.

Step 3. Elaborate all design classes that are not acquired as reusable components. Elaboration requires that all interfaces, attributes, and operations necessary to implement the class be described in detail. Design heuristics (e.g., component cohesion and coupling) must be considered as this task is conducted.

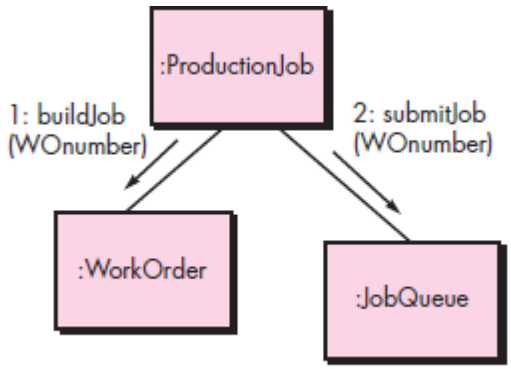


Fig 10.6: Collaboration diagram with messaging

Step 3a. Specify message details when classes or components collaborate. The requirements model makes use of a collaboration diagram to show how analysis classes collaborate with one another. As component-level design proceeds, it is sometimes useful to show the details of these collaborations by specifying the structure of messages that are passed between objects within a system. Although this design activity is optional, it can be used as a precursor to the specification of interfaces that show how components within the system communicate and collaborate.

Step 3b. Identify appropriate interfaces for each component. Within the context of component-level design, a UML interface is "a group of externally visible (i.e., public) operations. The interface contains no internal structure, it has no attributes, no associations. Interface is the equivalent of an abstract class that provides a controlled connection between design classes.

Step 3c. Elaborate attributes and define data types and data structures required to implement them. In general, data structures and types used to define attributes are defined within the context of the programming language that is to be used for implementation. UML defines an attribute's data type using the following syntax:

name : type-expression _ initial-value {property string}

where name is the attribute name, type expression is the data type, initial value is the value that the attribute takes when an object is created, and property-string defines a property or characteristic of the attribute.

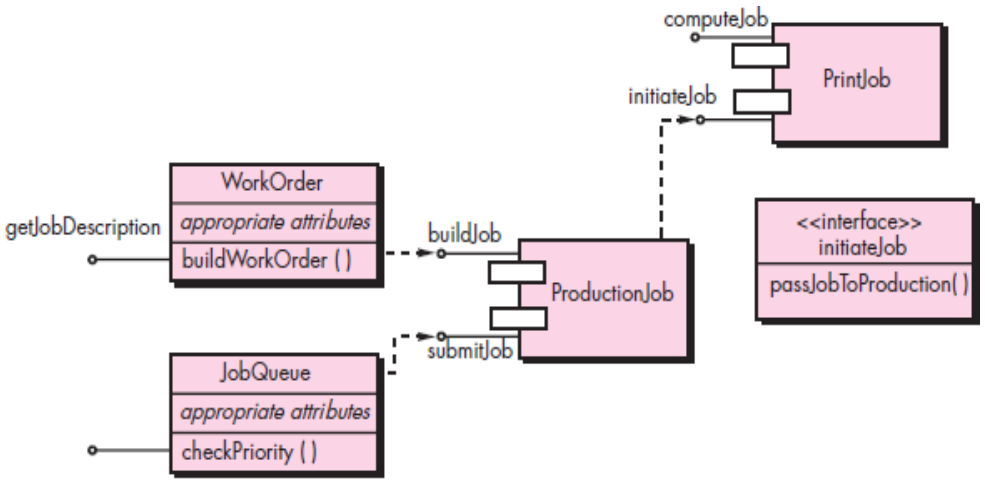


Fig 10.7: Refactoring interfaces and class definitions for PrintJob

Step 3d. Describe processing flow within each operation in detail. This may be accomplished using a programming language-based pseudo code or with a UML activity diagram. Each software component is elaborated through a number of iterations that apply the stepwise refinement concept.

Step 4. Describe persistent data sources (databases and files) and identify the classes required to manage them. Databases and files normally transcend the design description of an individual component. In most cases, these persistent data stores are initially specified as part of architectural design. However, as design elaboration proceeds, it is often useful to provide additional detail about the structure and organization of these persistent data sources.

Step 5. Develop and elaborate behavioral representations for a class or component. UML state diagrams were used as part of the requirements model to represent the externally observable behavior of the system and the more localized behavior of individual analysis classes. During component-level design, it is sometimes necessary to model the behavior of a design class. The dynamic behavior of an object is affected by events that are external to it and the current state of the object as illustrated in Figure

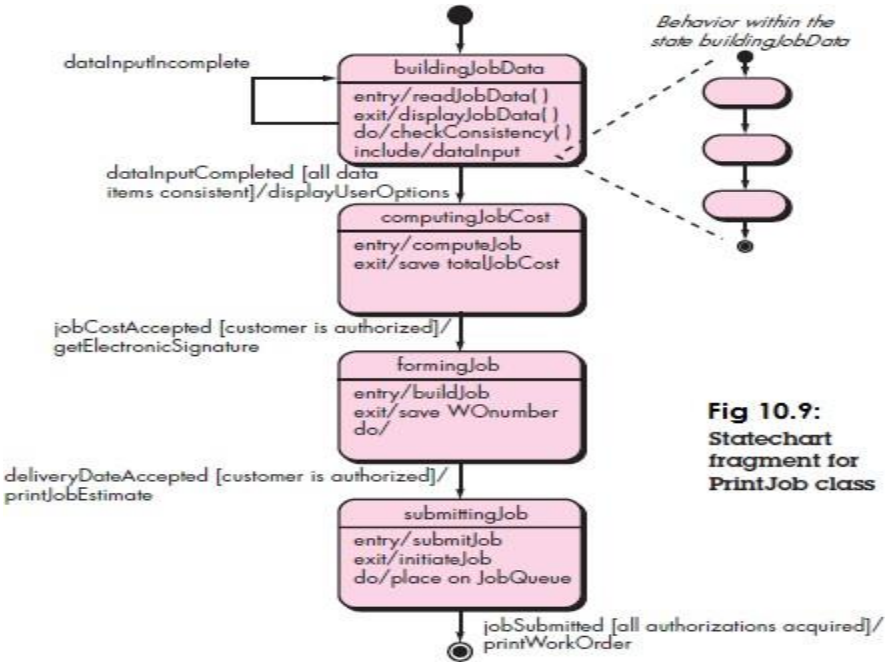


Fig 10.9: **Statechart fragment for PrintJob class**

Step 6. Elaborate deployment diagrams to provide additional implementation detail. Deployment diagrams are used as part of architectural design and are represented in descriptor form. In this form, major system functions are represented within the context of the computing environment that will house them.

Step 7. Refactor every component-level design representation and always consider alternatives. The design is an iterative process. The first component-level model you create will not be as complete, consistent, or accurate as the n^{th} iteration you apply to the model. It is essential to refactor as design work is conducted. Develop alternatives and consider each carefully, using the design principles and concepts.

User Interface Design

The Golden Rules

This section discusses three principles of user interface design. The first is to place the user in control (which means have the computer interface support the user's understanding of a task and do not force the user to follow the computer's way of doing things). The second (reduce the user's memory load) means place all necessary information on the screen at the same time. The third is consistency of form and behavior.

The three "golden rules" are:

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 software design action.

Place the User in Control

Mandel 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. The user should always 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 by using keyboard commands, mouse movements, digitizer pen or voice recognition commands.
- Allow user interaction to be interruptible and undoable. A user should be able to interrupt a sequence of actions to do something else without losing the work that has been done. The user should always be able to "undo" any action.
- Streamline interaction as skill levels advance and allow the interaction to be customized. Allow to design a macro if the user is to perform the same sequence of actions repeatedly.
- Hide technical internals from the casual user. The user interface should move the user into the virtual world of the application. A user should never be required to type O/S commands from within application software.
- 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.

Reduce the User's Memory Load

Whenever possible, the system should "remember" pertinent information and assist the user with an interaction scenario that assists recall.

- Reduce demand on short-term memory. Provide visual cues that enable a user to recognize past actions, rather than having to recall them.
- Establish meaningful defaults. A user should be able to specify individual preferences; however, a reset option should be available to enable the redefinition of original default values.
- Define shortcuts that are intuitive. "Example: Alt-P to print. Using easy to remember mnemonics."
- The visual layout of the interface should be based on a real world metaphor. Enable the user to rely on well-understood visual cues, rather than remembering an arcane interaction sequence. For a bill payment system use a check book and check register metaphor to guide the user through the process.

- Disclose information in a progressive fashion. The interface should be organized hierarchically. The information should be presented at a high level of abstraction.

Make the Interface Consistent

The interface should present and acquire information in a consistent manner:

1. All visual information is organized according to a design standard that is maintained throughout all screen displays,
2. Input mechanisms are constrained to a limited set that is used consistently throughout the application,
3. Mechanisms for navigating from task to task are consistently defined and implemented.

A set of design principles that help make the interface consistent:

Allow the user to put the current task into a meaningful context. 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 family of applications. "MS Office Suite"
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 (Alt-S ➔ save file), the user expects this in every application she encounters.

User Interface Analysis and Design
The overall process for analyzing and designing a UI begins with the creation of models of system.

User Interface Design Models

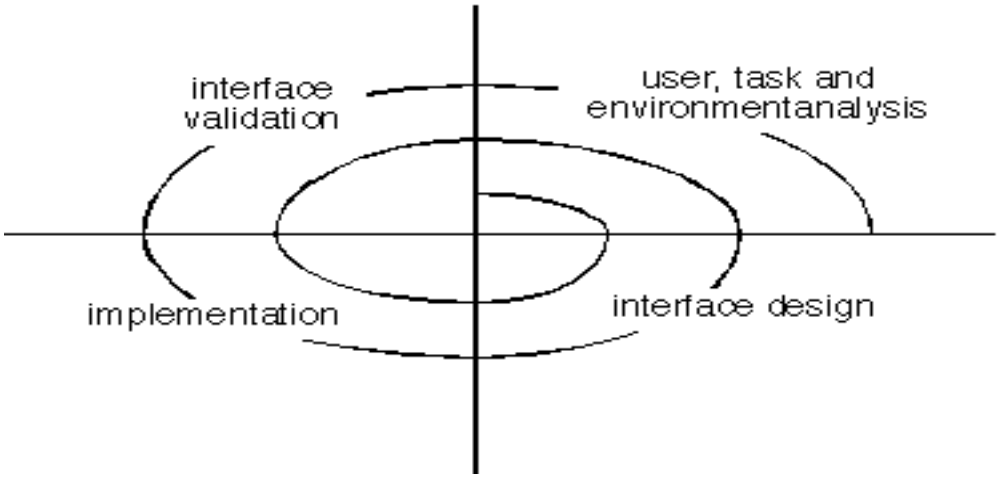
Four different models come into play when a user interface is to be analyzed and designed. "Prototyping"

1. User model: a profile of all end users of the system
   Users can be categorized as:
   - Novices: No syntactic and little semantic knowledge of the system.
   - Knowledgeable, intermittent users: reasonable knowledge of the system.
   - Knowledgeable, frequent users: good syntactic and semantic knowledge of the system.
2. Design model: a design realization of the user model that incorporates data, architectural, interface, and procedural representations of the software.
3. Mental model (system perception): the user's mental image of what the interface is. The user's mental model shapes how the user perceives the interface and whether the UI meets the user's needs.
4. Implementation model: the interface "look and feel of the interface" coupled with all supporting information (documentation) that describes interface syntax and semantics.

The Process

The analysis and design process for UIs is iterative and can be represented using a spiral model.
The user interface analysis and design process encompasses four distinct framework activities:

1. User, task and environment analysis and modeling.
2. Interface design
3. Interface construction (implementation)
4. Interface validation



The figure 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. The construction activity normally begins with the creation of a prototype that enables usage scenarios to be evaluated.

Interface Analysis

you better understand the problem before you attempt to design a solution.
Interface design analysis 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,
(4) The environment in which these tasks will be conducted.

User Analysis

The only way that a designer 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. This can be accomplished by:

User Interviews: The software team meets with the end-users to better understand their needs, motivations, work culture, and a myriad of other issues.
Sales Input: Sales people meet with customers and users to help developers categorize users and better understand their requirements.
Marketing Input: Market analysis can be invaluable in the definition of market segments while providing an understanding of how each segment might use the software in different ways.
Support Input: Support staff talks with users on a daily basis, making them the most likely source of information on what works and what doesn't, and what they like and what they don't.

Interface Design Steps

Once interface analysis has been completed, all tasks required by the end-user have been identified in detail.

1. Using information developed during interface analysis (Section 12.3), define interface objects and actions (operations).
2. Define events (user actions) that will cause the state of the user interface to change. Model this behavior.
3. Depict each interface state as it will actually look to the end-user.
4. Indicate how the user interprets the state of the system from information provided through the interface.

Interface Design Patterns

Patterns are available for

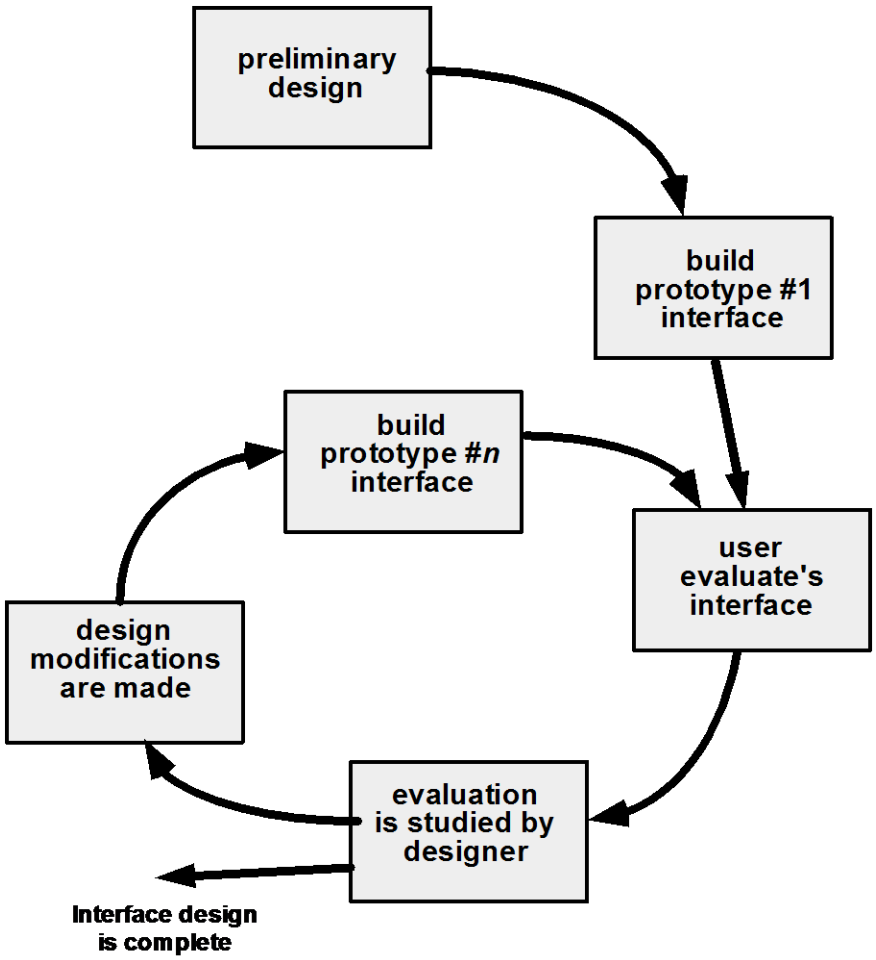
- The complete UI
- Page layout
- Forms and input
- Tables
- Direct data manipulation
- Navigation
- Searching
- Page elements
- e-Commerce

Design Issues

- Response time: System response time has 2 important characteristics: length and variability. Variability refers to the deviation from average response time.
- Help facilities: Help must be available for all system functions. Include help menus, print documents.
- Error handling: describe the problem in a language the user can understand. Never blame the user for the error that occurred.
- Menu and command labeling: menu options should have corresponding commands. Use control sequences for commands.
- Application accessibility: especially for the physically challenged.
- Internationalization: The Unicode standard has been developed to address the daunting challenge of managing dozens of natural languages with hundred of characters and symbols.

Design Evaluation

Two interface design evaluation techniques are mentioned in this section, usability questionnaires and usability testing. The process of learning how to design good user interfaces often begins with learning to identify the weaknesses in existing products.



Once the first prototype is built, the designer can collect a variety of qualitative and quantitative data that will assess in evaluating the interface.