UNIT - II

Software Design

Topics

- Software Design Fundamentals
- Design Standards - Design Type
- Design Model – Architectural design, Software Architecture
- Software Design Methods
- Top Down, Bottom Up
- Module Division (Refactoring)
- Module Coupling
- Component Level Design
- User Interface Design
- Pattern Oriented Design
- Web Application Design
- Design Reuse
- Concurrent Engineering in Software Design
- Design Life-Cycle Management

Software Design

Introduction

- Software design is the process where the software architecture is built and refined using the end user requirements.
- In the waterfall model, the software product will have a large number of parts which can be designed separately from each other and later can be joined.
- If you need to break the software product design into many parts then either top down approach or bottom up approach can be used.
- In top down approach, the software design for the entire product is built first and later its parts are designed.
- In bottom up approach, the software product parts are designed first and later they are joined to create design of the entire product.
- With each change request there will be a different version of the software design.
- Maintaining these design versions through the development process is very important.
- It is necessary to make sure that the right design is used for software construction.
- Software design should be robust and should also be able to take change requests without any problems.

Software Design

Introduction

- When a software product is incrementally built then the later versions of the software design should be fully compatible with the original software design.
- There are many software design techniques available to make appropriate software designs for the product being built.
- They include prototyping, structural models, object oriented design, entity relationship models etc.
- Refactoring is a design technique which is used in iterative models.
- When the design become bulky after many iterations of development, it starts getting crumbling against new features being added to the product.
- In such cases, the design is changed so that new factors of the features being added are incorporated.
- Software design development can be likened to designing a physical product.
- Suppose a new car model is to be developed.
- The car design is broken down into separate components and in the end assembling them will become a complete design for the car model.
- Various factors are considered during the design of the components.

Software Design

Introduction

- Suppose one factor to be considered is that during a car accident, the car body should take most of the impact and the passengers should get the least impact, so that injury to car passengers can be minimized during accidents.
- For this to happen, the car body should be made of material that can collapse on impact and thus take most of the impact.
- So during design, when selecting the material of the car with safety in mind, the body is one of the prime considerations.
- Similarly, an aerodynamic body helps in keeping the car from rolling over during accidents and thus it is a prime safety factor that the car body should be aerodynamic.
- During design, one consideration is also made that though each component is developed separately, after assembly the components should work with each other without any problems.
- That means assembling does not create any problems in the product itself.
- Similar considerations are also done when software products are designed.

Software Design

Introduction

- In fact, in designing software systems, consideration is given to things like how well the system will be maintained during operation and how easily the system will be actually developed and be tested.
- Software design is done using modeling languages like UML and using notation methods like use cases and activity diagrams.
- We will learn all about software design considerations, workflows involved in design, etc.

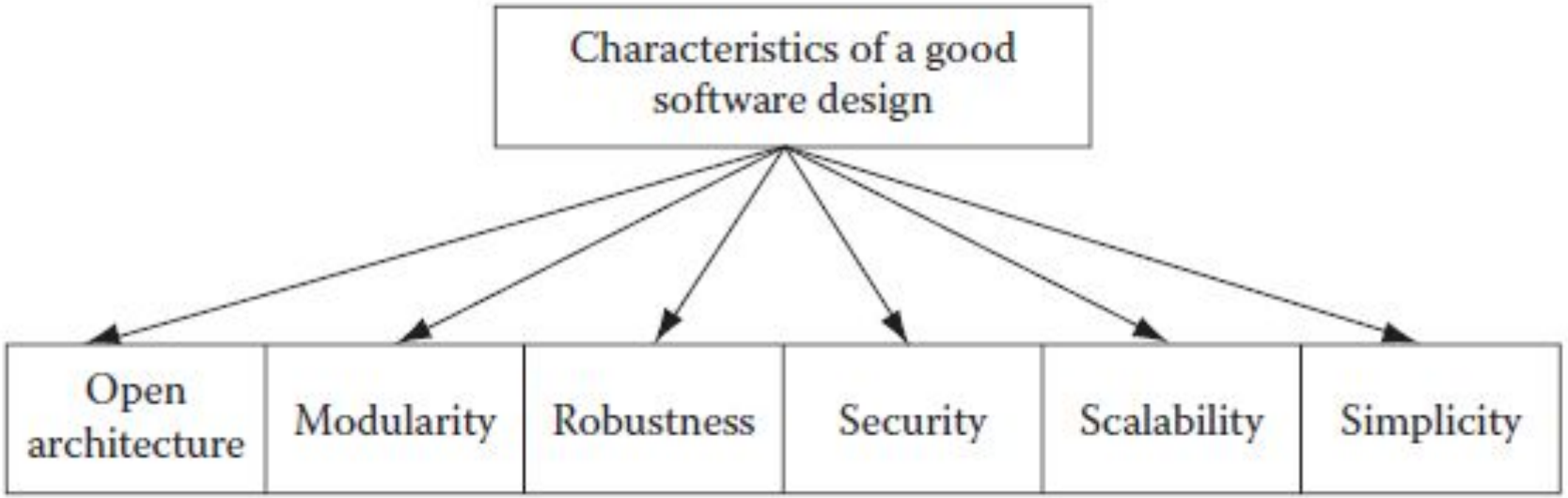
Software Design

Software Design Fundamentals

- When a building is constructed, a good foundation is laid out for the building, so that the building will have a long lifespan and will not collapse.
- Similarly, it is given a strong and resilient structure, so that even in case of an earthquake, it will not fall down.
- Similarly, software design provides the foundation and structure upon which the software system is constructed.
- The design should provide a sound, resilient and scalable structure to support the software system (Figure below-Characteristics of a good software design).
- In these days, most software systems are built incrementally.
- In the beginning, a software system may consist of only a few features.
- The feature set is expanded in future releases as and when it becomes necessary to include them in the system.
- If proper structure is not provided from the very beginning, the addition of these new features will make the system unstable.
- To deal with this problem, a technique called refactoring is used on these agile projects where incremental software development is done.

Software Design

Software Design Fundamentals



- Some of the design techniques that help make good software design include open architecture, modularity and scalability.
- The current trend of service-oriented architecture (SOA) has also helped tremendously in changing the design concepts.
- SOA is built on Web services and loose coupling of software components.
- The asynchronous messaging method of SOA is a vital aspect for developing Web-based applications.

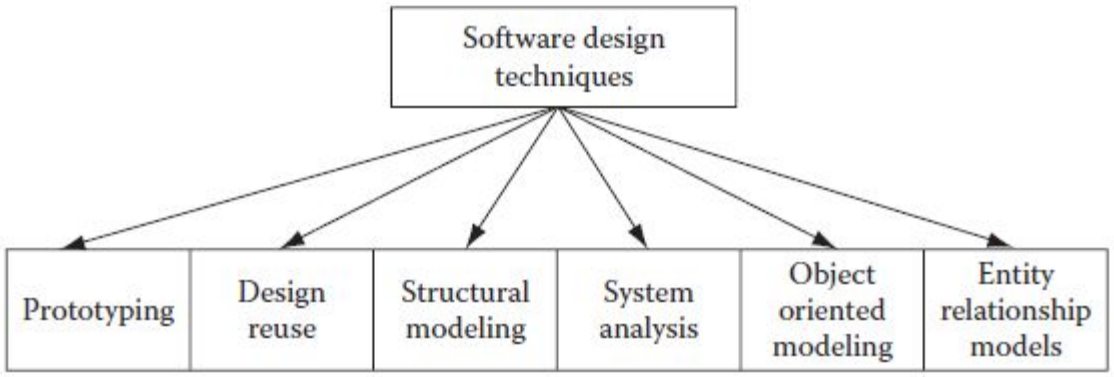
Software Design

Design Standards

- If design standards are implemented on a project, then it will help in streamlining activities that are involved during the software design phase.
- Some industry standards for software design include operator interface standards, test scenarios, safety standards, design constraints and design tolerances.

Design Types

- Software design on any project may consist of many work products, which together can be termed the software design for the software product that will be built during the software project.
- Some examples include prototypes, structural models, object-oriented design, systems analysis and entity relationship models.



Software Design

Design Types

- A good software design not only ensures a smooth transition to the development phase but also ensures that the software product has a good shelf life during operation.
- So what are the keys to a good design? A good design should start from the most possible abstract architecture of the software product often termed as "high level design".
- Subsequent transition of the abstract design should lead to platform-specific design often termed as "low level design".
- The platform-specific design or low level design will be in terms of a good database model and a good application model (Figure above - Software design techniques).
- Over the years, many software design techniques have evolved with the evolution of different programming paradigms.
- Starting with the early procedural programming paradigms, programming has evolved into present day "service-oriented architecture".
- Software design has kept the pace with these evolving paradigms, and thus it has also been evolving. So, we have early structural design paradigms to modern day SOA designs. Let us discuss some of these design techniques.

Software Design

Design Types – Prototypes

- Prototyping is cheap and fast.
- It also gets a buy in from customer at an early stage of the project.
- If not, a full prototype of the application, a partial prototype can contribute to win over the customer.
- There are many automatic code generation tools that allows to drag and drop some components on screens, and the tool generates the code and makes a working prototype of the application that can be demonstrated to the customer.
- A miscommunication or misunderstanding between the customer and the project team gets cleared once the difference of opinion are sorted out early on during the prototype demonstration sessions.
- This greatly helps in reducing the risks of not meeting customer expectations.
- In any case, customers do not care about internal workings of the application.
- They are always concerned about what the application screens look like and how the application behaves with different kind of inputs and events.

Software Design

Design Types – Prototypes

- The downside about prototypes is that many customers assume the prototype is the fully functional application and later on wonder why the application is taking so much time in development when they saw the working demonstration so early in the project.
- Customer expectations become difficult to manage in such instances.
- Prototypes can only show the user interface screens.
- When complex logic is involved in developing applications, that logic cannot be depicted in prototypes, as program logic is mostly not visible and cannot be developed in prototypes.

Software Design

Design Types – Design Reuse

- For large software products, the design can be broken into many design parts representing each module of the product.
- Each of these design modules contain a lot of design information that can be represented as design components.
- Many details inside these design components can be repeated inside different components.
- If a standard method of representing the same information can be used for these components, then it is possible to use these pieces of information in many components by reusing them.
- It will reduce effort in designing the product.
- This method of design reuse is known as internal design reuse.
- A more potent design reuse is becoming available after the advent of the open source paradigm and SOA.
- In the case of open source, the design reuse is in fact a case of copying existing design and then using it exactly as it is or modifying it to suit the needs.
- But in the case of SOA, there is no copying or modifying a software design.
- The existing design is utilized as it is.

Software Design

Design Types – Design Reuse

- In addition, there is no process of buying the application/component whose design is required.
- There is simply buying a service from the owner of the application/component and using that service in building the application.
- The owner of that application/component publishes full details as to how to integrate with the application for the according application/component.
- The full interface details are provided by the owner.
- Using this information, the design is done for the application.
- There is an assumption that as if the application/component provided as a service is available and the application uses this application/component.
- SOA is indeed leading to a reuse model that is going to transform the world of computing and the lives in years to come.

Software Design

Design Types – Structural Models

- Most software applications are built using components.
- At the bottom are the smallest units of functions and procedures in a software application.
- These functions are contained within classes or packages depending on the programming language used.
- Many classes together build a component.
- Components in turn make modules. Modules in turn make the complete application.
- For ease of working, maintenance, and breaking development tasks to allocate to group of developers, it is essential that an application is broken down into manageable parts.
- Breaking into parts for an application can best be done using a structural analysis.
- From requirement specifications, a feature set is made to decide what features will be in the application.
- This feature set is analyzed and broken down into smaller sets of features, which will go into different modules.
- This is represented in a structural model of the application.

Software Design

Design Types – Object-oriented Design

- It has always been difficult to represent business entities and business information flow in a software model.
- With object-oriented design, this problem was solved.
- Business entities are represented as objects in the object- oriented software design.
- Properties of these objects are made in such a way that they are similar to the properties of the business entities.
- These objects are instantiated from classes in the form of child classes.
- These child classes inherit all the properties of their parent class, and they can have some more properties of their own in addition.
- So if we have a group of similar objects with somewhat different properties, then we can implement classes in such a way that a base parent class has child classes with different properties.
- This concept aligns very much to the real-world scenarios.
- Object-oriented design takes input from use cases, activity diagrams, user interfaces etc.

Software Design

Design Types – Systems Analysis

- System analysis is the process of finding solutions in the form of a system designed from the inputs coming from business needs.
- The fundamental question addressed in system analysis is whether a business scenario can be converted into a software application, so that the user can use the software application to do his routine business tasks.
- For instance, a person may want to access his bank account using an Internet connection to the online web site of the bank.
- This scenario calls for many things that are involved in the whole chain of objects and events.
- The system analysis will be concerned with **user activities, what objects on the web site act with user activities, how these objects interact with the underlying software system of the bank, and how connections are made between the user and the website and between the website and the bank system.**
- System analysis will analyze all these things.
- Based on the analysis, a system model can be made that will be used in developing the application.

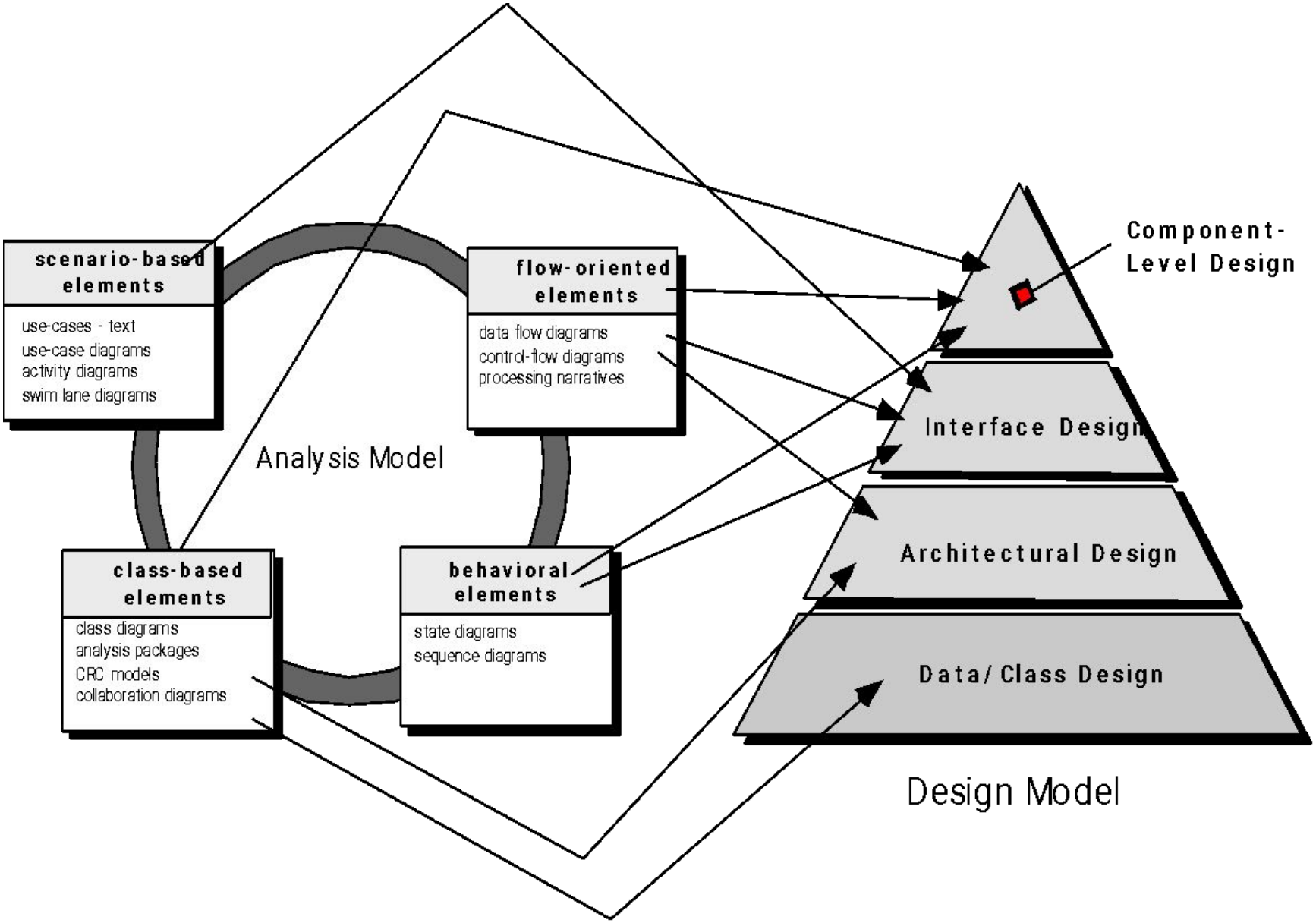
Software Design

Design Types – Entity Relationship Models

- Entity relationship models are one of the ways to represent business entities and their relationships to each other through diagrams.
- These diagrams are used for creating databases and database tables.
- How many tables are needed to fulfill the needs of the software product, how these tables are related to each other, and in what form data are to be kept inside these tables, etc. are decided through these diagrams.
- With object-oriented modeling, it is possible to correlate each object with a corresponding database object.
- This kind of representation helps to make a clean database design.

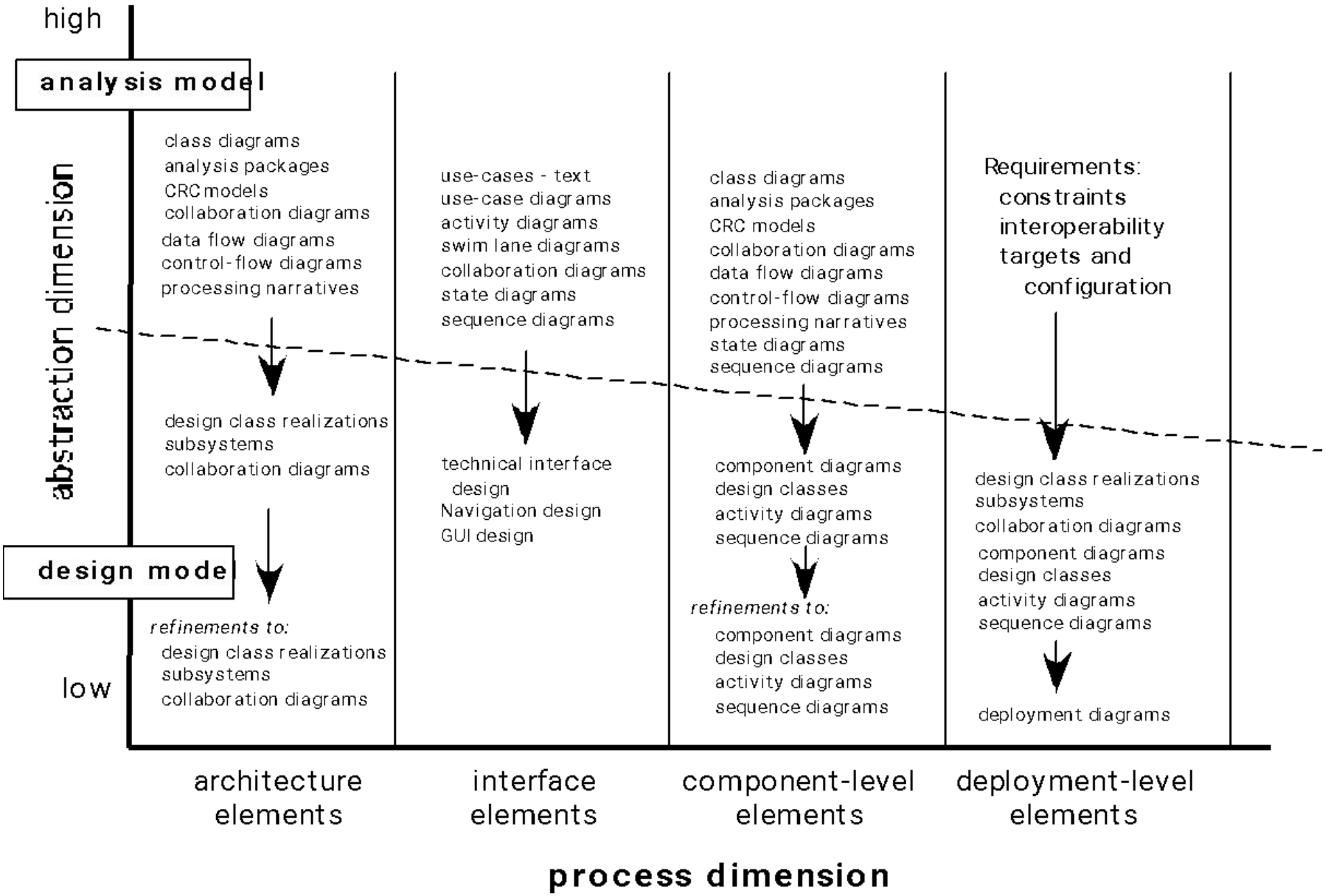
Design Model

Analysis Model -> Design Model



Design Model

Analysis Model -> Design Model



Architectural Design

Architectural Design

- The overall structure of the software and the ways in which that structure provides conceptual integrity for a system.

Structural properties

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

Extra-functional properties

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

Families of related systems

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

Software Architecture

Why Architecture?

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

  (1) Analyze 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.

Architecture - Significance

- 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 mode of how the system is structured and how its components work together”.

Software Architecture

Architectural Descriptions

- The IEEE Computer Society has proposed IEEE-Std-1471-2000, Recommended Practice for Architectural Description of Software-Intensive System,
  - To establish a conceptual framework and vocabulary for use during the design of software architecture,
  - To provide detailed guidelines for representing an architectural description, and
  - To encourage sound architectural design practices.
- The IEEE Standard defines an architectural description (AD) as a "**a collection of products to document an architecture**."
  - The description itself is represented using multiple views, where each view is "a representation of a whole system from the perspective of a related set of [stakeholder] concerns."

Architectural Design

Architectural Genres

- Genre implies a specific category within the overall software domain.
- Within each category, there are a number of subcategories.
  - For example, within the genre of buildings, there are following general styles: houses, condos, apartment buildings, office buildings, industrial building, warehouses, and so on.
  - Within each general style, more specific styles might apply.
  - Each style would have a structure that can be described using a set of predictable patterns.

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,

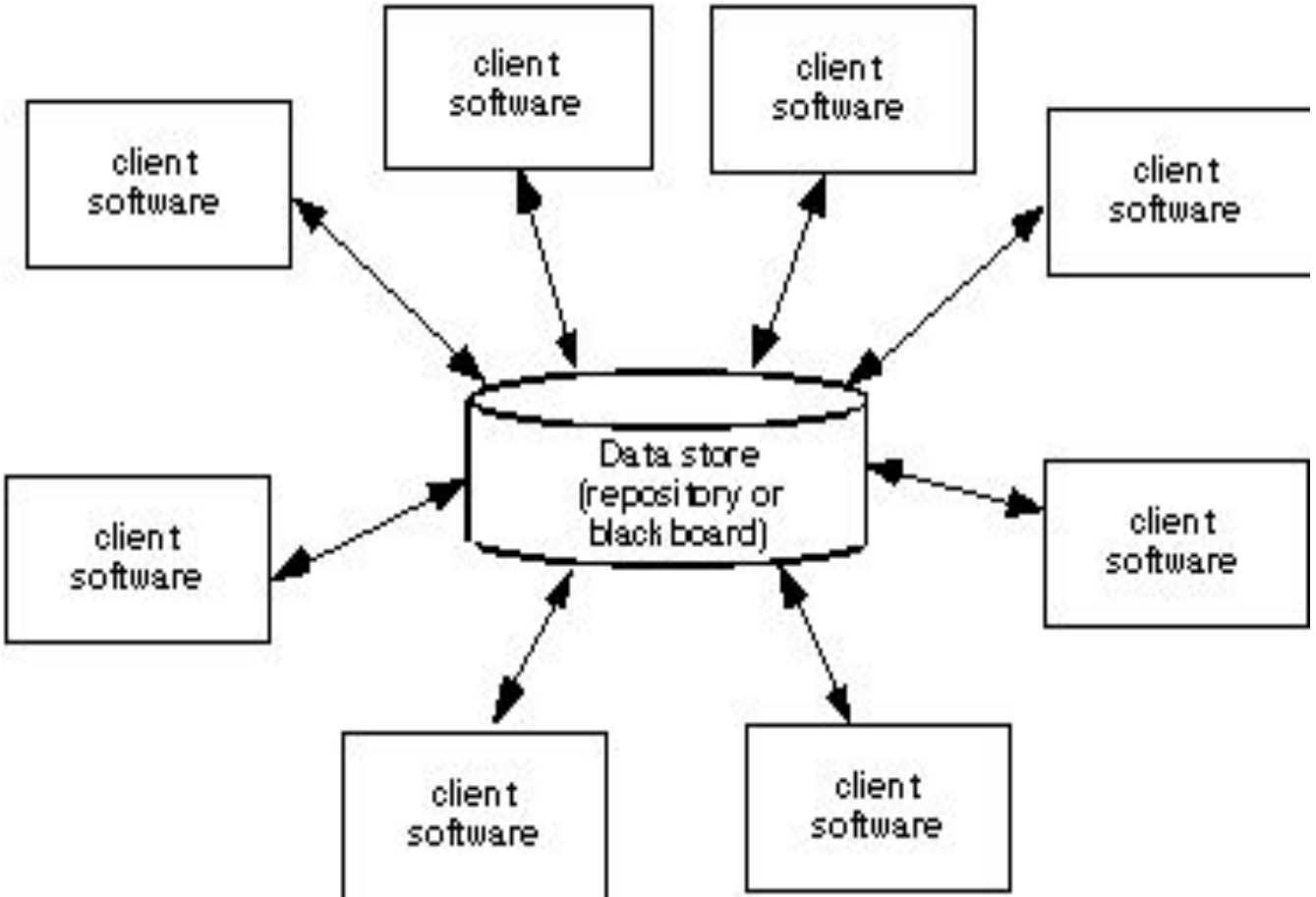
  (3) constraints that define how components can be integrated to form the system, and

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

Architectural Design

Architectural Styles – Data Centered Architecture

- 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.
- Figure below illustrates a typical data-centered style.



Architectural Design

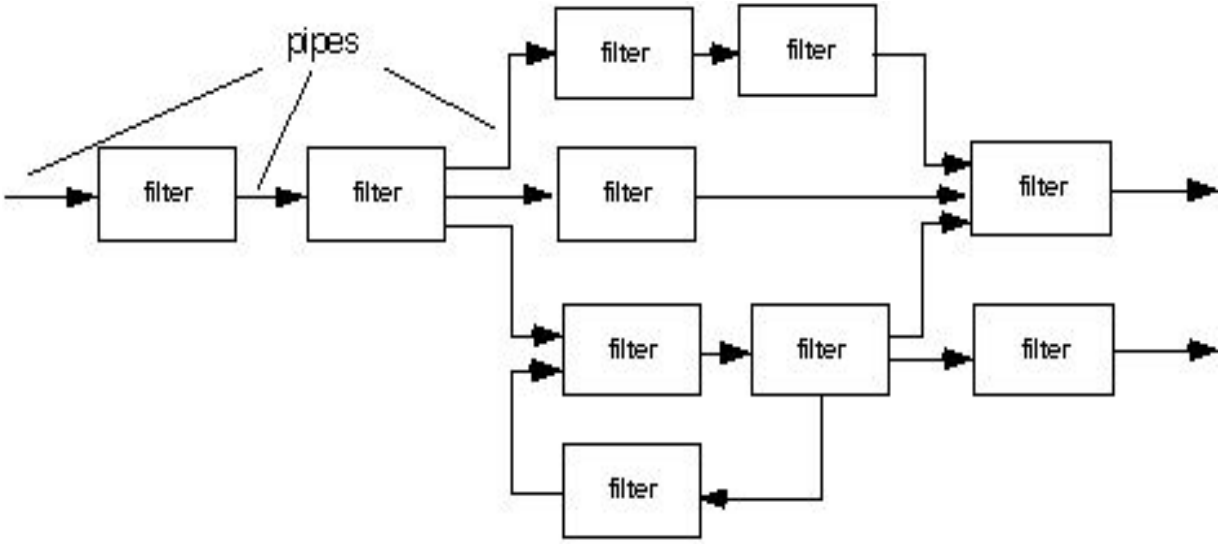
Architectural Styles – Data Centered Architecture

- 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.

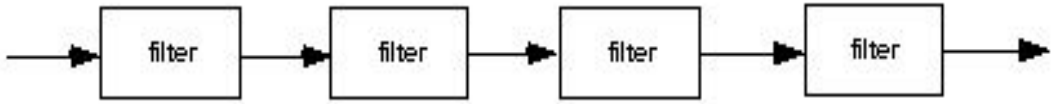
Architectural Design

Architectural Styles – Data Flow Architecture

- 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.



(a) pipes and filters



(b) batch sequential

Architectural Design

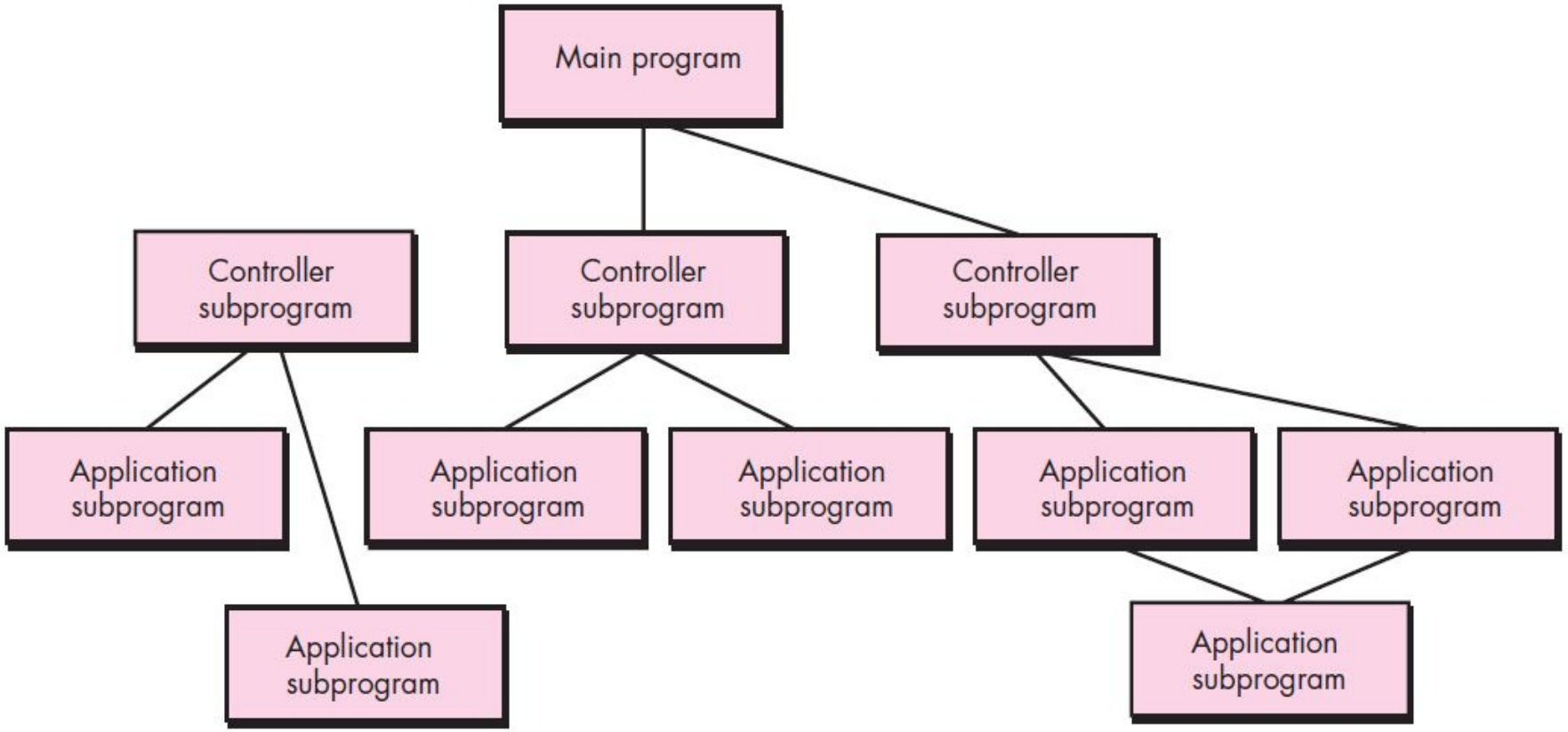
Architectural Styles – Data Flow Architecture

- Each filter works independently of those components upstream and downstream, is designed to expect data input of a certain form, and produces data output (to the next filter) of a specified form.
- However, the filter does not require knowledge of the workings of its neighboring filters.
- If the data flow degenerates into a single line of transforms, it is termed batch sequential.
- This structure accepts a batch of data and then applies a series of sequential components (filters) to transform it.

Architectural Design

Architectural Styles – Call and Return Architecture

- This architectural style enables you to achieve a program structure that is relatively easy to modify and scale.
- A number of sub-styles exist within this category:
- Main program/subprogram architectures: This classic program structure decomposes function into a control hierarchy where a "main program invokes a number of program components that in turn may invoke still other components.



Architectural Design

Architectural Styles – Call and Return Architecture

- The figure above 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.

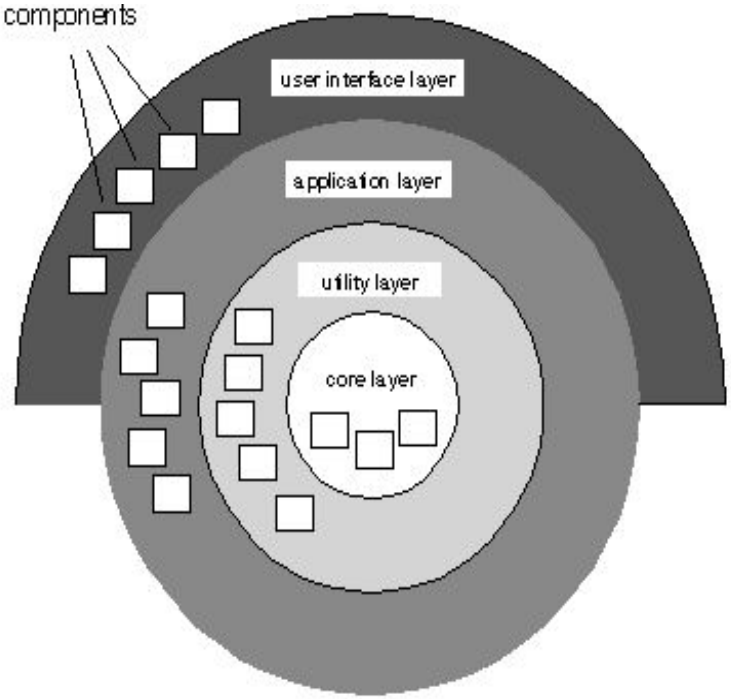
Architectural Styles – Object-oriented Architecture

- 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.

Architectural Design 52,54,57,58,59,60,88,91,92,

Architectural Styles – Layered Architecture

- The basic structure of a layered architecture is illustrated below.
- A number of different layers are defined, each accomplishing operations that progressively become closer to the machine instruction set.
- At the outer layer, components service user interface operations.
- At the inner layer, components perform operating system interfacing.
- Intermediate layers provide utility services and application software functions.
- These architectural styles are only a small subset of those available.
- Once requirements engineering uncovers the characteristics and constraints of the system to be built, the architectural style and/or combination of patterns that best fits those characteristics and constraints can be chosen.



Architectural Design

Architectural Patterns

- Concurrency—applications must handle multiple tasks in a manner that simulates parallelism
  - Operating system process management pattern
  - Task scheduler pattern

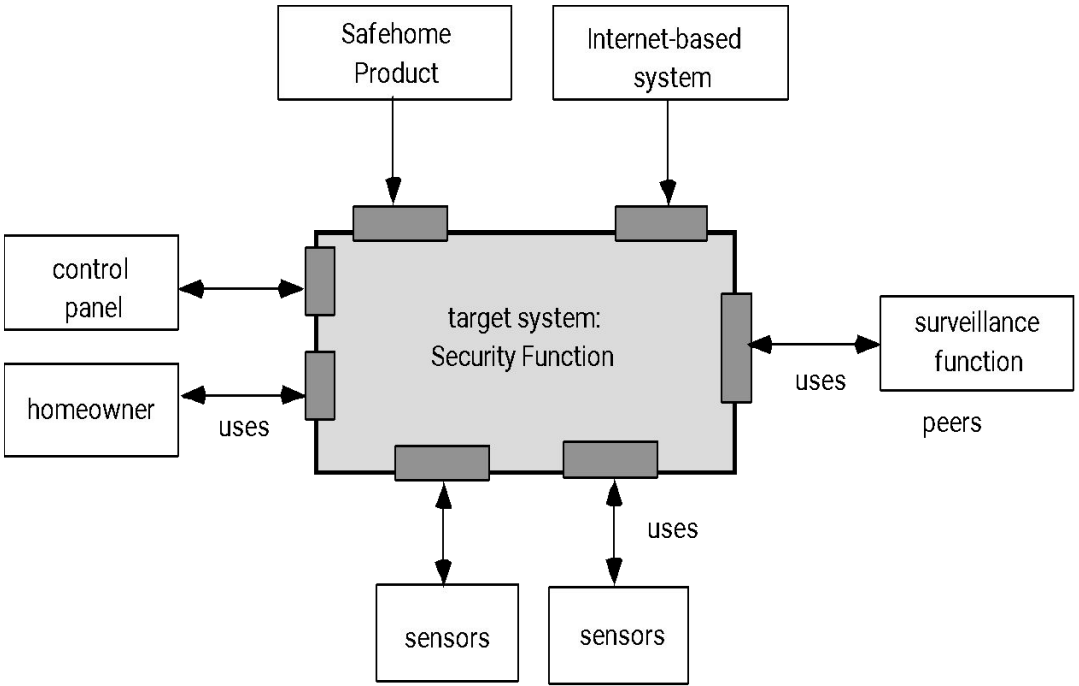
- Persistence—Data persists if it survives past the execution of the process that created it. Two patterns are common:
  - A database management system pattern that applies the storage and retrieval capability of a DBMS to the application architecture.
  - An application level persistence pattern that 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

Architectural Context

- The software must be placed into context
  - The design should define the external entities (other systems, devices, people) that the software interacts with and the nature of the interaction.
- A set of architectural archetypes should be identified.
- The designer specifies the structure of the system by defining and refining software components that implements each archetype.
- The below diagram illustrates Architectural context diagram for the SafeHome security function.



Architectural Design

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 are required to design even relatively complex systems.
- In relation to the SafeHome home security function, it is necessary to define the following archetypes:
- Node – represents a cohesive collection of input and output elements of the home security function.
- Detector – An abstraction that encompasses all sensing equipment that feeds information into the target system.
- The diagram illustrates the UML relationships for SafeHome security function archetypes.

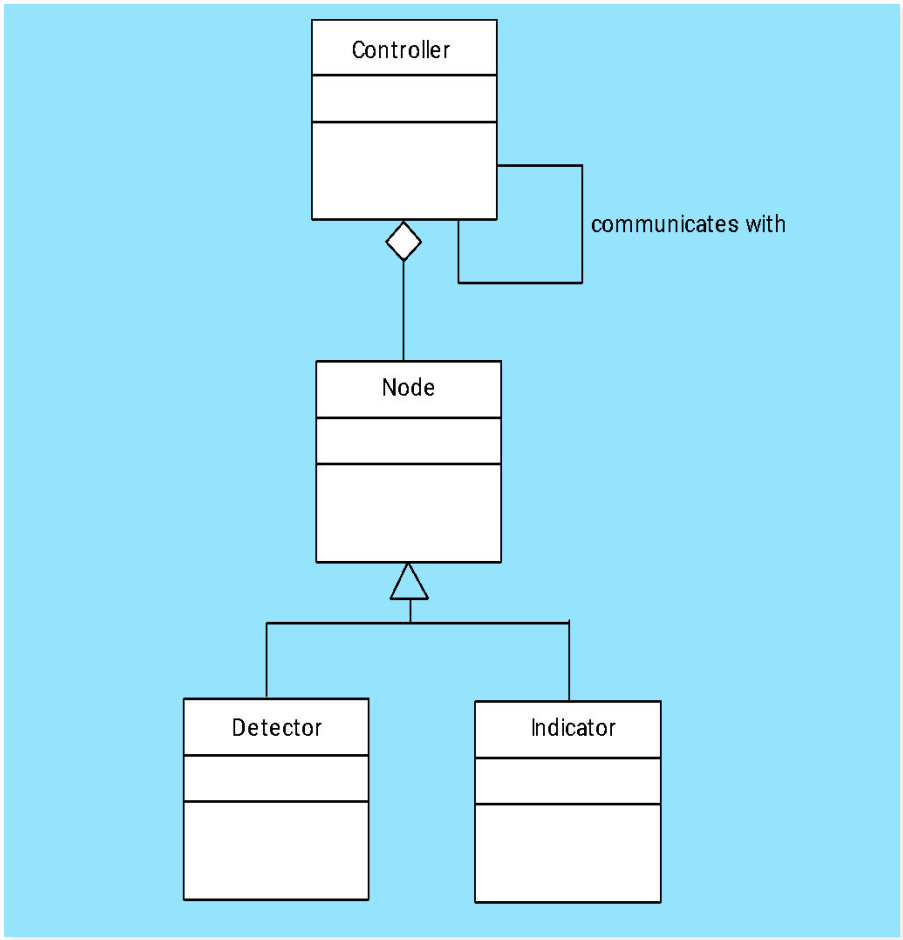


Figure 10.7 UML relationships for SafeHome security function archetypes (adapted from [BOS00])

Architectural Design

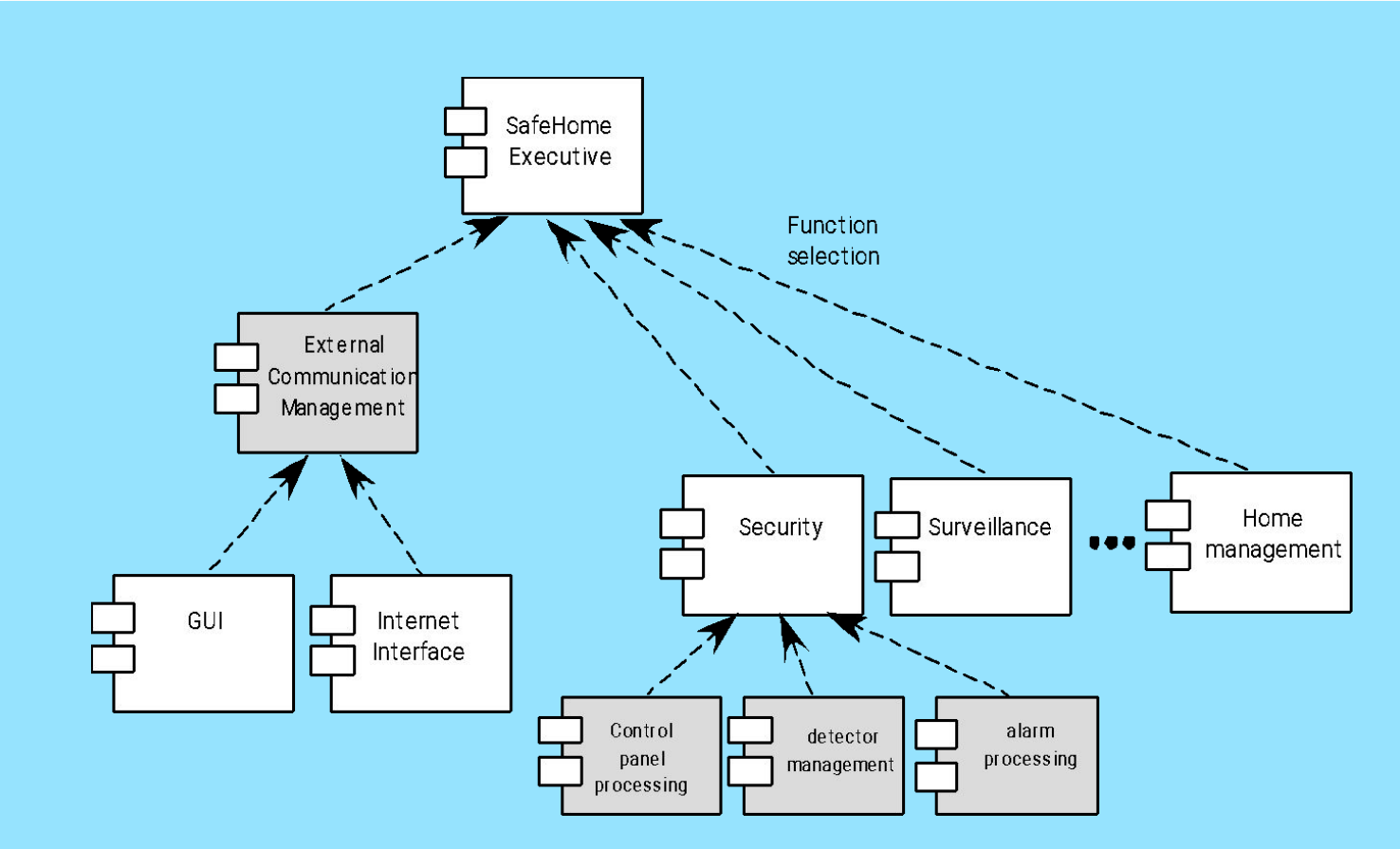
Archetypes

- Indicator – An abstraction that represents all mechanisms (e.g., alarm siren, flashing lights, 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.
- Each of these archetypes is depicted using UML notation as shown in above figure.
- Recall that the archetypes form the basis for the architecture but are abstractions that must be further refined as architectural design proceeds.
- For example, Detector might be refined into a class hierarchy of sensors.

Architectural Design

Component Structure

- As the software architecture is refined into components, the structure of the system begins to emerge.
- But how are these components chosen? In order to answer this question, it is necessary to begin with the classes that were described as part of the requirements model.



Architectural Design

Component Structure

- The figure above illustrates the overall architectural structure for SafeHome with top-level components.
- These analysis classes represent entities within the application (business) domain that must be addressed within the software architecture.
- Hence, the application domain is one source for the derivation and refinement of components.
- Another source is the infrastructure domain. The architecture must accommodate many infrastructure components that enable application components but have no business connection to the application domain.
- In regards to the SafeHome home security function example, the set of top-level components might be defined to address the following functionality:

  1. External communication management – coordinates communication of the security function with external entities such as other Internet-based systems and external alarm notification.

  2. Control panel processing – manages all control panel functionality.

  3. Detector management – coordinates access to all detectors attached to the system.

  4. Alarm processing – verifies and acts on all alarm conditions.

Architectural Design

Analyzing Architectural Design

1. Collect scenarios.
2. Elicit requirements, constraints, and environment description.
3. Describe the architectural styles/patterns that have been chosen to address the scenarios and requirements:
   - Module view
   - Process view
   - Data flow view
4. Evaluate quality attributes by considering each attribute in isolation.
5. Identify the sensitivity of quality attributes to various architectural attributes for a specific architectural style.
6. Critique candidate architectures (developed in step 3) using the sensitivity analysis conducted in step 5.

Architectural Design

Architectural Complexity

- The overall complexity of a proposed architecture is assessed by considering the dependencies between components within the architecture.
  - **Sharing dependencies** represent dependence relationships among consumers who use the same resource or producers who produce for the same consumers.
  - **Flow dependencies** represent dependence relationships between producers and consumers of resources.
  - **Constrained dependencies** represent constraints on the relative flow of control among a set of activities.

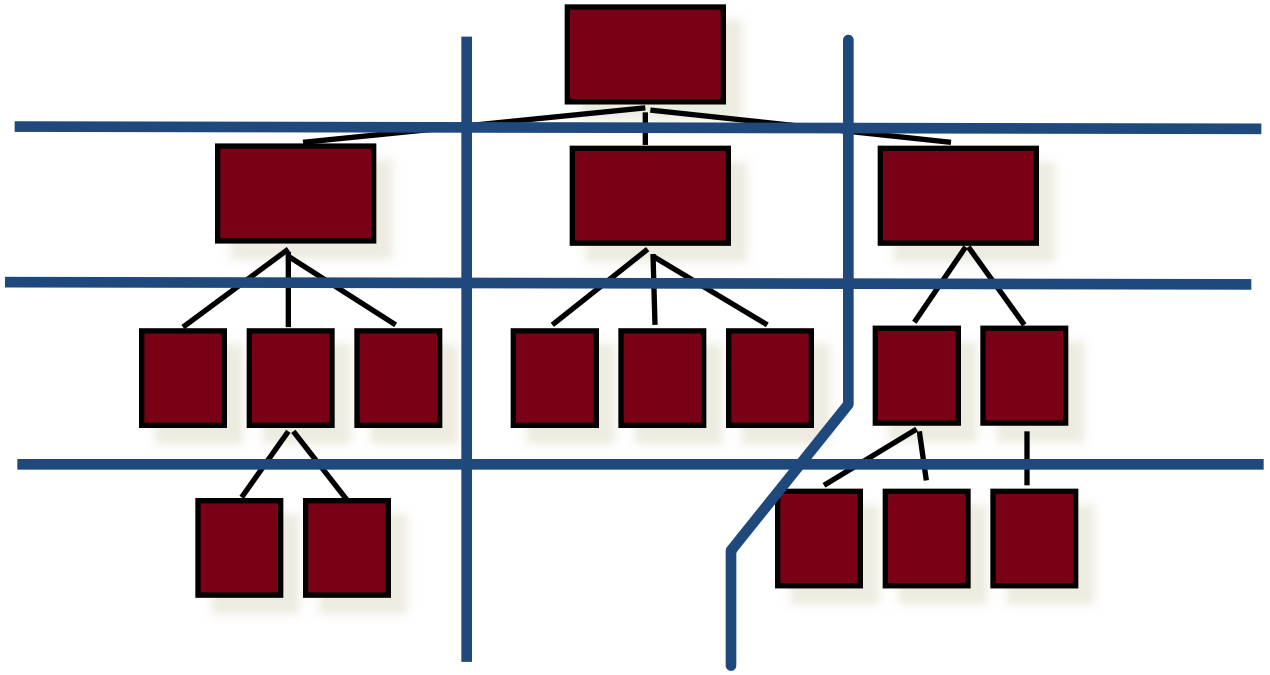
ADL

- Architectural description language (ADL) provides a semantics and syntax for describing a software architecture
- Provide the designer with the ability to:
  - Decompose architectural components
  - Compose individual components into larger architectural blocks and
  - Represent interfaces (connection mechanisms) between components.

Architectural Design

Partitioning the Architecture

- The “horizontal” and “vertical” partitioning are required



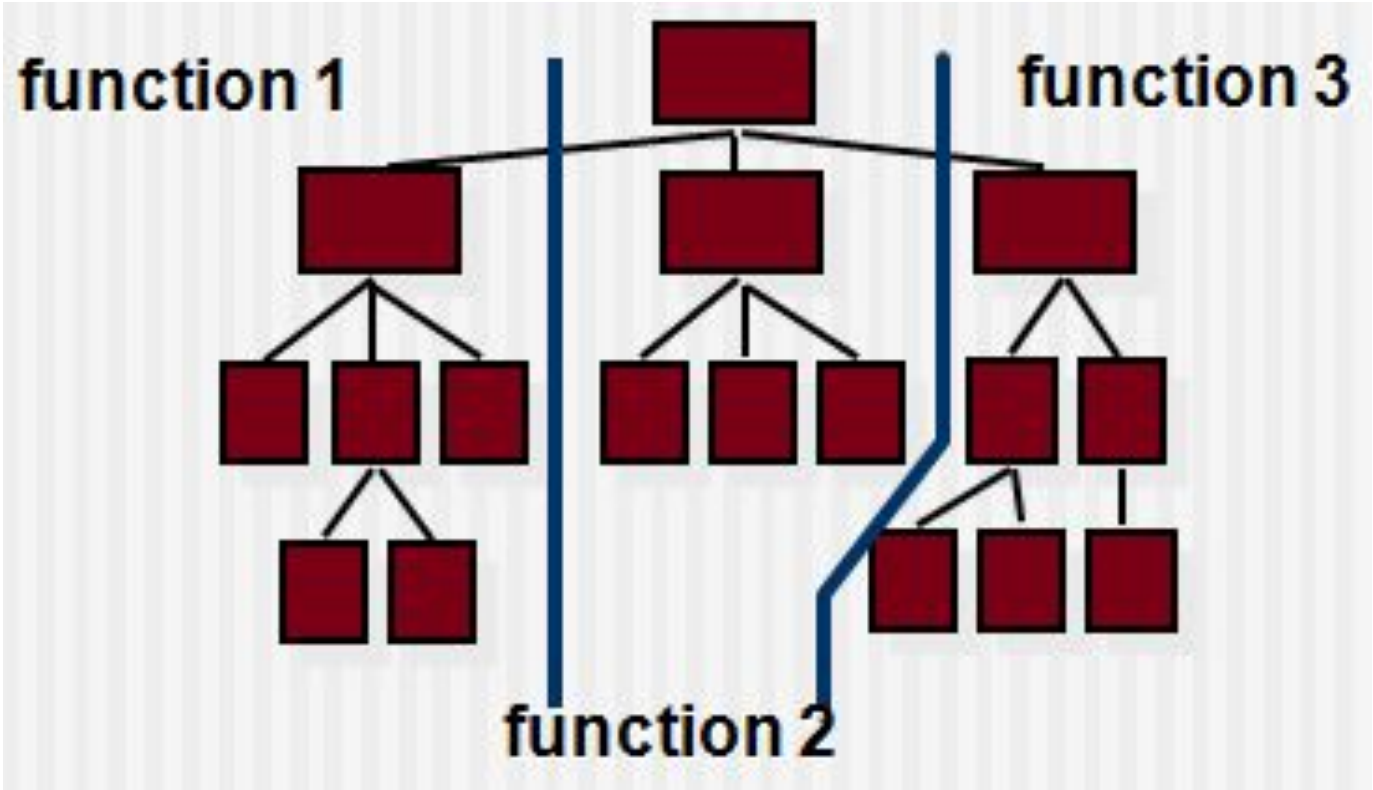
Why Partitioned Architecture?

- Results in software that are easier to test
- Leads to software that are easier to maintain
- Results in propagation of fewer side effects
- Results in software that are easier to extend

Architectural Design

Horizontal Partitioning

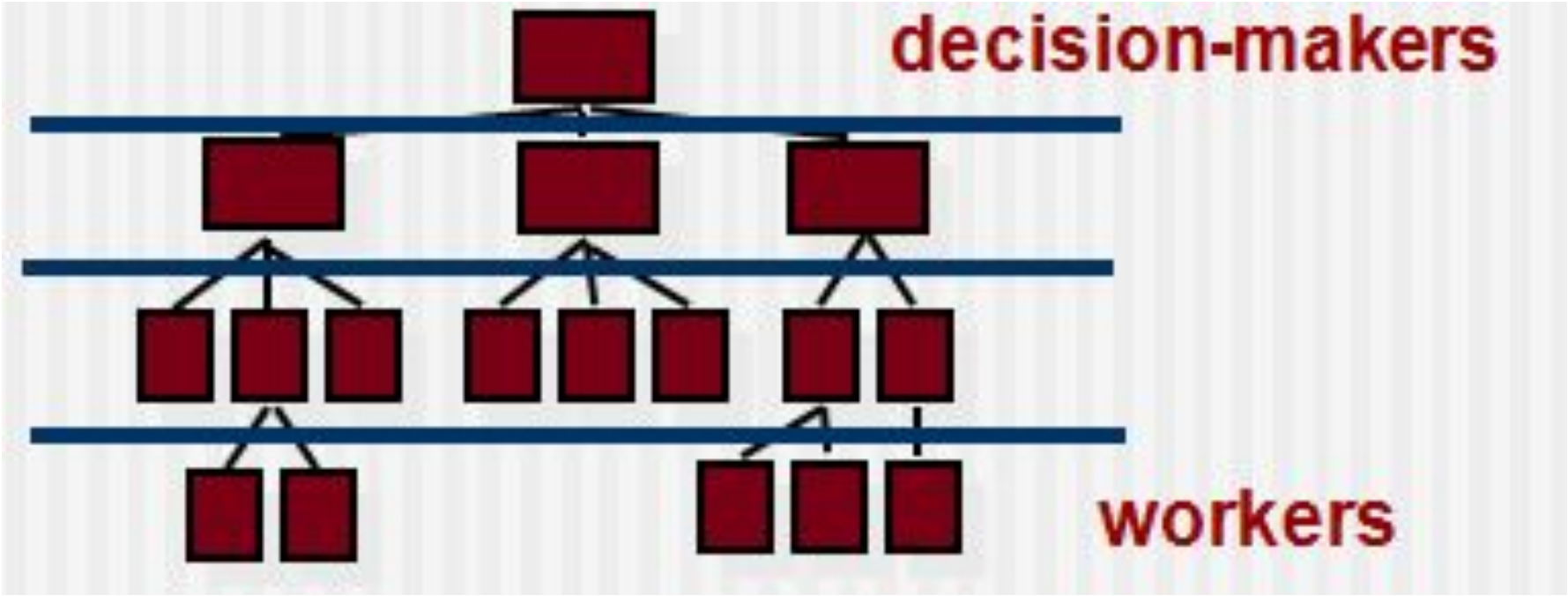
- Define separate branches of the module hierarchy for each major function
- Use control modules to coordinate communication between functions



Architectural Design (5,10,11,12,17,21,22,23,24,25,29,32,34,38,39,43,45,48,49,52,53,54

Vertical Partitioning – Factoring

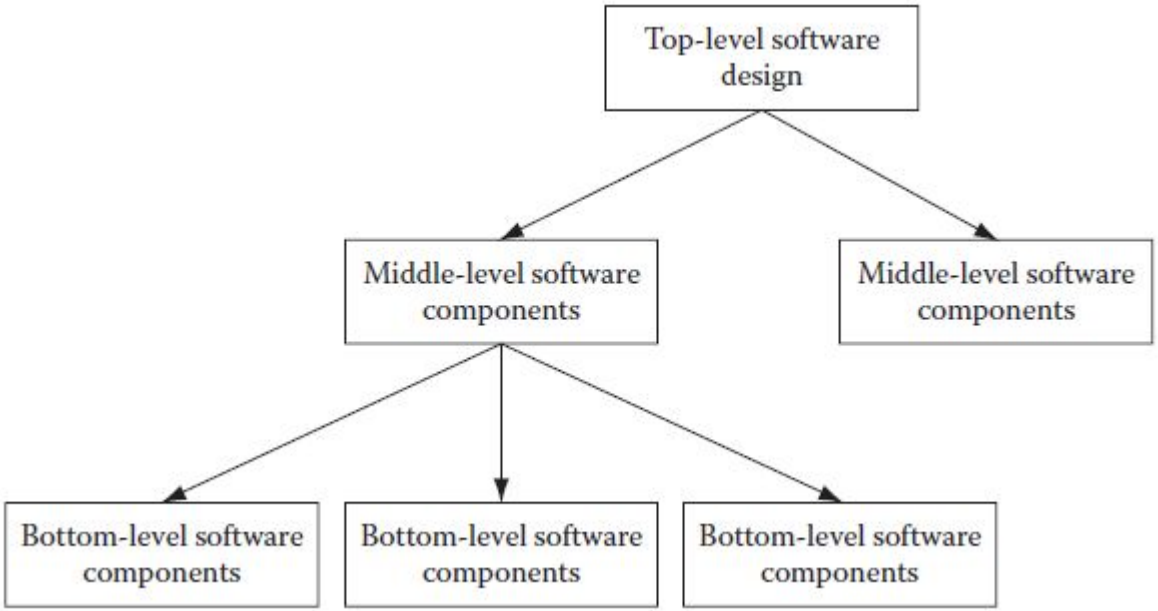
- Design so that decision making and work are stratified
- Decision making modules should reside at the top of the architecture



Software Design Methods

Top Down

- In the top-down approach, the top structure of the product is conceived and designed first.
- Once the structure is perfected, components that will make the product are designed.
- Once the major components are designed, the features that make the component are designed (Figure below – Top-down Software Design).
- Apart from the functional consideration for making the structure, nonfunctional considerations are also considered from the top level for example, how the security, performance, usability, aspects will be provided in the product.



Software Design Methods

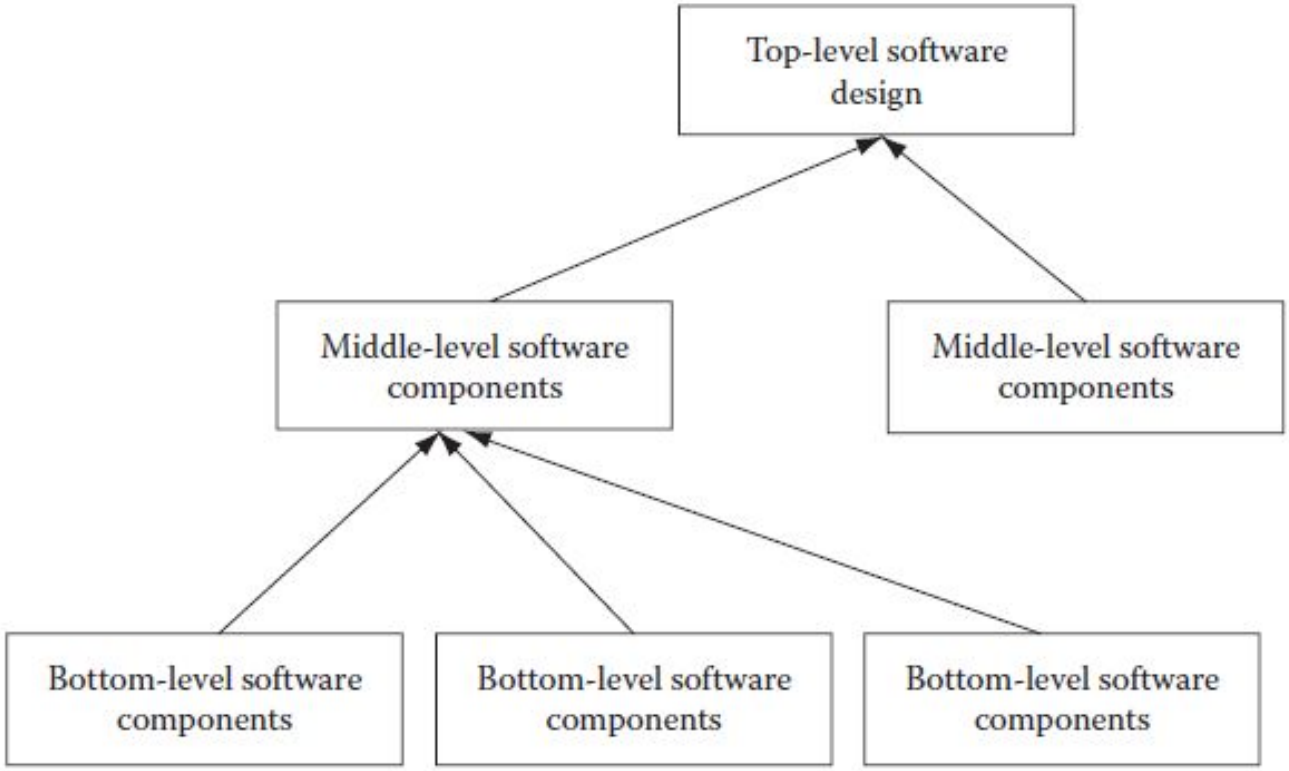
Top Down

- There are many benefits to the top -down approach.
- Nonfunctional aspects are taken care of at the beginning of design, and hence they are an integral part of the product and not an after-thought.
- This makes a secure, robust, and usable product.
- A top- down approach also helps in creating reusable components and hence increases productivity as well as maintainability.
- This approach also promotes integrity, as the whole product is designed inside a single framework.
- So a fragmented and dissimilar approach for designing different parts of the product is avoided.
- The drawback of the top -down approach is that it is a risky model.
- The whole design has to be made in one go instead of making attempts to incrementally building the design, which is relatively a safer option.
- Generally, the top-down design approach is adopted on waterfall model-based projects.

Software Design Methods

Bottom Up

- In the bottom-up approach, first, the minute functions of the software product are structured and designed.
- Then, the middle-level components are designed, and, finally, the top-level structure is designed.
- Once some components are designed, they can be shown to the customer, and a buy in can be made for the project.



Software Design Methods

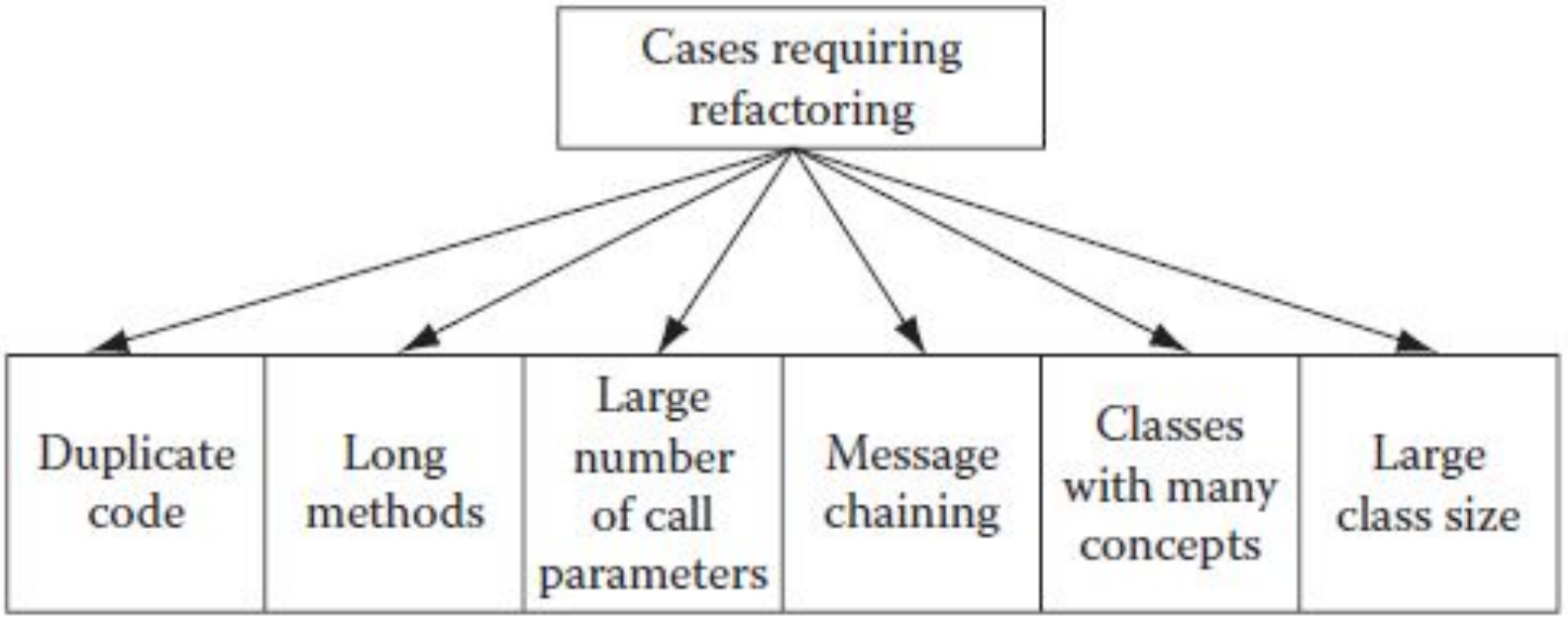
Bottom Up

- There are some benefits to the bottom-up approach.
- It leads to incremental building of design that ensures that any missing information can be accommodated later in the design (Figure above – Bottom-up Software Design).
- With increasing use of incremental and iterative development methodologies, the bottom-up design approach is becoming more popular than the top-down approach.
- In fact, nowadays, agile models do not go for elaborate and complete software design from the beginning of the project.
- In each iteration, a design is thought of for the requirements that are taken during the iteration.
- To compensate for a sturdy and elaborate design upfront, the project team engages in refactoring (discussed next) the design to make sure that it does not become bulgy and unmanageable in later iterations.

Module Division (Refactoring)

- Whenever a software product is designed, it is done with good intentions.
- Care is taken to ensure that the design is extensible, so that when customer needs increase over time, the product can be extended to take care of those increased needs.
- Unfortunately, even this foresight is not enough, and it becomes difficult to extend the product functionality further. In such cases, it becomes necessary to change the internal structure of software code without changing external behavior of the software product.

Figure - Characteristics of a software product code that requires refactoring



Module Division (Refactoring)

- To do this, one technique is employed, which is known as refactoring.
- Using refactoring, the internal design of a piece of software code is improved by decreasing coupling among classes of objects and increasing cohesion among classes.
- Refactoring is very similar to the concept of normalization in relational databases.
- Some of the indications of code analysis that may suggest that the code needs refactoring include duplicate code at many places, using long methods, a large class with many concepts, the need to pass a large number of parameters, too much communication between classes resulting from a large number of calls for methods in code, and message chaining by calling one method which in turn calls another method.
- When software code starts having these characteristics, then it is better to go for code cleaning or refactoring.
- Going for refactoring will be justified by savings in time due to better code reuse and make it easier to maintain code and scale up the product.
- Refactoring can be achieved by dividing cumbersome classes into smaller classes that can be managed and used in a better way.

Module Division (Refactoring)

- In the new code, the functions will be the same, but many of the functions will be moved now into new classes.
- On agile projects, the project team builds the software product without making an elaborate design from start.
- One product module is built after another in the subsequent project iterations.
- This fact makes it necessary to adjust the software design as the product evolves in this fashion.
- The adjustment in the software design in such cases is done using refactoring.

Module Coupling

- One area similar to refactoring is coupling between modules.
- As products mature and more and more lines of code are added to the existing product, coupling between modules tends to increase.
- This has a profound impact when any changes in code are required.
- Changes in code result in more than normal occurrence of defects as dependency between modules keeps increasing with increase in the size of the product.
- To reduce the chances of product defects, it is necessary to reduce the number of calls among different modules and classes.
- Service Oriented Architecture (SOA) architecture provides great help here.
- SOA architecture essentially promotes loose coupling, and this implies more or less self-contained classes having less dependency on other classes.
- Increasing module coupling with increase in size of software product is always a concern.
- Frequent refactoring can help in reducing module coupling among classes.

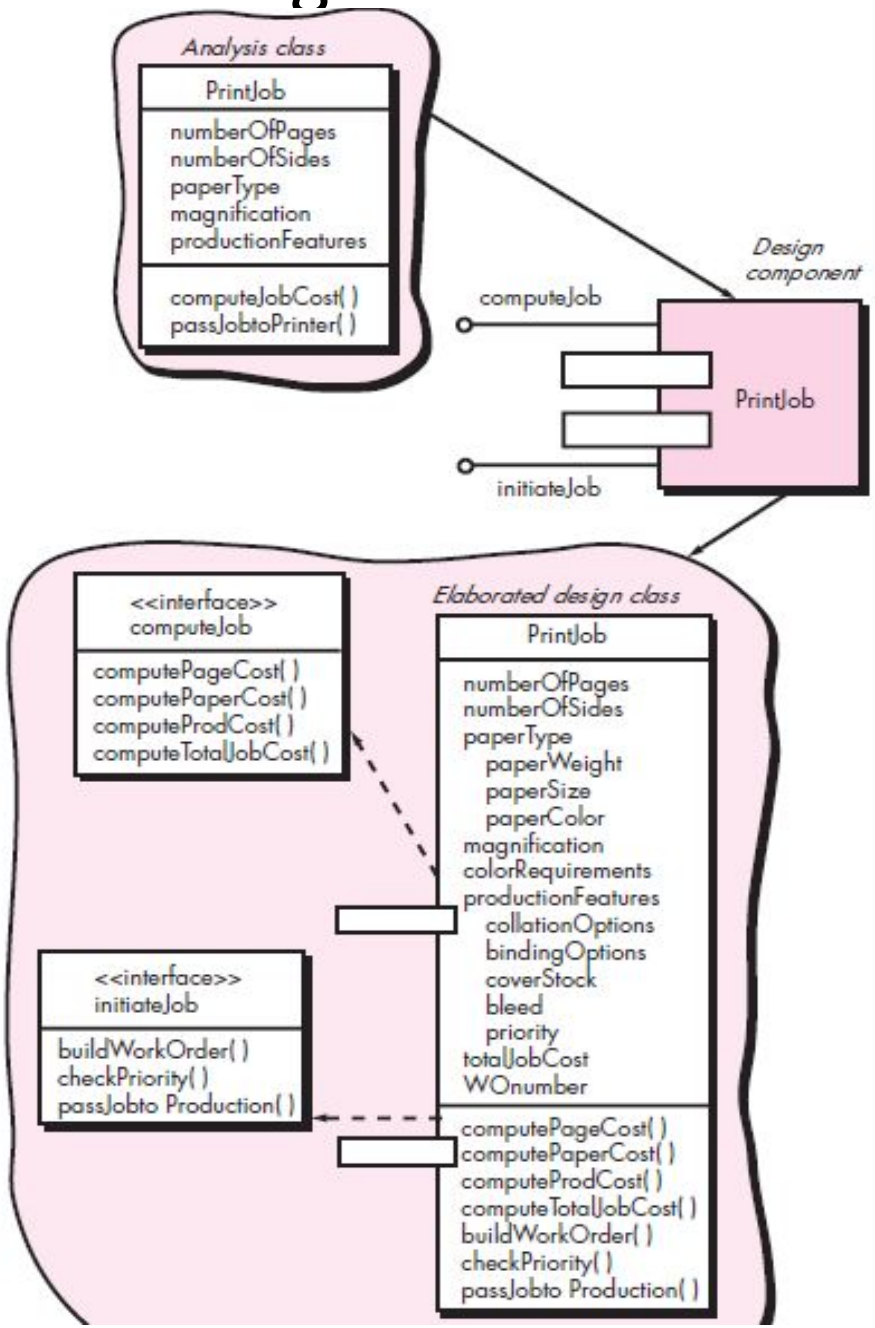
Component Level Design

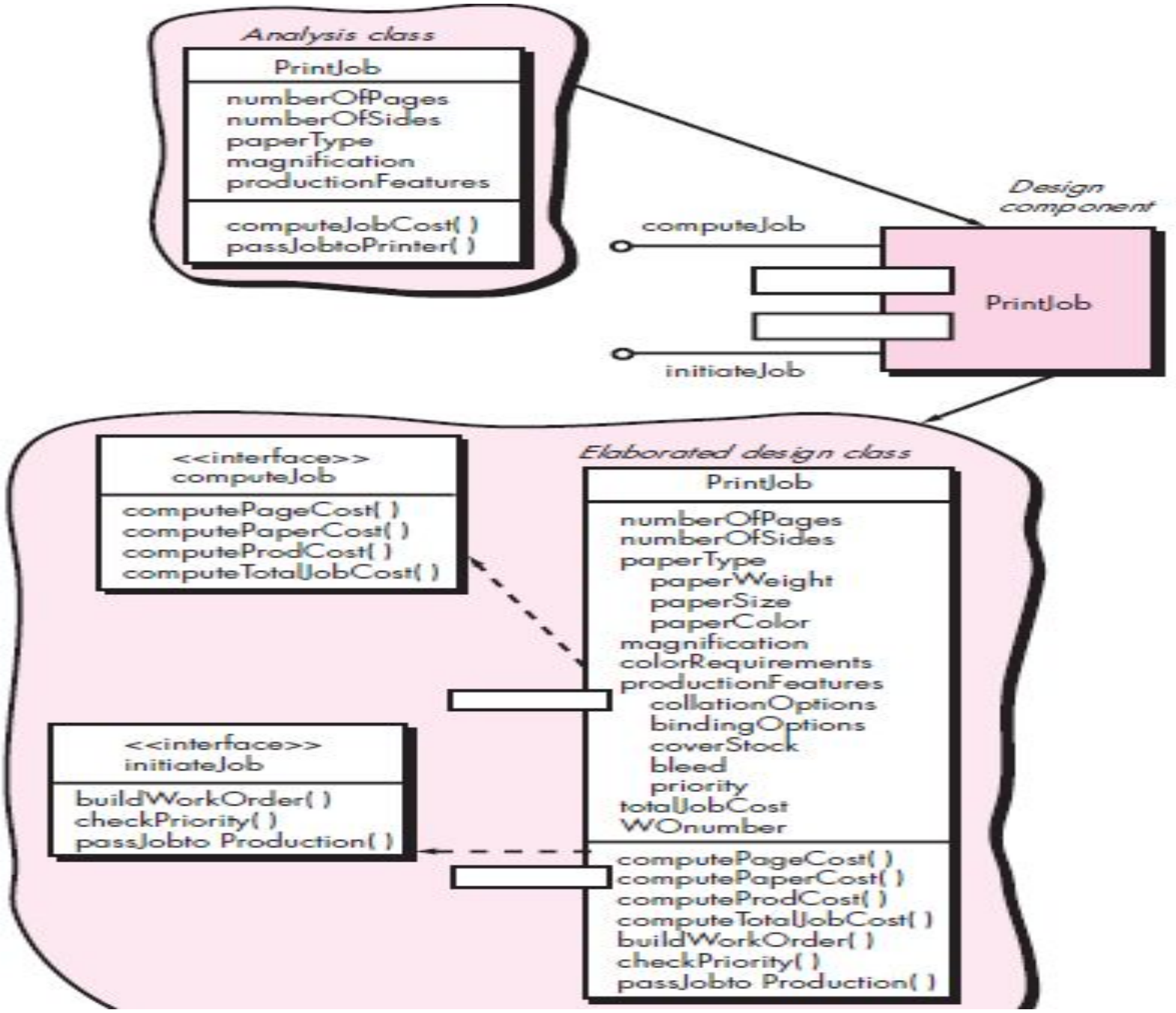
- OMG (Object Management Group) Unified Modeling Language Specification defines a component as “A modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces.”
- **OO view**: A component contains a set of collaborating classes
- **Conventional view**: A component contains 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.

Component Level Design

Object Oriented Component

- 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.
- To illustrate this process of design elaboration, consider software to be built for a sophisticated print shop.
- The overall intent of the software is tc collect the customer's requirements a the front counter, cost a print job, anc then pass the job on to an automated production facility.
- During requirements engineering, an analysis class called PrintJob was deriv




Analysis class
PrintJob
numberOfPages
numberOfSides
paperType
magnification
productionFeatures
computeJobCost( )
passJobtoPrinter( )
Design component
computeJob
PrintJob
initiateJob
<<interface>>
computeJob
computePageCost( )
computePaperCost( )
computeProdCost( )
computeTotalJobCost( )
Elaborated design class
PrintJob
numberOfPages
numberOfSides
paperType
paperWeight
paperSize
paperColor
magnification
colorRequirements
productionFeatures
collationOptions
bindingOptions
coverStock
bleed
priority
totalJobCost
WOnumber
computePageCost( )
computePaperCost( )
computeProdCost( )
computeTotalJobCost( )
buildWorkOrder( )
checkPriority( )
passJobto Production( )
<<interface>>
initiateJob
buildWorkOrder( )
checkPriority( )
passJobto Production( )

Component Level Design

Object Oriented Component

- The attributes and operations defined during analysis are noted at the top of figure given here.
- During architectural design, PrintJob is defined as a component within the software architecture and is represented using the shorthand UML notation shown in the middle right of the figure.
- Note that PrintJob has two interfaces, computejob which provides job costing capability and initiatejob, which passes the job along to the production facility.
- These are represented using the "lollipop" symbols shown to the left of the component box.
- Component level design begins at this point.
- The details of the component PrintJob must be elaborated to provide sufficient information to guide implementation.
- The original analysis class is elaborated to flush out all attributes and operations required to implement the class as the component PrintJob.
- Referring to the lower right portion of figure given, the elaborated design class PrintJob contains more detailed attribute information as well as an expanded description of operations required to implement the component.

Component Level Design

Conventional Component

- 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,

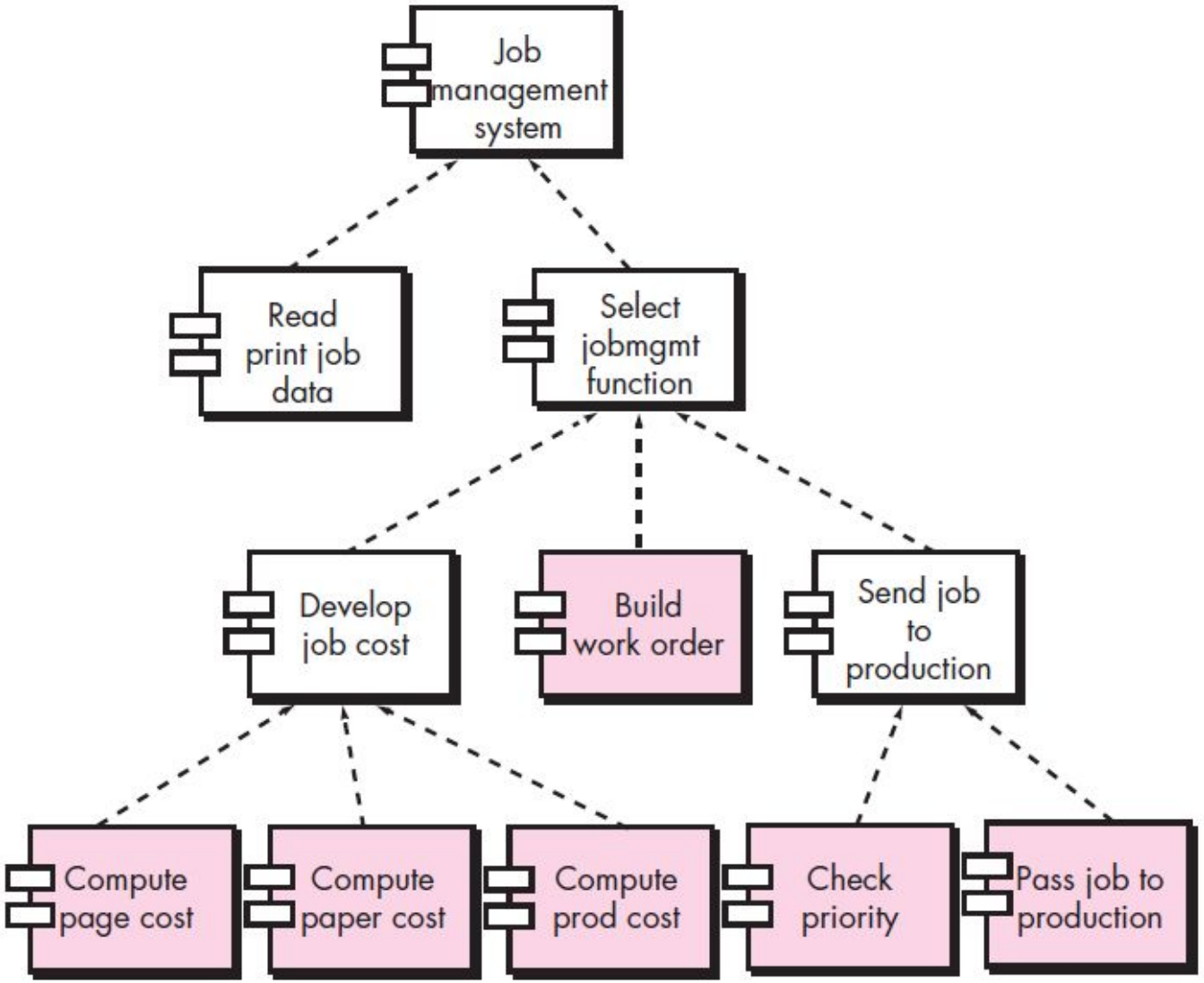
  (2) A **Problem domain component** that implements a complete or partial function that is required by the customer, or

  (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.
- In this case, however, the data flow-oriented element of the analysis model serves as the basis for the derivation.
- To illustrate this process of design elaboration for traditional components, again consider software to be built for a sophisticated print shop.
- A set of data flow diagrams would be derived during requirements modeling.

Component Level Design

Conventional Component

- Assume that these are mapped into an architecture shown in below figure.
- Each box represents a software component.



Component Level Design

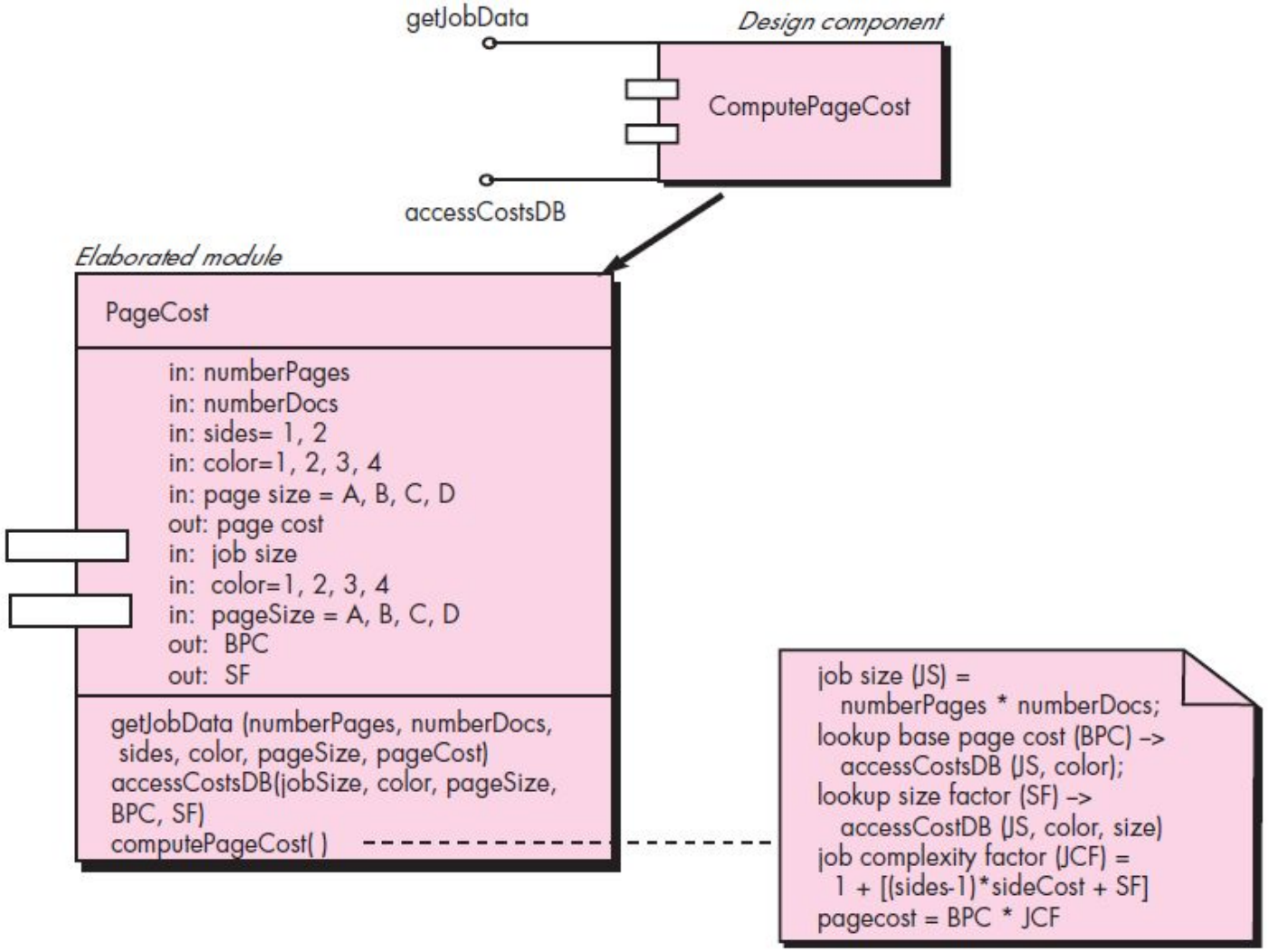
Conventional Component

- During component-level design, each module in above figure 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.
- To illustrate this process, consider the module *ComputePageCost.*
- The intent of this module is to compute the printing cost per page based on specifications provided by the customer.
- Data required to perform this function are: number of pages in the document, total number of documents to be produced, one- or two-side printing, color requirements, and size requirements.
- These data are passed to *ComputePageCost* via the module's interface.
- *ComputePageCost* uses these data to determine a page cost that is based on the size and complexity of the job—a function of all data passed to the module via the interface.

Component Level Design

Conventional Component

- Page cost is inversely proportional to the size of the job and directly proportional to the complexity of the job.



Component Level Design

Conventional Component

- The figure above represents the component-level design using a modified UML notation.
- The *ComputePageCost* module accesses data by invoking the module *getJobData,* which allows all relevant data to be passed to the component, and a database interface, *accessCostsDB,* which enables the module to access a database that contains all printing costs.
- As design continues, the *ComputePageCost* module is elaborated to provide algorithm detail and interface detail (Figure above).

Component Level Design

Basic Design Principles

- The Open-Closed Principle (OCP). "A module [component] should be open for extension but closed for modification".
- The Liskov Substitution Principle (LSP). "Subclasses should be substitutable for their base classes".
- Dependency Inversion Principle (DIP). "Depend on abstractions. Do not depend on concretions."
- The Interface Segregation Principle (ISP). "Many client-specific interfaces are better than one general purpose interface."
- The Release Reuse Equivalency Principle (REP). "The granule of reuse is the granule of release."
- The Common Closure Principle (CCP). "Classes that change together belong together."
- The Common Reuse Principle (CRP). "Classes that aren't reused together should not be grouped together."

Component Level Design

Design Guidelines

- 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 (as well as helping us to achieve the OCP-Open-Closed Princi).
- Dependencies and Inheritance
  - It is a good idea to model dependencies from left to right and inheritance from bottom (derived classes) to top (base classes).

Cohesion

- Conventional view:
  - The “single-mindedness” of a module
- OO view:
  - Cohesion implies that a component or class encapsulates only attributes and operations that are closely related to one another and to the class or component itself.

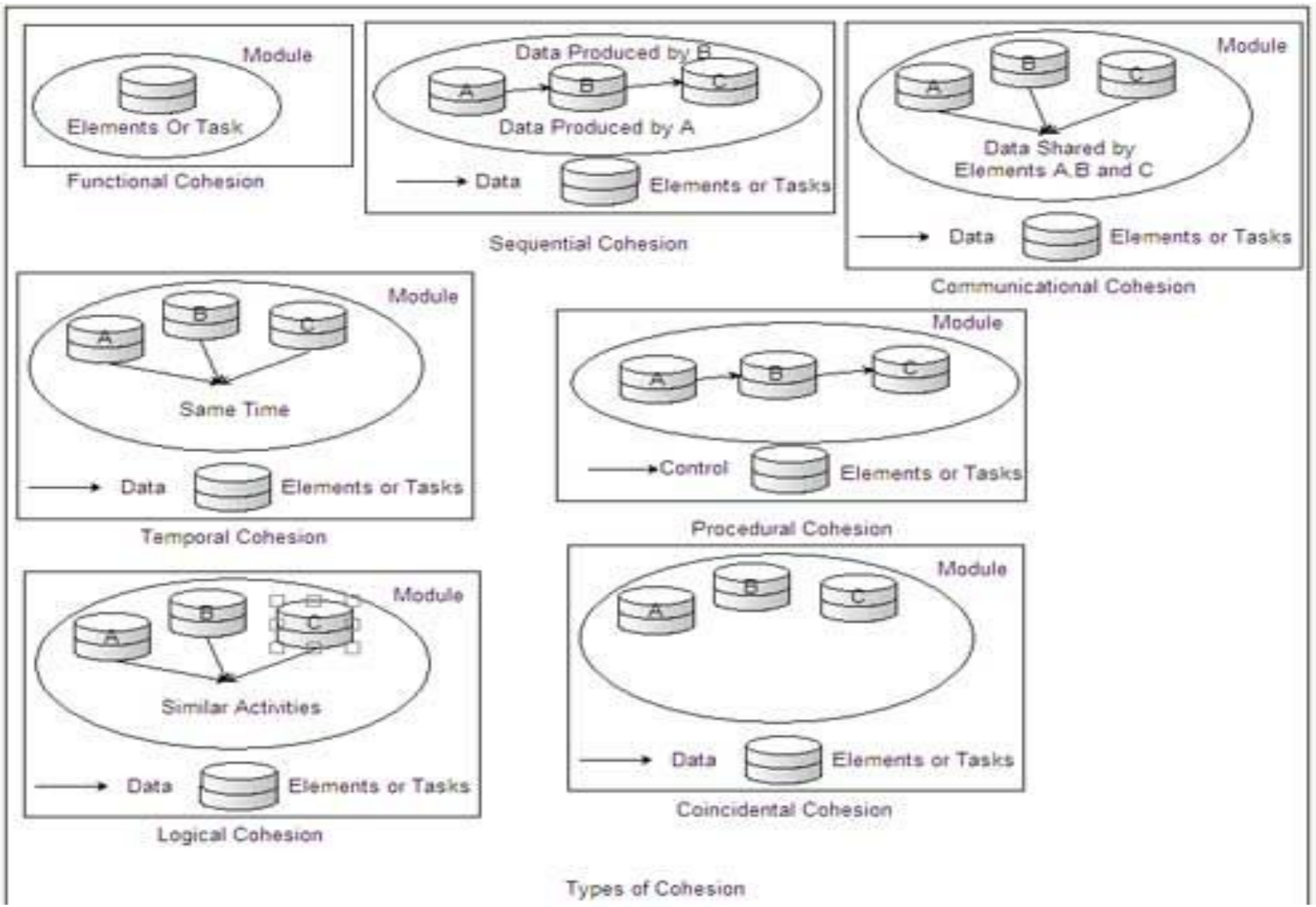
Component Level Design

Cohesion

- Levels of cohesion
  - Functional
  - Layer
  - Communicational
  - Sequential
  - Procedural
  - Temporal
  - Utility

Coupling

- Conventional view:
  - The degree to which a component is connected to other components and to the external world.
- OO view:
  - A qualitative measure of the degree to which classes are connected to one another.



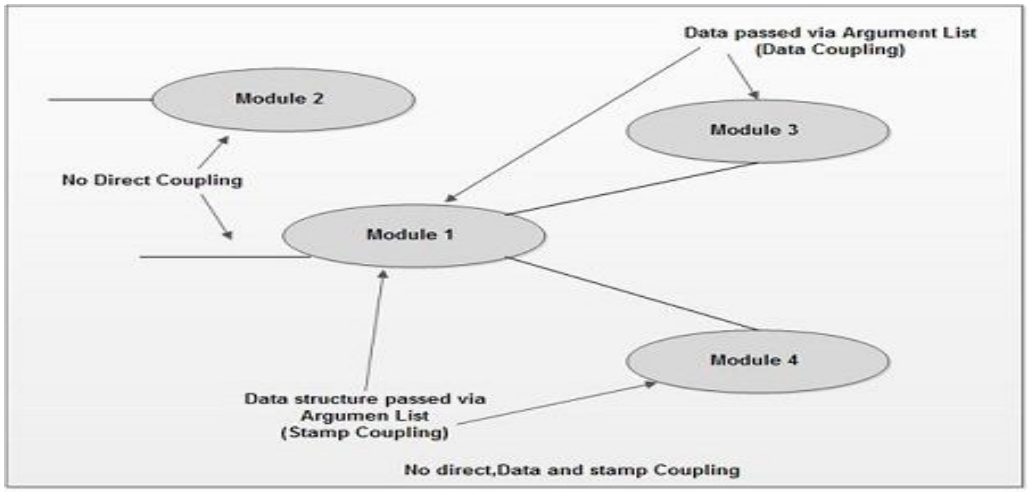
Types of Cohesion

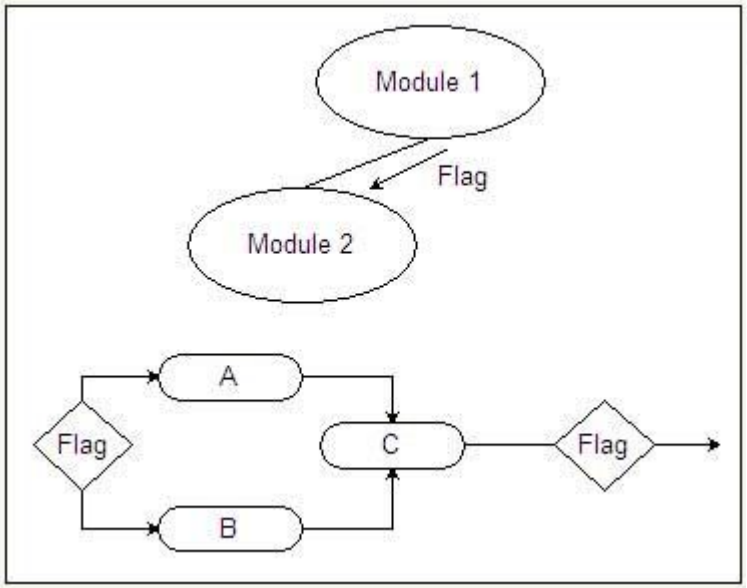
- **Functional cohesion:** In this, the elements within the modules are concerned with the execution of a single function.
- **Sequential cohesion:** In this, the elements within the modules are involved in activities in such a way that the output from one activity becomes the input for the next activity.
- **Communicational cohesion:** In this, the elements within the modules perform different functions, yet each function references the same input or output information.
- **Procedural cohesion:** In this, the elements within the modules are involved in different and possibly unrelated activities.
- **Temporal cohesion:** In this, the elements within the modules contain unrelated activities that can be carried out at the same time.
- **Logical cohesion:** In this, the elements within the modules perform similar activities, which are executed from outside the module.
- **Coincidental cohesion:** In this, the elements within the modules perform activities with no meaningful relationship to one another

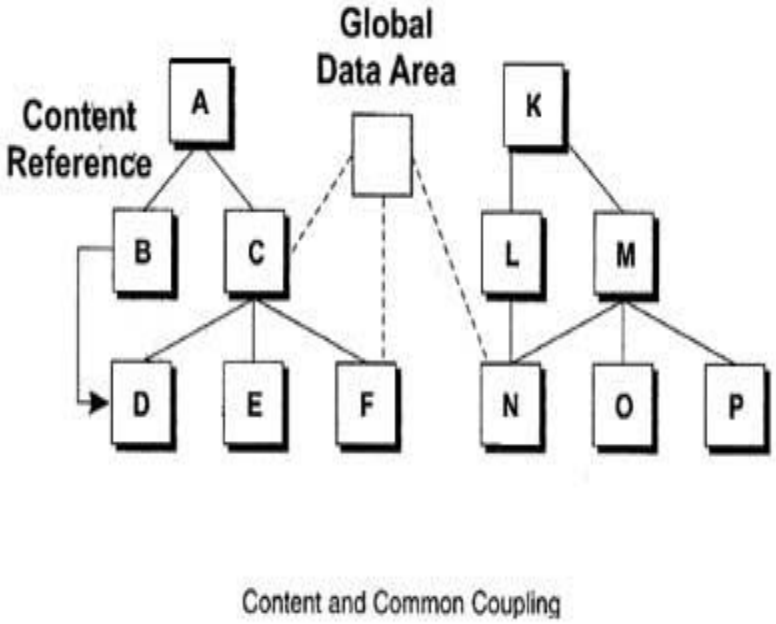
Component Level Design

Coupling

- Levels of coupling
  - Content
  - Common
  - Control
  - Stamp
  - Data
  - Routine call
  - Type use
  - Inclusion or import
  - External







- **Data coupling:** Two modules are said to be ‘data coupled’ if they use parameter list to pass data items for communication.
- **Stamp coupling:** Two modules are said to be ‘stamp coupled’ if they communicate by passing a data structure that stores additional information than what is required to perform their functions.
- **Control coupling:** Two modules are said to be ‘control coupled’ if they communicate (pass a piece of information intended to control the internal logic) using at least one ‘control flag’. The control flag is a variable whose value is used by the dependent modules to make decisions.

Component Level Design

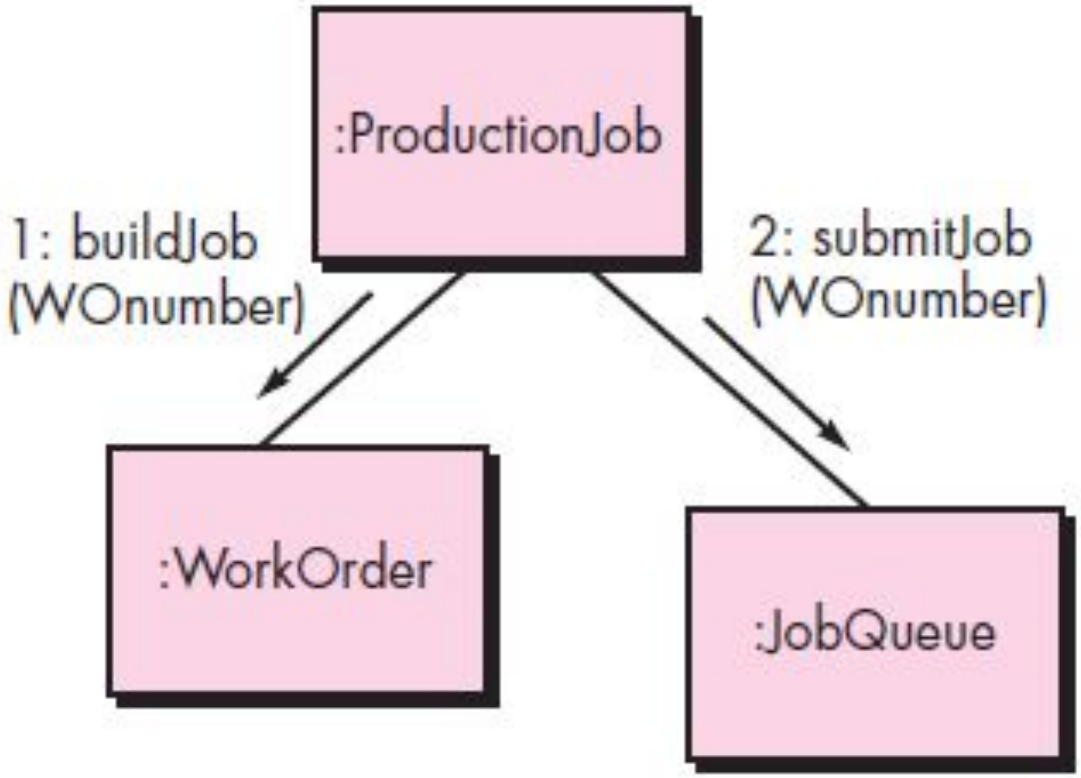
Conducting Component-Level Design

- Step 1. Identify all design classes that correspond to the problem domain.
- Step 2. Identify all design classes that correspond to the infrastructure domain.
- Step 3. Elaborate all design classes that are not acquired as reusable components.
- Step 3a. Specify message details when classes or component collaborate.
- Step 3b. Identify appropriate interfaces for each component.
- Step 3c. Elaborate attributes and define data types and data structures required to implement them.
- Step 3d. Describe processing flow within each operation in detail.
- Step 4. Describe persistent data sources (databases and files) and identify the classes required to manage them.
- Step 5. Develop and elaborate behavioral representations for a class or component.
- Step 6. Elaborate deployment diagrams to provide additional implementation detail.
- Step 7. Factor every component-level design representation and always consider alternatives.

Component Level Design

Conducting Component-Level Design

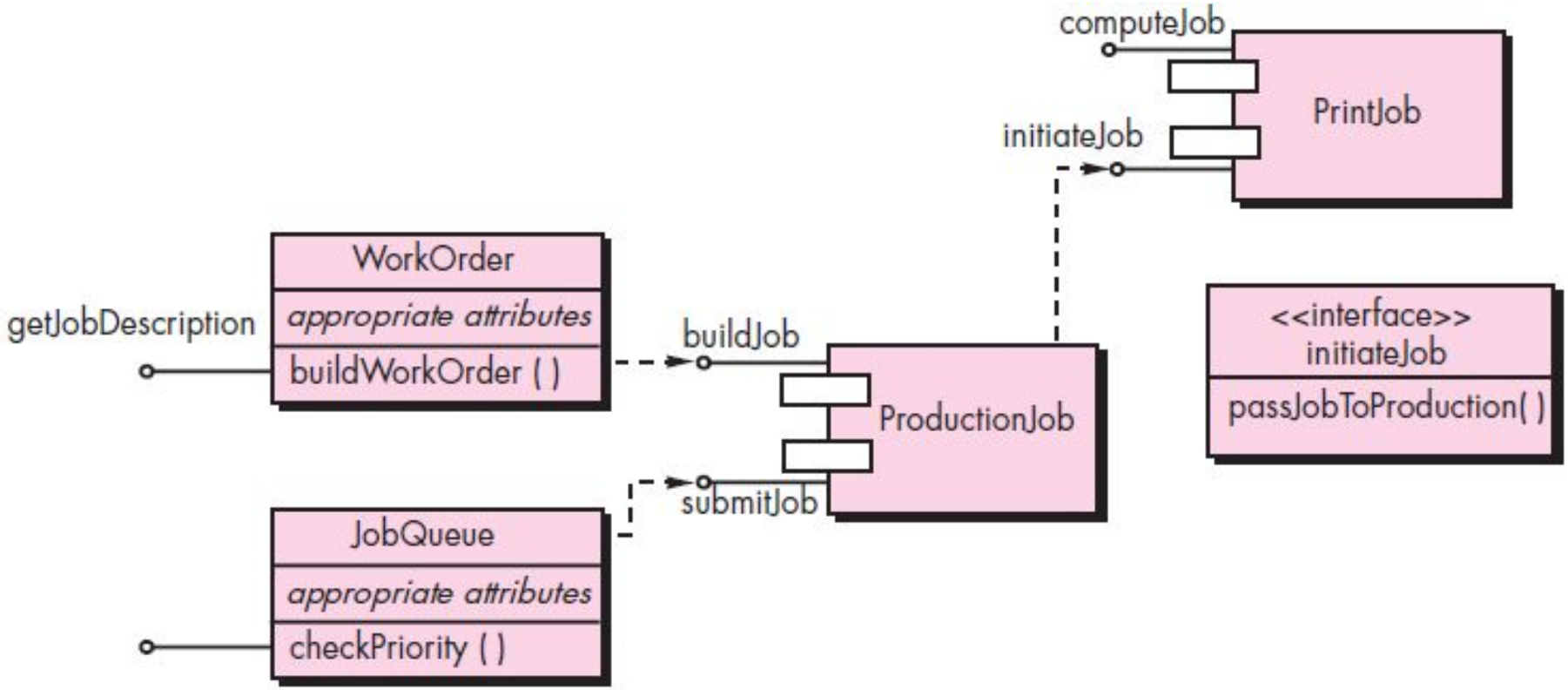
- Collaboration Diagram with Messaging (step 3)



Component Level Design

Conducting Component-Level Design

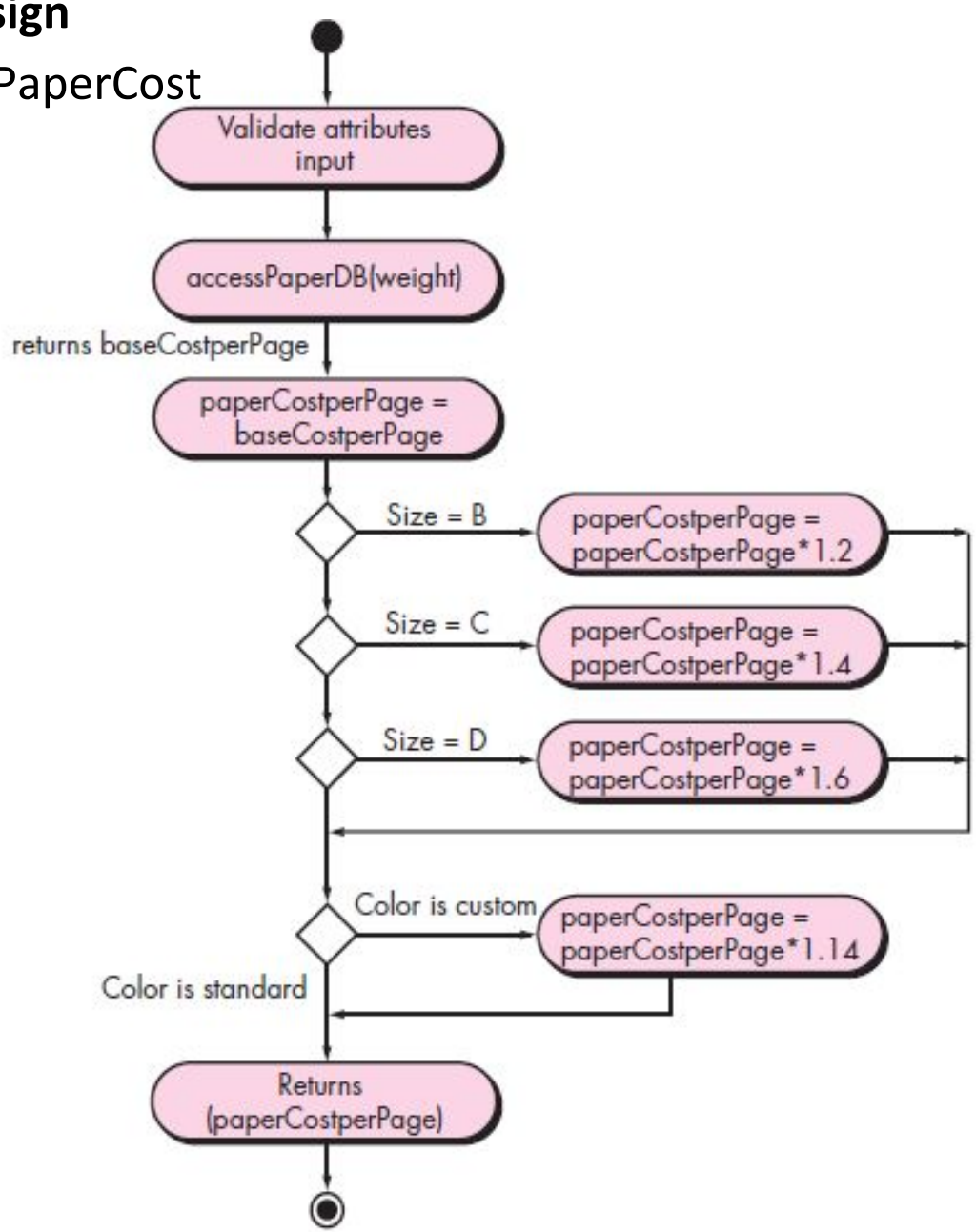
- Refactoring Interfaces and Class Definitions for PrintJob(step 3c)



Component Level Design

Conducting Component-Level Design

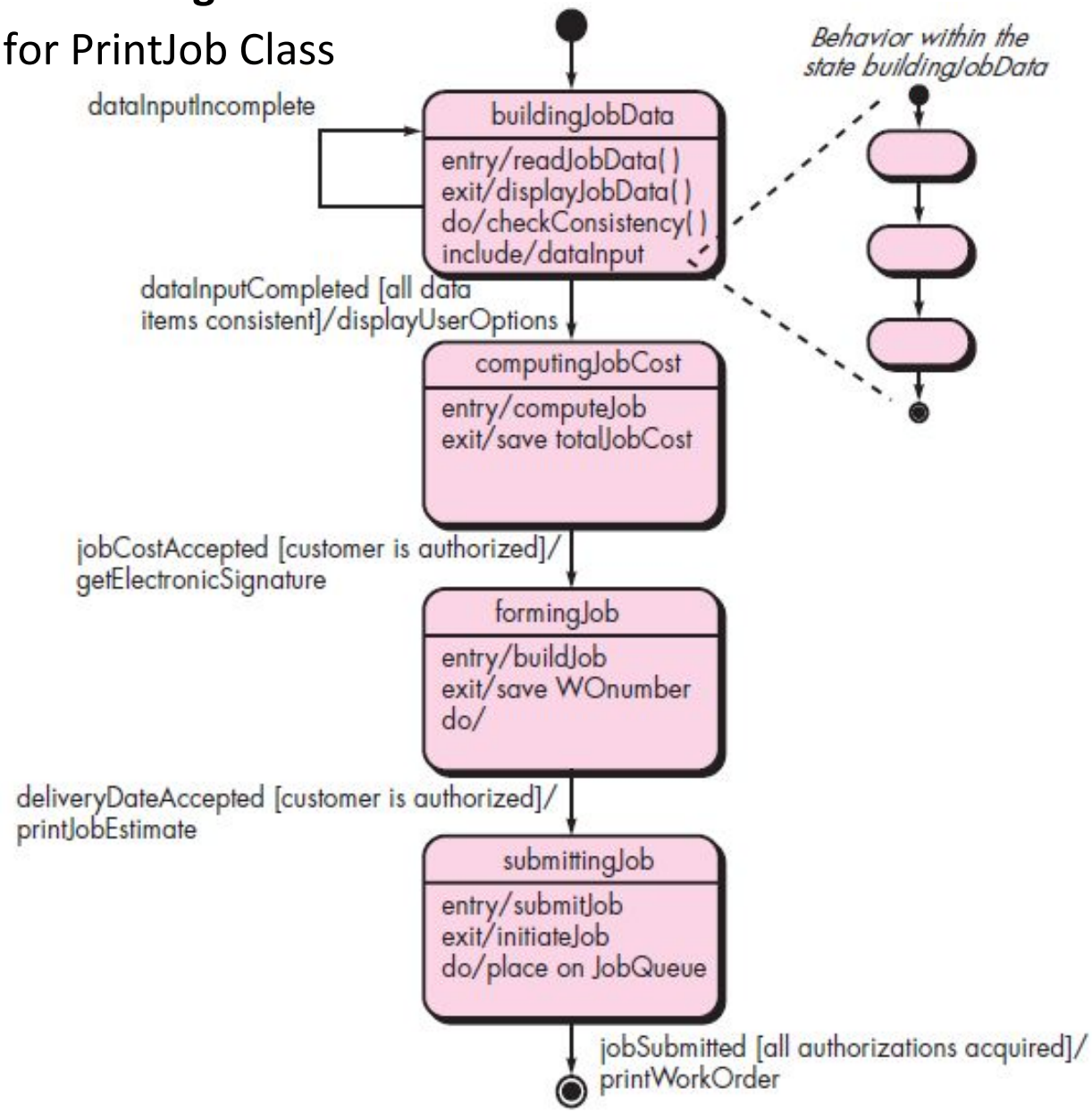
- Activity Diagram for Compute-PaperCost



Component Level Design

Conducting Component-Level Design

- State Chart Fragment for PrintJob Class



Component Level Design

Component Design for WebApps

- WebApp component is
  - (1) A well-defined cohesive function that manipulates content or provides computational or data processing for an end-user, or
  - (2) A cohesive package of content and functionality that provides end-user with some required capability.
- Therefore, component-level design for WebApps often incorporates elements of content design and functional design.

Content Design for WebApps

- Focuses on content objects and the manner in which they may be packaged for presentation to a WebApp end-user.
- Consider a Web-based video surveillance capability within SafeHomeAssured.com - Potential content components can be defined for the video surveillance capability:

  (1) The content objects that represent the space layout (the floor plan) with additional icons representing the location of sensors and video cameras;

  (2) The collection of thumbnail video captures (each an separate data object)

  (3) The streaming video window for a specific camera.
  - Each of these components can be separately named and manipulated as a package.

Component Level Design

Functional Design for WebApps

- Modern Web applications deliver increasingly sophisticated processing functions that:

  (1) Perform localized processing to generate content and navigation capability in a dynamic fashion;

  (2) Provide computation or data processing capability that is appropriate for the WebApps business domain;

  (3) Provide sophisticated database query and access, or

  (4) Establish data interfaces with external corporate systems.

- To achieve these (and many other) capabilities, it is suggested to design and construct WebApp functional components that are identical in form to software components for conventional software.

Component Level Design

Designing Conventional Components

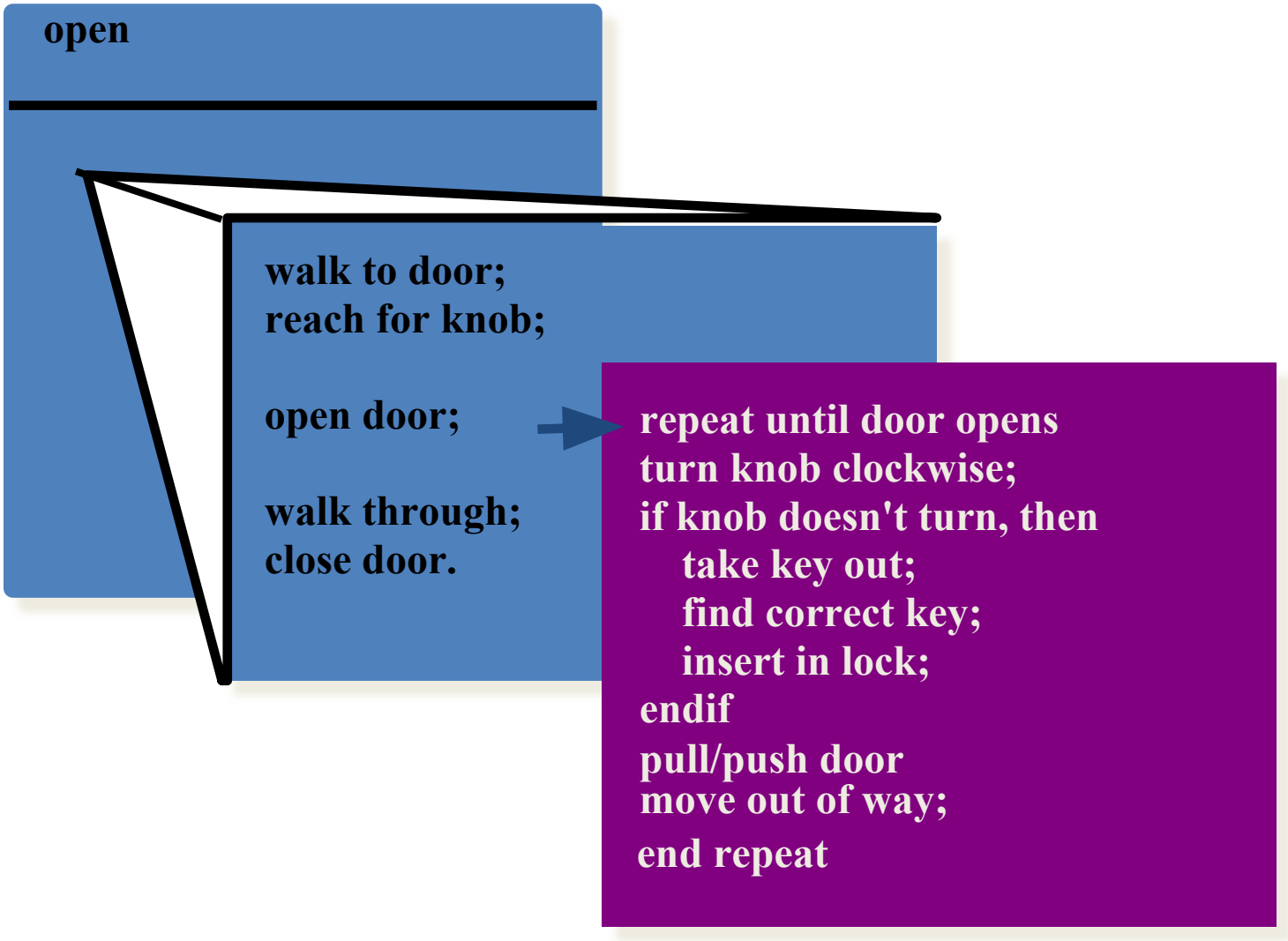
- The design of processing logic is governed by the basic principles of algorithm design and structured programming.
- The design of data structures is defined by the data model developed for the system.
- The design of interfaces is governed by the collaborations that a component must effect.

Algorithm Design

- The closest design activity to coding
- The approach:
  1. Review the design description for the component
  2. Use stepwise refinement to develop algorithm
  3. Use structured programming to implement procedural logic
  4. Use ‘formal methods’ to prove logic

Component Level Design

Algorithm Design – Stepwise Refinement



Component Level Design

Algorithm Design

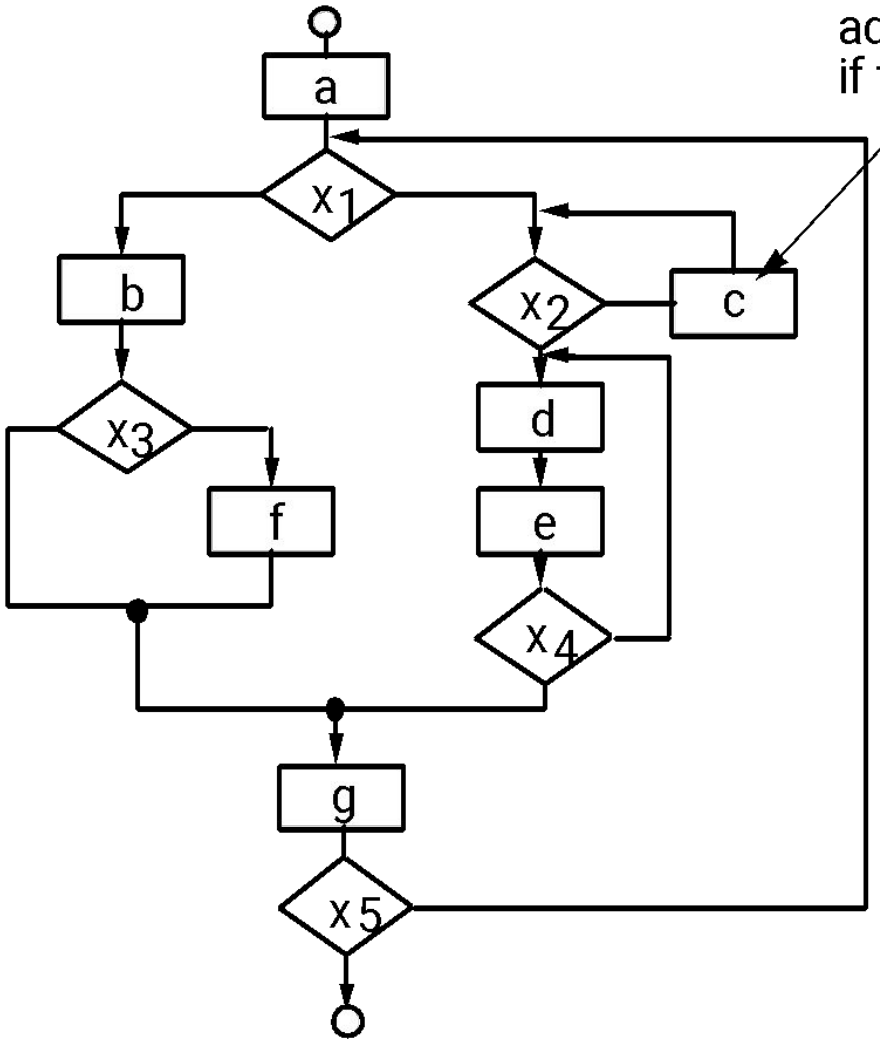
- Represents the algorithm at a level of detail that can be reviewed for quality
- Options:
  - Graphical (E.g. flowchart, box diagram)
  - Pseudo code (E.g., PDL) ... choice of many
  - Decision table

Structured Programming

- Uses a limited set of logical constructs:
  - Sequence
  - Conditional – If-then-else, Select-case
  - Loops – Do-while, Repeat until
- Leads to more readable, testable code
- Can be used in conjunction with “proof of correctness”

Component Level Design

Structured Procedural Design and Decision Table

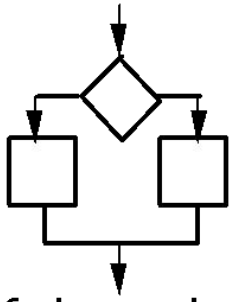


add a condition Z,
if true, exit the program

	Rules					
Conditions	**1**	**2**	**3**	**4**	**5**	**6**
regular customer	T	T				
silver customer			T	T		
gold customer					T	T
special discount	F	T	F	T	F	T
Rules						
no discount	✓					
apply 8 percent discount			✓	✓		
apply 15 percent discount					✓	✓
apply additional x percent discount		✓		✓		✓

Component Level Design

Program Design Language (PDL)



if-then-else

```
if condition x
  then process a;
  else process b;
endif
```

PDL

- easy to combine with source code
- machine readable, no need for graphics input
- graphics can be generated from PDL
- enables declaration of data as well as procedure
- easier to maintain

Component Level Design

Why Design Language?

- Machine readable and able to process
- Can be embedded with source code, therefore easier to maintain
- Can be represented in great detail, if designer and coder are different
- Easy to review

Component-based Development

- When faced with the possibility of reuse, the software team asks:
  - Are commercial off-the-shelf (COTS) components available to implement the requirement?
  - Are internally-developed reusable components available to implement the requirement?
  - Are the interfaces for available components compatible within the architecture of the system to be built?
- At the same time, they are faced with the following impediments to reuse ...

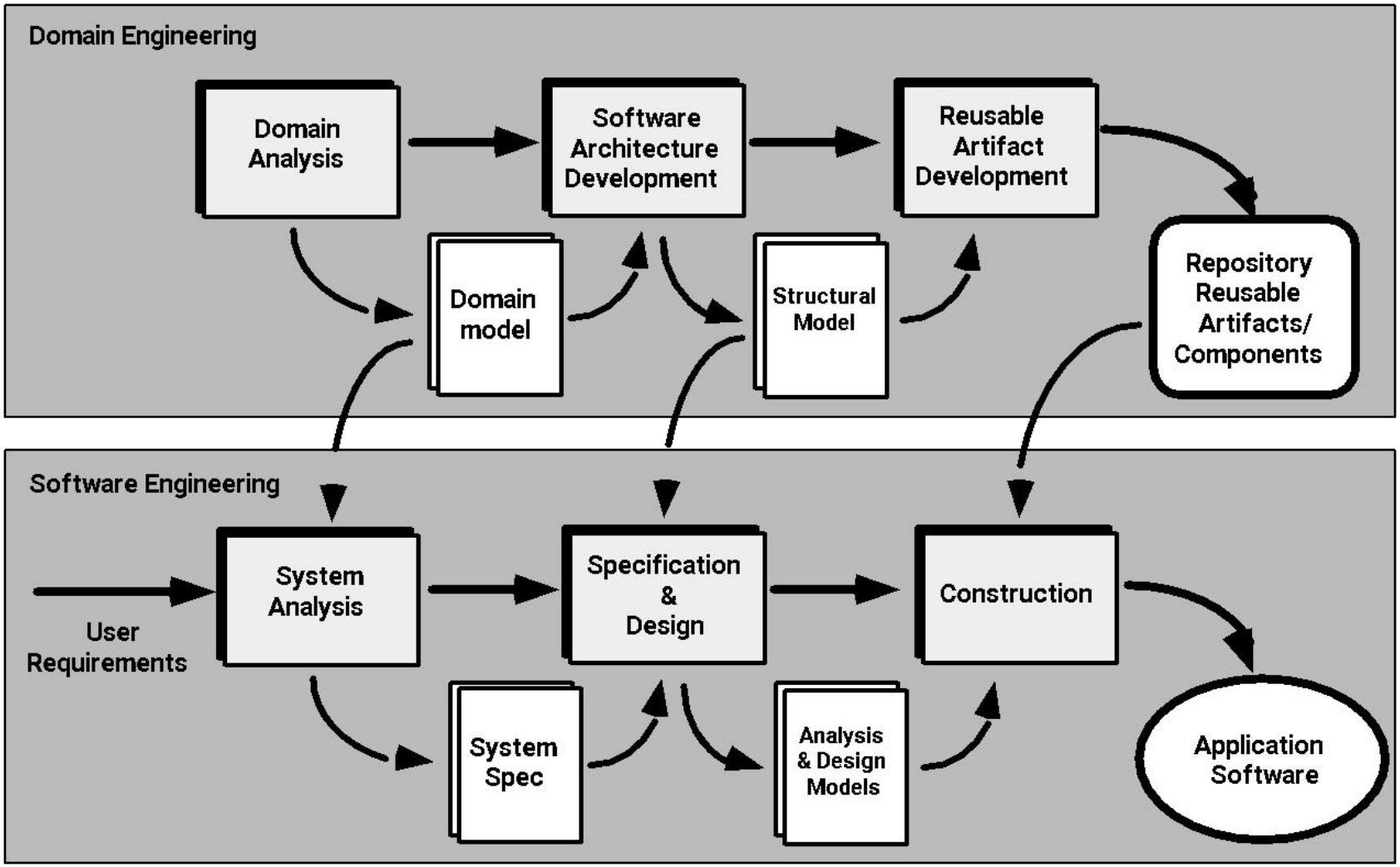
Component Level Design

Impediments to Reuse

- Few companies and organizations have anything that even slightly resembles a comprehensive software reusability plan.
- Although an increasing number of software vendors currently sell tools or components that provide direct assistance for software reuse, the majority of software developers do not use them.
- Relatively little training is available to help software engineers and managers understand and apply reuse.
- Many software practitioners continue to believe that reuse is “more trouble than it’s worth.”
- Many companies continue to encourage of software development methodologies which do not facilitate reuse.
- Few companies provide incentives to produce reusable program components.

Component Level Design

Component Based Software Engineering (CBSE) Process



Component Level Design

Domain Engineering

1. Define the domain to be investigated.
2. Categorize the items extracted from the domain.
3. Collect a representative sample of applications in the domain.
4. Analyze each application in the sample.
5. Develop an analysis model for the objects.

Identifying Reusable Components

- Is component functionality required on future implementations?
- How common is the component's function within the domain?
- Is there duplication of the component's function within the domain?
- Is the component hardware-dependent?
- Does the hardware remain unchanged between implementations?
- Can the hardware specifics be removed to another component?
- Is the design optimized enough for the next implementation?
- Can we parameterize a non-reusable component so that it becomes reusable?
- Is the component reusable in many implementations with only minor changes?

Component Level Design

Identifying Reusable Components

- Is reuse through modification feasible?
- Can a non-reusable component be decomposed to yield reusable components?
- How valid is component decomposition for reuse?

Component-Based SE

- A library of components must be available
- Components should have a consistent structure
- A standard should exist, e.g.,
  - OMG/CORBA
  - Microsoft COM
  - Sun JavaBeans

CBSE Activities

- Component qualification
- Component adaptation
- Component composition
- Component update

Component Level Design

CBSE Activities – Qualification

Before a component can be used, it is necessary to consider:

- Application programming interface (API)
- Development and integration tools required by the component
- Run-time requirements including resource usage (e.g., memory or storage), timing or speed, and network protocol.
- Service requirements including operating system interfaces and support from other components.
- Security features including access controls and authentication protocol.
- Embedded design assumptions including the use of specific numerical or non-numerical algorithms.
- Exception handling

Component Level Design

CBSE Activities – Adaptation

The implication of “easy integration” is:

(1) that consistent methods of resource management have been implemented for all components in the library;

(2) that common activities such as data management exist for all components, and

(3) that interfaces within the architecture and with the external environment have been implemented in a consistent manner.

CBSE Activities – Composition

- An infrastructure must be established to bind components together.
- Architectural ingredients for composition include:
  - Data exchange model
  - Automation
  - Structured storage and Underlying object model
  - **Component Update:**
    This activity ensures the updation of reusable components. Sometimes, updates are complicated due to inclusion of third party

Component Level Design

OMG/CORBA

- The **Object Management Group** has published a **Common Object Request Broker Architecture** (OMG/CORBA).
- An object request broker (ORB) provides services that enable reusable components (objects) to communicate with other components, regardless of their location within a system.
- Integration of CORBA (Common Object Request Broker Architecture) components (without modification) within a system is assured if an **Interface Definition Language (IDL)** interface is created for every component.
- Objects within the client application request one or more services from the ORB server.
- Requests are made via an IDL or dynamically at run time.
- An interface repository contains all necessary information about the service's request and response formats.

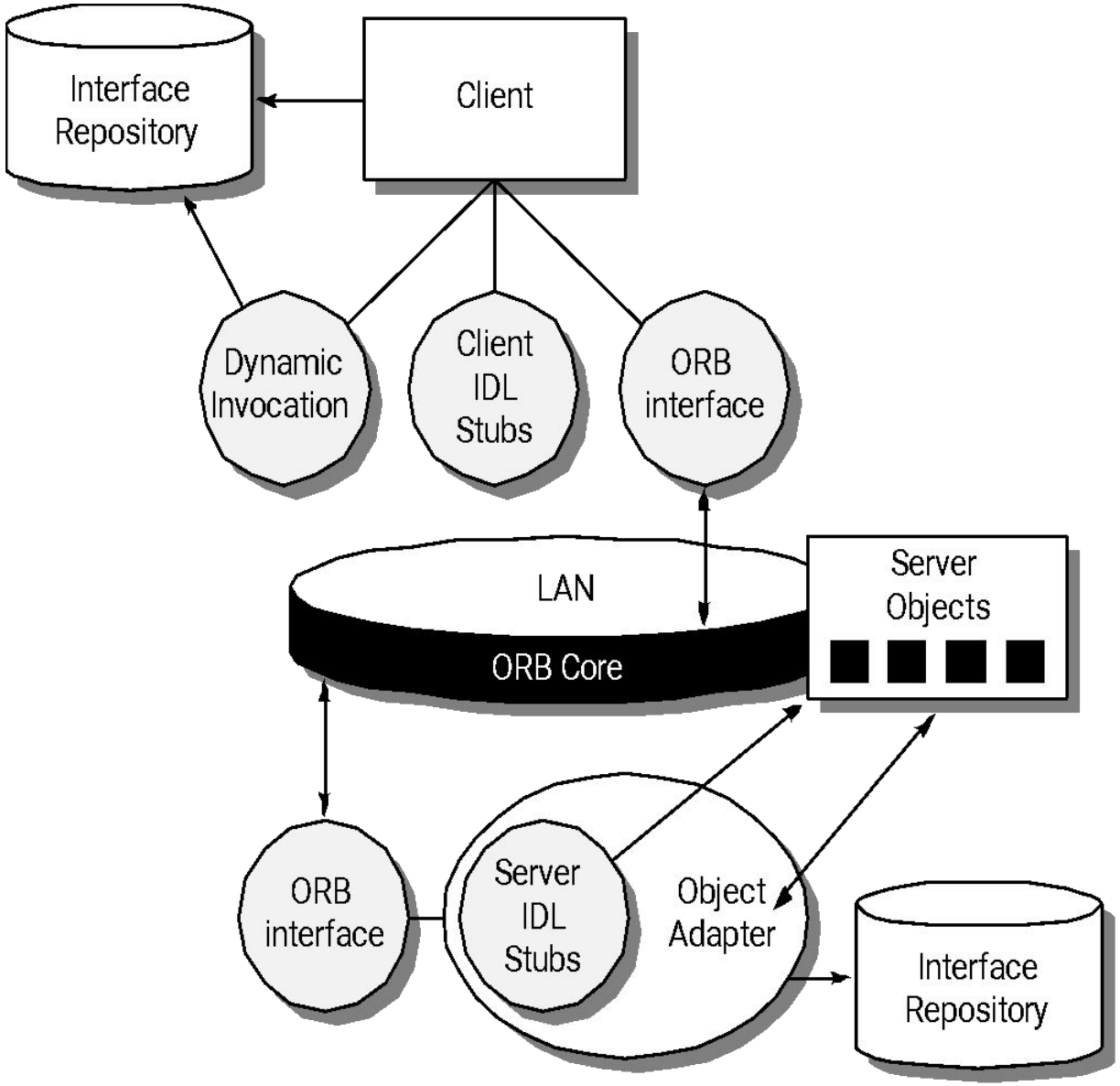
- An object request broker (ORB) is a middleware application component that uses the common object request broker architecture (CORBA) specification, enabling developers to make application calls within a computer network.
- ORB is an agent that transmits client/server operation invocations in a distributed environment and ensures transparent object communication.

- **ORB does the following:**

- Searches, matches and instantiates remote machine objects
- Gathers parameters between application objects
- Handles security issues across machine boundaries
- Retrieves and publishes data objects on local machines available for other ORBs
- Invokes remote object methods using static and dynamic method invocation.
- Instantiates idle objects automatically
- Routes callback methods
- Communicates Inter-ORB Protocol (IIOP) with other ORBs via the Internet

Component Level Design

ORB Architecture



Component Level Design

Microsoft COM

- The Microsoft Component Object Model (COM) is a platform-independent, distributed, object-oriented system for creating binary software components that can interact.
- The component object model (COM) provides a specification for using components produced by various vendors within a single application running under the Windows operating system.
- COM encompasses two elements:
  - COM interfaces (implemented as COM objects)
  - A set of mechanisms for registering and passing messages between COM interfaces.

SUN JavaBeans

- The JavaBeans component system is a portable, platform independent CBSE infrastructure developed using the Java programming language.
- The JavaBeans component system encompasses a set of tools, called the **Bean Development Kit (BDK**), that allows developers to
  - Analyze how existing Beans (components) work
  - Customize their behavior and appearance
  - Establish mechanisms for coordination and communication
  - Develop custom Beans for use in a specific application

User Interface Design

Interface Design

- Easy to learn?
- Easy to use?
- Easy to understand?

Typical Design Errors

- Lack of consistency
- Too much memorization
- No guidance / help
- No context sensitivity
- Poor response
- Arcane/unfriendly

User Interface Design

Golden Rules

- Place the user in control
- Reduce the user's memory load
- Make the interface consistent

Place the User in Control

- Define interaction modes in a way that does not force a user into unnecessary or undesired actions.
- Provide for flexible interaction.
- Allow user interaction to be interruptible and undoable.
- Streamline interaction as skill levels advance and allow the interaction to be customized.
- Hide technical internals from the casual user.
- Design for direct interaction with objects that appear on the screen.

User Interface Design

Reduce the User's Memory Load

- Reduce demand on short-term memory.
- Establish meaningful defaults.
- Define shortcuts that are intuitive.
- The visual layout of the interface should be based on a real world metaphor.
- Disclose information in a progressive fashion.

Make the Interface Consistent

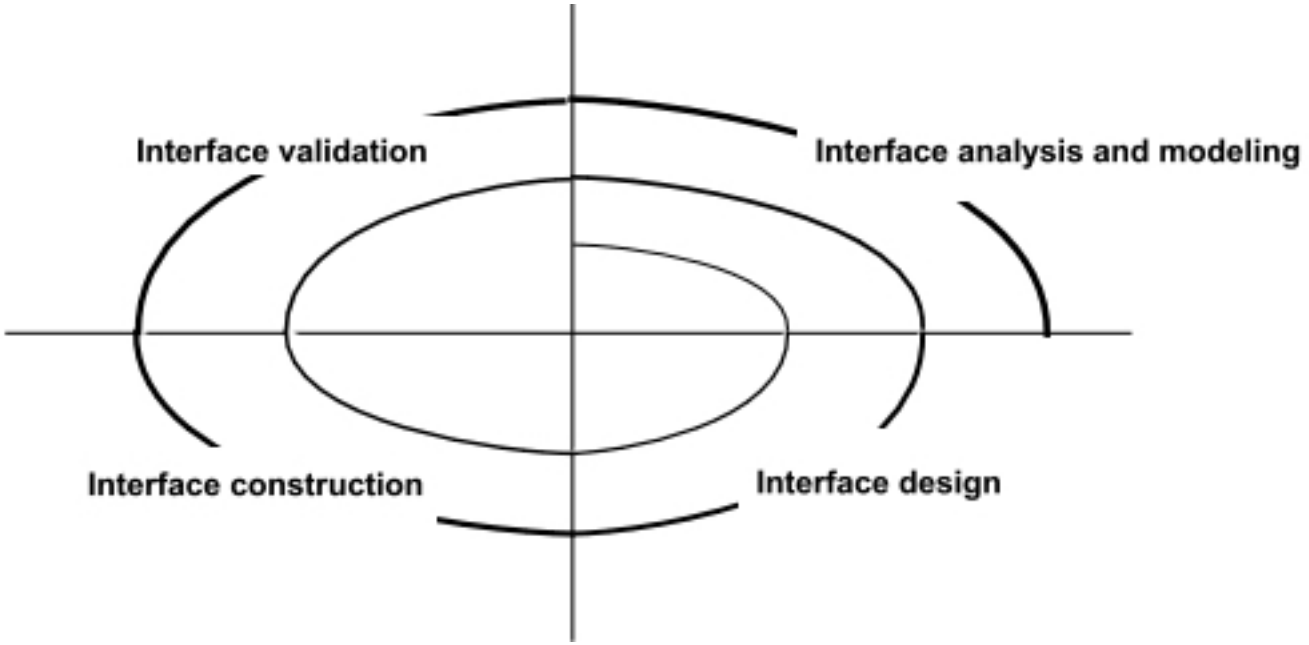
- Allow the user to put the current task into a meaningful context.
- Maintain consistency across a family of applications.
- If past interactive models have created user expectations, do not make changes unless there is a compelling reason to do so.

User Interface Design

User Interface Design Models

- User model — A profile of all end users of the system
- Design model — A design realization of the user model
- Mental model (system perception) — The user's mental image of what the interface is
- Implementation model — The interface "look and feel" coupled with supporting information that describe interface syntax and semantics.

User Interface Design Process



User Interface Design

User Interface Design Process – Interface Analysis

- Interface 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

- Are users trained professionals, technician, clerical, or manufacturing workers?
- What level of formal education does the average user have?
- Are the users capable of learning from written materials or have they expressed a desire for classroom training?
- Are users expert typists or keyboard phobic?
- What is the age range of the user community?
- Will the users be represented predominately by one gender?
- How are users compensated for the work they perform?
- Do users work normal office hours or do they work until the job is done?

User Interface Design

User Interface Design Process – Interface Analysis

User Analysis

- Is the software to be an integral part of the work users do or will it be used only occasionally?
- What is the primary spoken language among users?
- What are the consequences if a user makes a mistake using the system?
- Are users experts in the subject matter that is addressed by the system?
- Do users want to know about the technology that sits behind the interface?

Task Analysis and Modeling

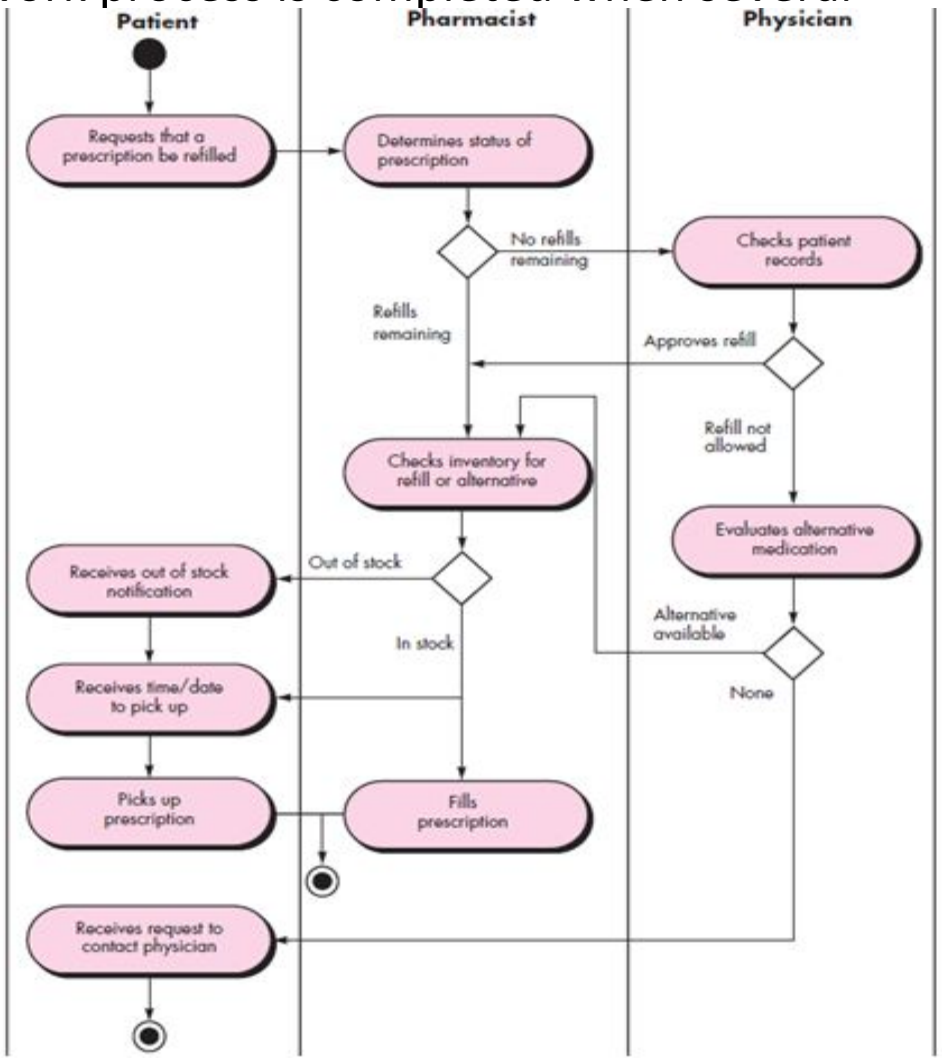
- Answers the following questions …
  - What work will the user perform in specific circumstances?
  - What tasks and subtasks will be performed as the user does the work?
  - What specific problem domain objects will the user manipulate as work is performed?
  - What is the sequence of work tasks—the workflow?
  - What is the hierarchy of tasks?
- Use-cases define basic interaction
- Task elaboration refines interactive tasks

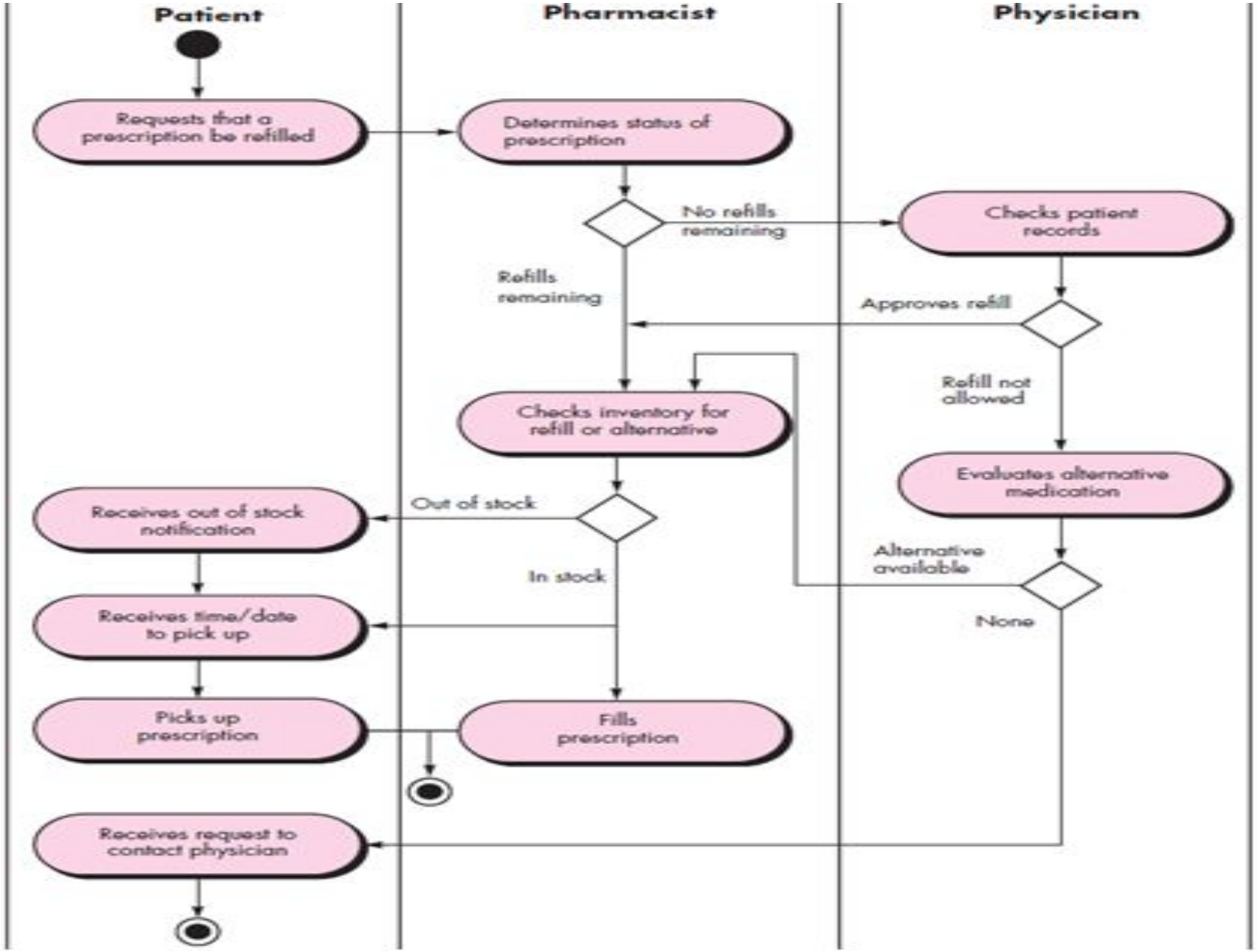
User Interface Design

User Interface Design Process – Interface Analysis

Task Analysis and Modeling

- Object elaboration identifies interface objects (classes)
- Workflow analysis defines how a work process is completed when several people (and roles) are involved
  - **Swimlane Diagram for Prescription refill function**




Patient
Pharmacist
Physician
Requests that a prescription be refilled
Determines status of prescription
No refills remaining
Checks patient records
Refills remaining
Approves refill
Refill not allowed
Checks inventory for refill or alternative
Evaluates alternative medication
Out of stock
Receives out of stock notification
Alternative available
In stock
None
Receives time/date to pick up
Picks up prescription
Fills prescription
Receives request to contact physician

User Interface Design

Analysis of Display Content

- Are different types of data assigned to consistent geographic locations on the screen (e.g., photos always appear in the upper right hand corner)?
- Can the user customize the screen location for content?
- Is proper on-screen identification assigned to all content?
- If a large report is to be presented, how should it be partitioned for ease of understanding?
- Will mechanisms be available for moving directly to summary information for large collections of data?
- Will graphical output be scaled to fit within the bounds of the display device that is used?
- How will color be used to enhance understanding?
- How will error messages and warning be presented to the user?

User Interface Design

Interface Design Steps

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

Design Issues

- Response time
- Help facilities
- Error handling
- Menu and command labeling
- Application accessibility
- Internationalization

User Interface Design

WebApp Interface Design

- Where am I?

  The interface should
  - Provide an indication of the WebApp that has been accessed
  - Inform the user of her location in the content hierarchy.

- What can I do now?

  The interface should always help the user understand his current options
  - What functions are available?
  - What links are live?
  - What content is relevant?

- Where have I been, where am I going?

  The interface must facilitate navigation.
  - Provide a “map” (implemented in a way that is easy to understand) of where the user has been and what paths may be taken to move elsewhere within the WebApp.

User Interface Design

Interface Design Principles

- **Anticipation**—A WebApp should be designed so that it anticipates the use's next move.
- **Communication**—The interface should communicate the status of any activity initiated by the user
- **Consistency**—The use of navigation controls, menus, icons, and aesthetics (e.g., color, shape, layout)
- **Controlled autonomy**—The interface should facilitate user movement throughout the WebApp, but it should do so in a manner that enforces navigation conventions that have been established for the application.
- **Efficiency**—The design of the WebApp and its interface should optimize the user's work efficiency, not the efficiency of the Web engineer who designs and builds it or the client-server environment that executes it.
- **Focus**—The WebApp interface (and the content it presents) should stay focused on the user task(s) at hand.
- **Fitt's Law**—"The time to acquire a target is a function of the distance to and size of the target."
- **Human interface objects—A** vast library of reusable human interface objects has been developed for WebApps.

User Interface Design

Interface Design Principles

- **Latency reduction** —The WebApp should use multi-tasking in a way that lets the user proceed with work as if the operation has been completed.
- **Learnability**— A WebApp interface should be designed to minimize learning time, and once learned, to minimize relearning required when the WebApp is revisited.
- **Maintain work product integrity** —A work product (e.g., a form completed by the user, a user specified list) must be automatically saved so that it will not be lost if an error occurs.
- **Readability** —All information presented through the interface should be readable by young and old.
- **Track state** —When appropriate, the state of the user interaction should be tracked and stored so that a user can logoff and return later to pick up where she left off.
- **Visible navigation** —A well-designed WebApp interface provides “the illusion that users are in the same place, with the work brought to them.”

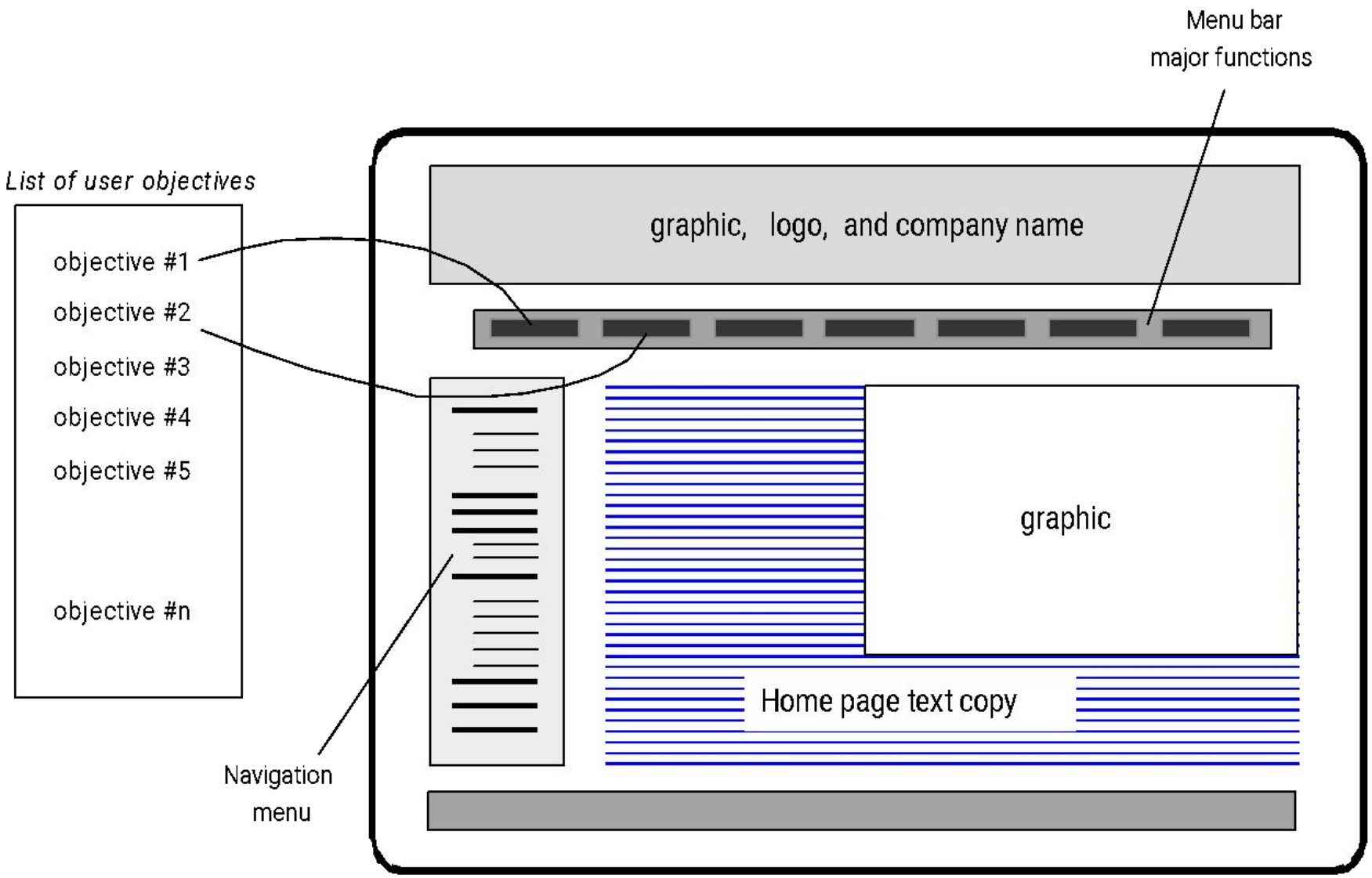
User Interface Design

Interface Design Workflow

- Review information contained in the analysis model and refine as required.
- Develop a rough sketch of the WebApp interface layout.
- Map user objectives into specific interface actions.
- Define a set of user tasks that are associated with each action.
- Storyboard screen images for each interface action.
- Refine interface layout and storyboards using input from aesthetic design.
- Identify user interface objects that are required to implement the interface.
- Develop a procedural representation of the user's interaction with the interface.
- Develop a behavioral representation of the interface.
- Describe the interface layout for each state.
- Refine and review the interface design model.

User Interface Design

Mapping User Objectives



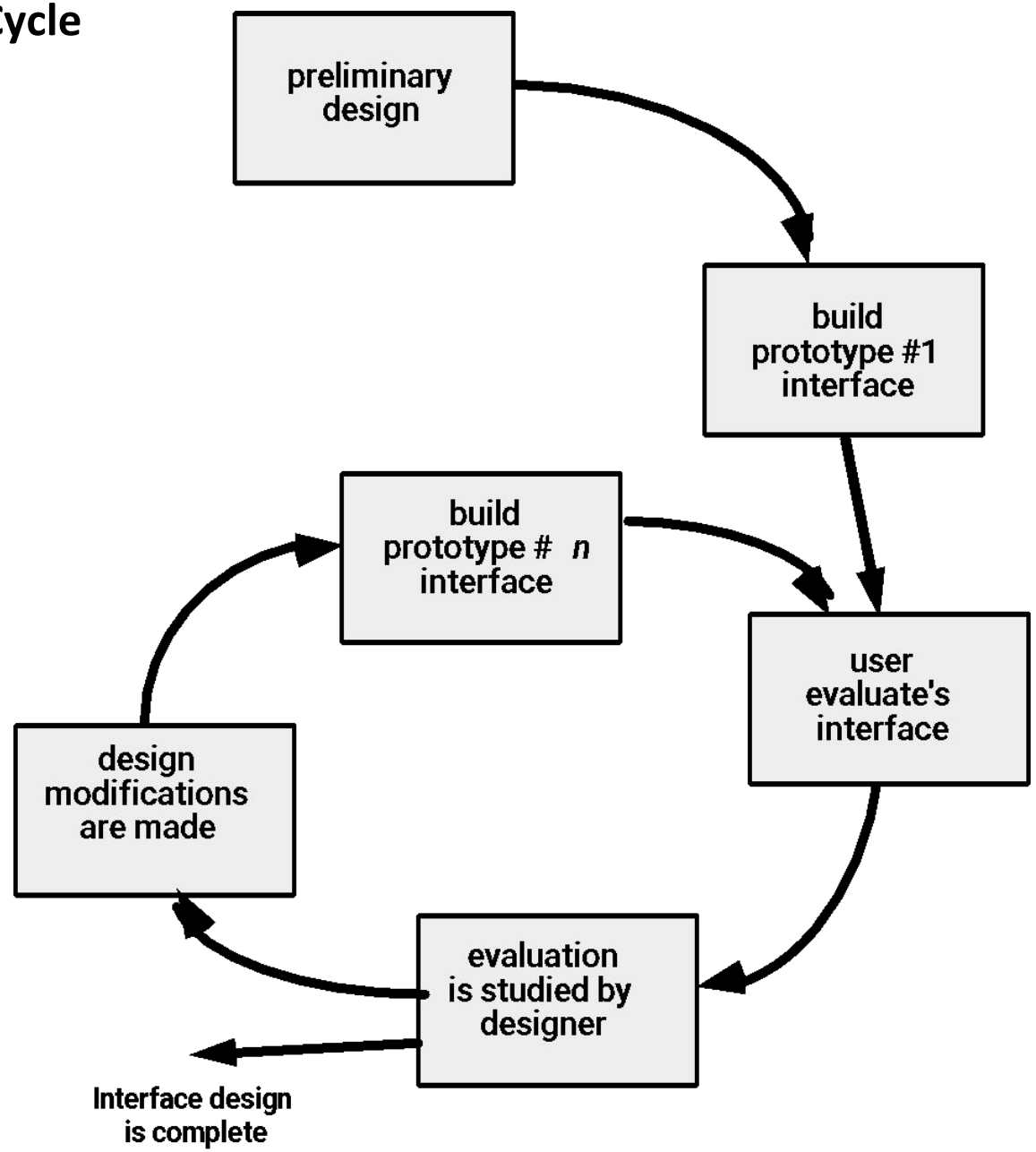
User Interface Design

Aesthetic Design

- Don’t be afraid of white space.
- Emphasize content.
- Organize layout elements from top-left to bottom right.
- Group navigation, content, and function geographically within the page.
- Don’t extend your real estate with the scrolling bar.
- Consider resolution and browser window size when designing layout.

User Interface Design

Design Evaluation Cycle



Pattern Oriented Design

Patterns for Requirements Modeling

- Software patterns are a mechanism for capturing domain knowledge in a way that allows it to be reapplied when a new problem is encountered.
  - Domain knowledge can be applied to a new problem within the same application domain
  - The domain knowledge captured by a pattern can be applied by analogy to a completely different application domain.
- The original author of an analysis pattern does not “create” the pattern, but rather, discovers it as requirements engineering work is being conducted.
- Once the pattern has been discovered, it is documented.

Discovering Analysis Patterns

- The most basic element in the description of a requirements model is the use case.
- A coherent set of use cases may serve as the basis for discovering one or more analysis patterns.
- A semantic analysis pattern (SAP) “is a pattern that describes a small set of coherent use cases that together describe a basic generic application.”

Pattern Oriented Design

An Example

- Consider the following preliminary use case for software required to control and monitor a rear-view camera and proximity sensor for an automobile:
    - **Use case: Monitor reverse motion**
    - **Description:** When the vehicle is placed in reverse gear, the control software enables a video feed from a rear-placed video camera to the dashboard display.
    - The control software superimposes a variety of distance and orientation lines on the dashboard display so that the vehicle operator can maintain orientation as the vehicle moves in reverse.
    - The control software also monitors a proximity sensor to determine whether an object is inside 10 feet of the rear of the vehicle.
    - It will automatically break the vehicle if the proximity sensor indicates an object within 3 feet of the rear of the vehicle.
- This use case implies a variety of functionality that would be refined and elaborated (into a coherent set of use cases) during requirements gathering and modeling.

Pattern Oriented Design

An Example

- Regardless of how much elaboration is accomplished, the use case(s) suggest(s) a simple, yet widely applicable SAP (Semantic Analysis Pattern)—the software-based monitoring and control of sensors and actuators in a physical system.
- In this case, the “sensors” provide information about proximity and video information.
- The “actuator” is the breaking system of the vehicle (invoked if an object is very close to the vehicle.
- But in a more general case, a widely applicable pattern is discovered --> Actuator-Sensor

Pattern Oriented Design

Actuator-Sensor Pattern

- **Pattern Name:** Actuator-Sensor
- **Intent:** Specify various kinds of sensors and actuators in an embedded system.

Motivation:

- Embedded systems usually have various kinds of sensors and actuators.
- These sensors and actuators are all either directly or indirectly connected to a control unit.
- Although many of the sensors and actuators look quite different, their behavior is similar enough to structure them into a pattern.
- The pattern shows how to specify the sensors and actuators for a system, including attributes and operations.
- The Actuator-Sensor pattern uses a pull mechanism (explicit request for information) for Passive Sensors and a push mechanism (broadcast of information) for the Active Sensors.

Pattern Oriented Design

Actuator-Sensor Pattern

Constraints:

- Each passive sensor must have some method to read sensor input and attributes that represent the sensor value.
- Each active sensor must have capabilities to broadcast update messages when its value changes.
- Each active sensor should send a life tick, a status message issued within a specified time frame, to detect malfunctions.
- Each actuator must have some method to invoke the appropriate response determined by the ComputingComponent.
- Each sensor and actuator should have a function implemented to check its own operation state.
- Each sensor and actuator should be able to test the validity of the values received or sent and set its operation state if the values are outside of the specifications.

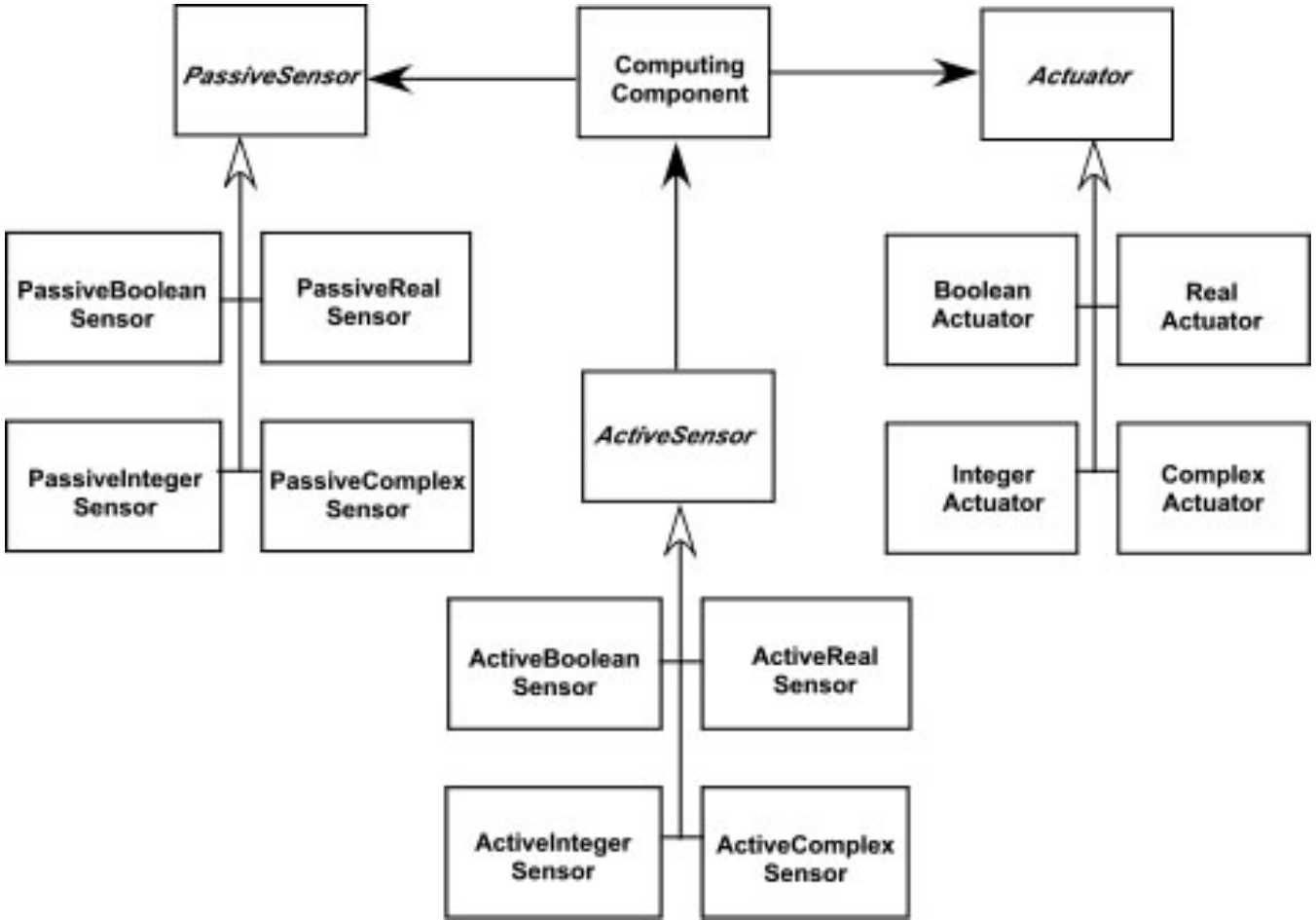
Pattern Oriented Design

Actuator-Sensor Pattern

Applicability: Useful in any system in which multiple sensors and actuators are present.

Structure:

- A UML class diagram for the Actuator-Sensor Pattern is shown below.



Pattern Oriented Design

Actuator-Sensor Pattern

Structure:

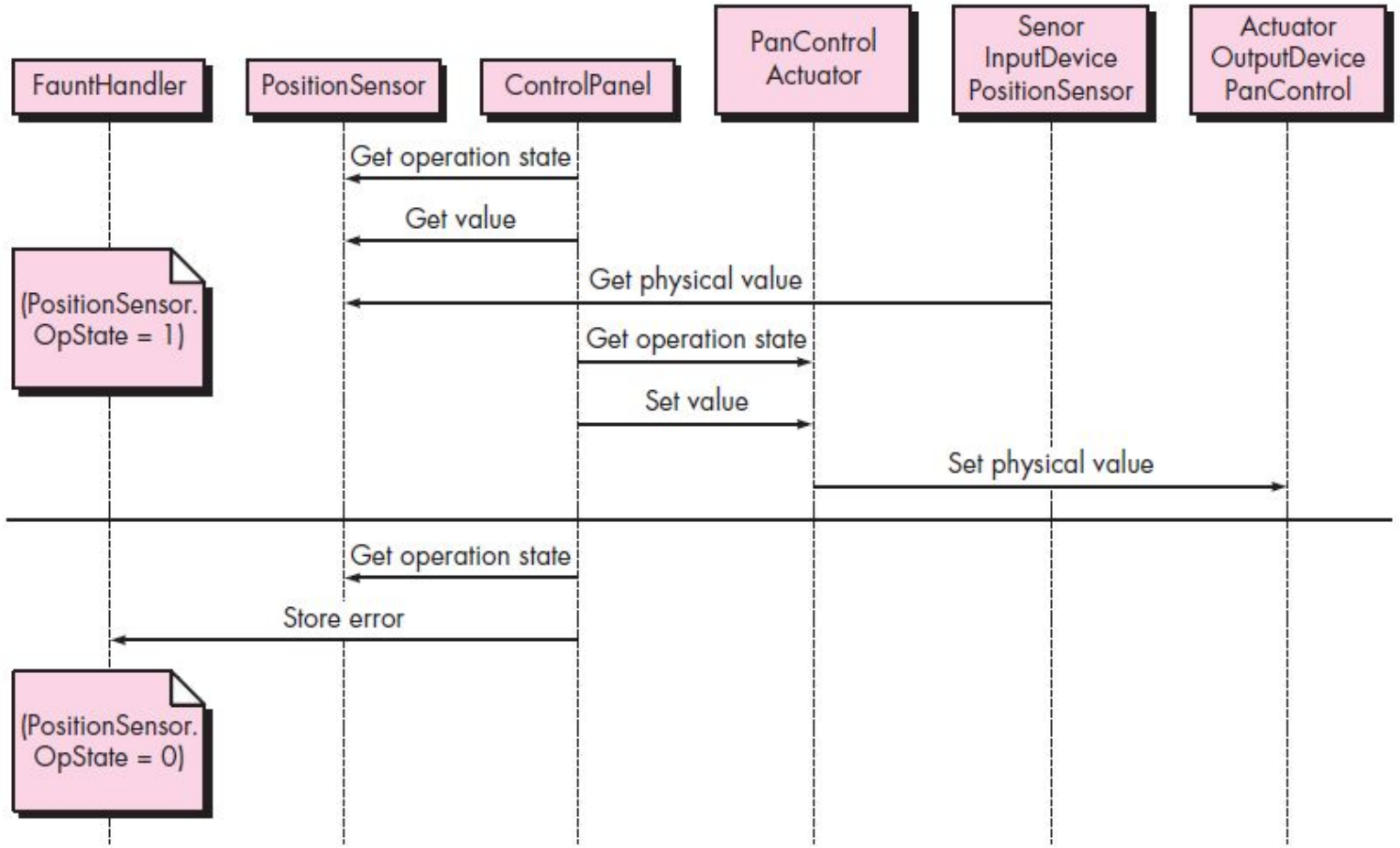
- Actuator, PassiveSensor and ActiveSensor are abstract classes and denoted in italics.
- There are four different types of sensors and actuators in this pattern.
- The Boolean, integer, and real classes represent the most common types of sensors and actuators.
- The complex classes are sensors or actuators that use values that cannot be easily represented in terms of primitive data types, such as a radar device.
- Nonetheless, these devices should still inherit the interface from the abstract classes since they should have basic functionalities such as querying the operation states.

Pattern Oriented Design

Actuator-Sensor Pattern

Behavior:

- The figure below presents a UML sequence diagram for an example of the Actuator-Sensor Pattern as it might be applied for the SafeHome function that controls the positioning (e.g., pan, zoom) of a security camera.



Pattern Oriented Design

Actuator-Sensor Pattern

Behavior:

- Here, the Control Panel queries a sensor (a passive position sensor) and an actuator (pan control) to check the operation state for diagnostic purposes before reading or setting a value.
- The messages Set Physical Value and Get Physical Value are not messages between objects.
- Instead, they describe the interaction between the physical devices of the system and their software counterparts.
- In the lower part of the diagram, below the horizontal line, the Position Sensor reports that the operation state is zero.
- The ComputingComponent then sends the error code for a position sensor failure to the Fault Handler that will decide how this error affects the system and what actions are required.
- It gets the data from the sensors and computes the required response for the actuators.

Web Application Design

- **Content Analysis** - The full spectrum of content to be provided by the WebApp is identified, including text, graphics and images, video, and audio data. Data modeling can be used to identify and describe each of the data objects.

- **Interaction Analysis** - The manner in which the user interacts with the WebApp is described in detail. Use-cases can be developed to provide detailed descriptions of this interaction.

- **Functional Analysis** - The usage scenarios (use-cases) created as part of interaction analysis define the operations that will be applied to WebApp content and imply other processing functions. All operations and functions are described in detail.

- **Configuration Analysis** - The environment and infrastructure in which the WebApp resides are described in detail.

Web Application Design

When Do We Perform Analysis?

- In some Web situations, analysis and design merge.
- However, an explicit analysis activity occurs when…
  - The WebApp to be built is large and/or complex
  - The number of stakeholders is large
  - The number of Web engineers and other contributors is large
  - The goals and objectives (determined during formulation) for the WebApp will effect the business' bottom line
  - The success of the WebApp will have a strong bearing on the success of the business

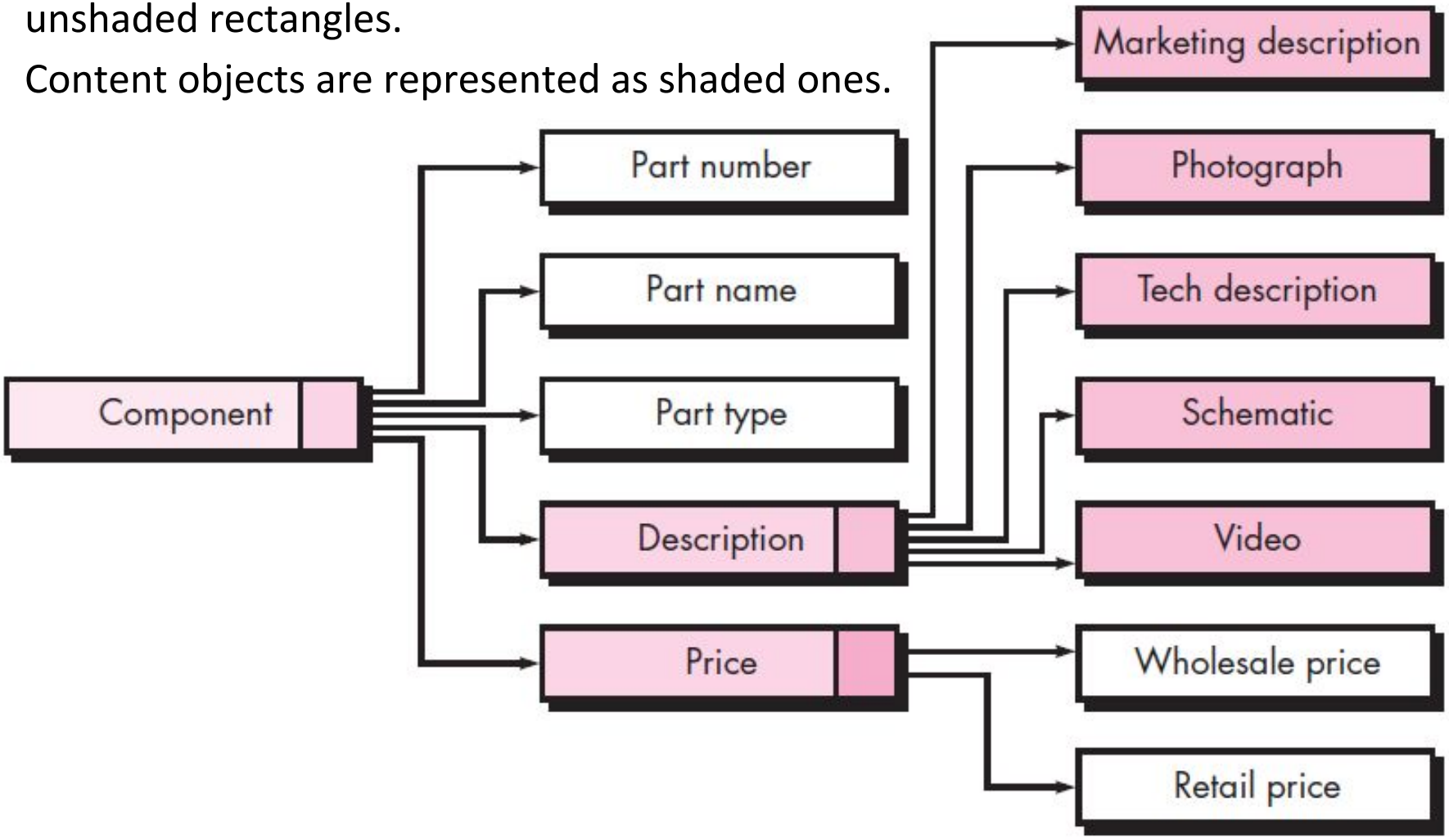
The Content Model

- Content objects are extracted from use-cases
  - Examine the scenario description for direct and indirect references to content
- Attributes of each content object are identified
- The relationships among content objects and/or the hierarchy of content maintained by a WebApp
  - Relationships—Entity-relationship diagram or UML
  - Hierarchy—Data tree or UML

Web Application Design

Data Tree

- Data tree for a SafeHome-Assured.com component – represents a hierarchy of information that is used to describe a component.
- Simple or composite data items (one or more data values) are represented as unshaded rectangles.
- Content objects are represented as shaded ones.



Web Application Design

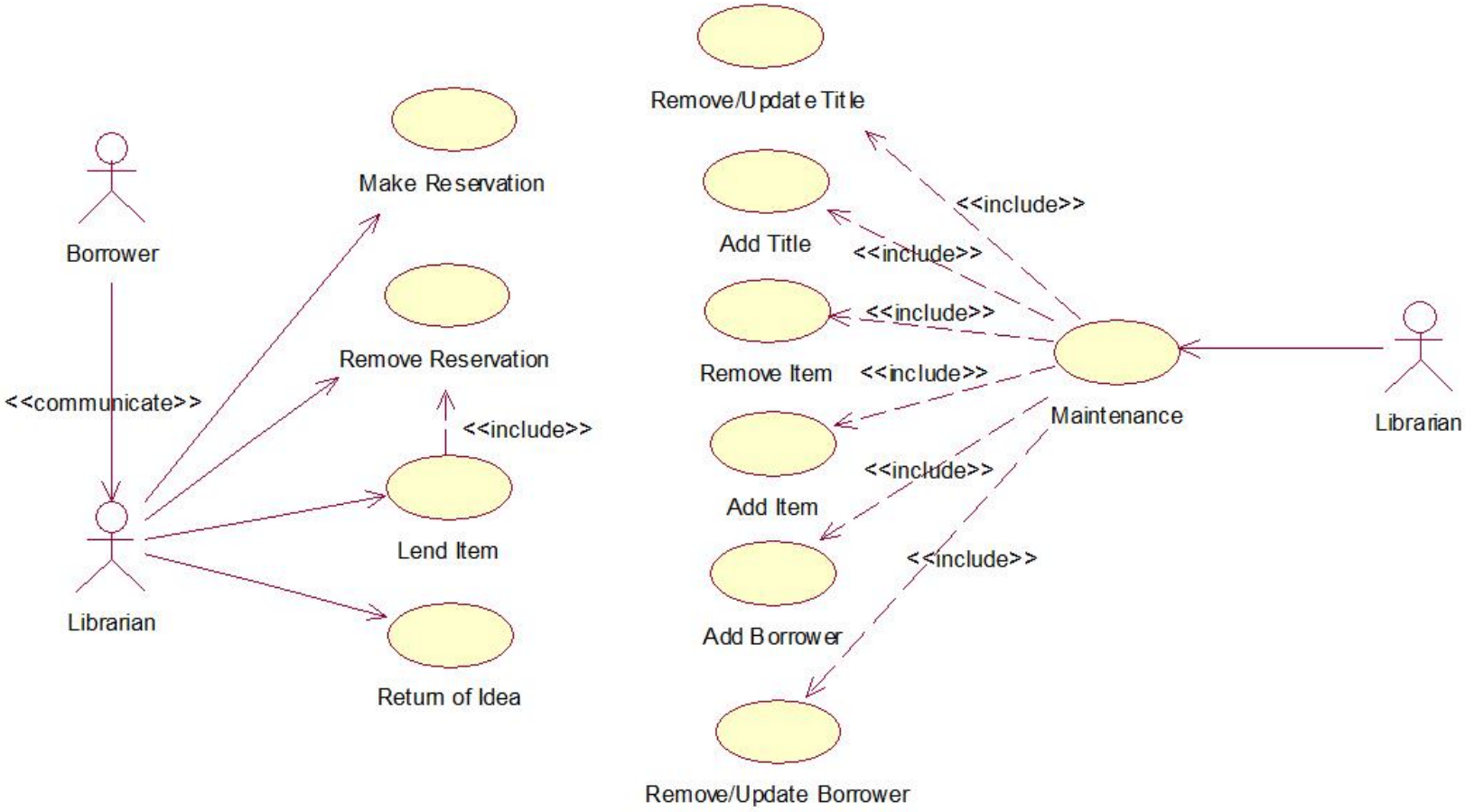
Interaction Model

Composed of four elements:

- Use-cases
- Sequence diagrams
- State diagrams
- User interface prototype

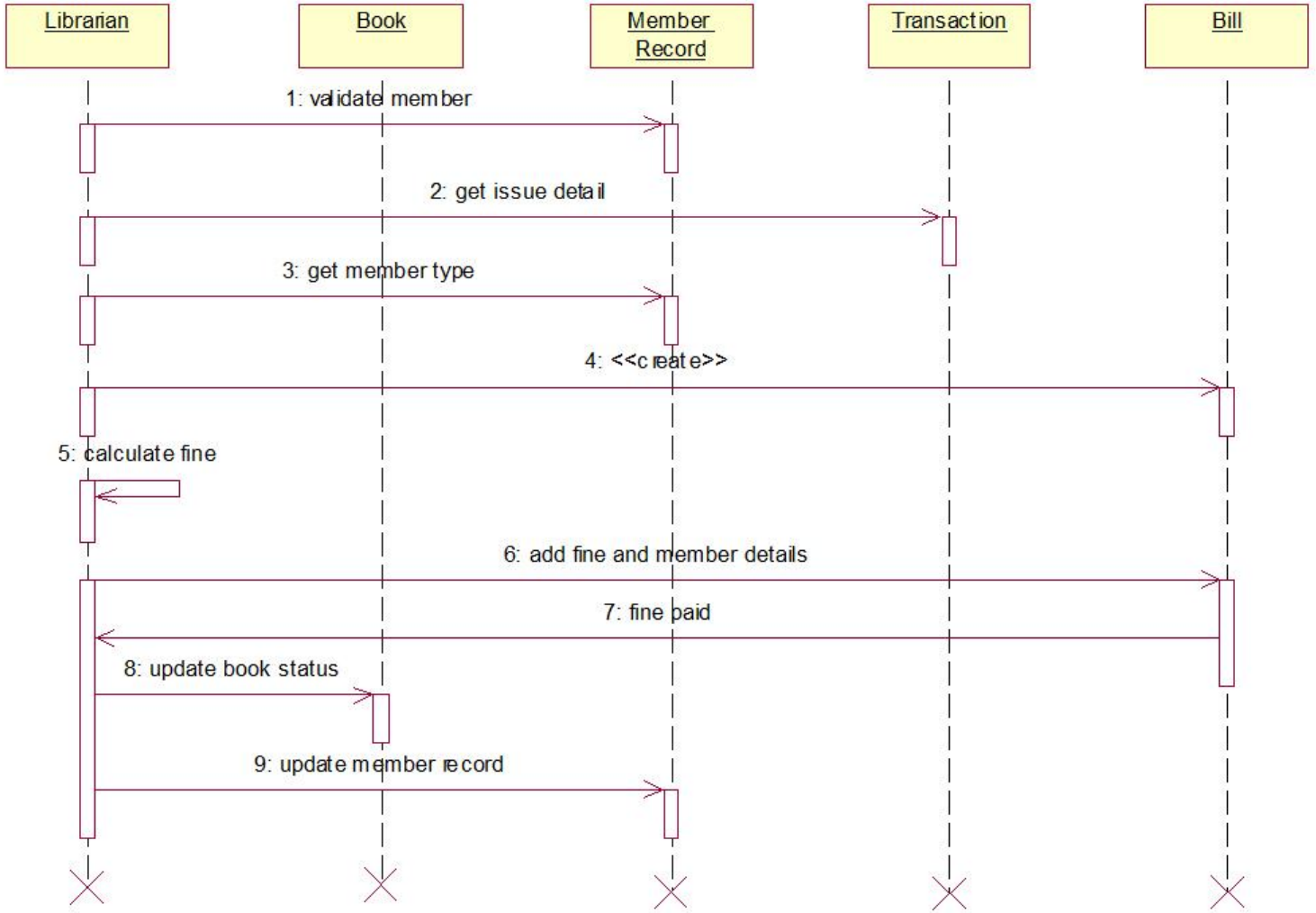
- https://www.startertutorials.com/uml/uml-diagrams-library-management-system.html

Use case diagram



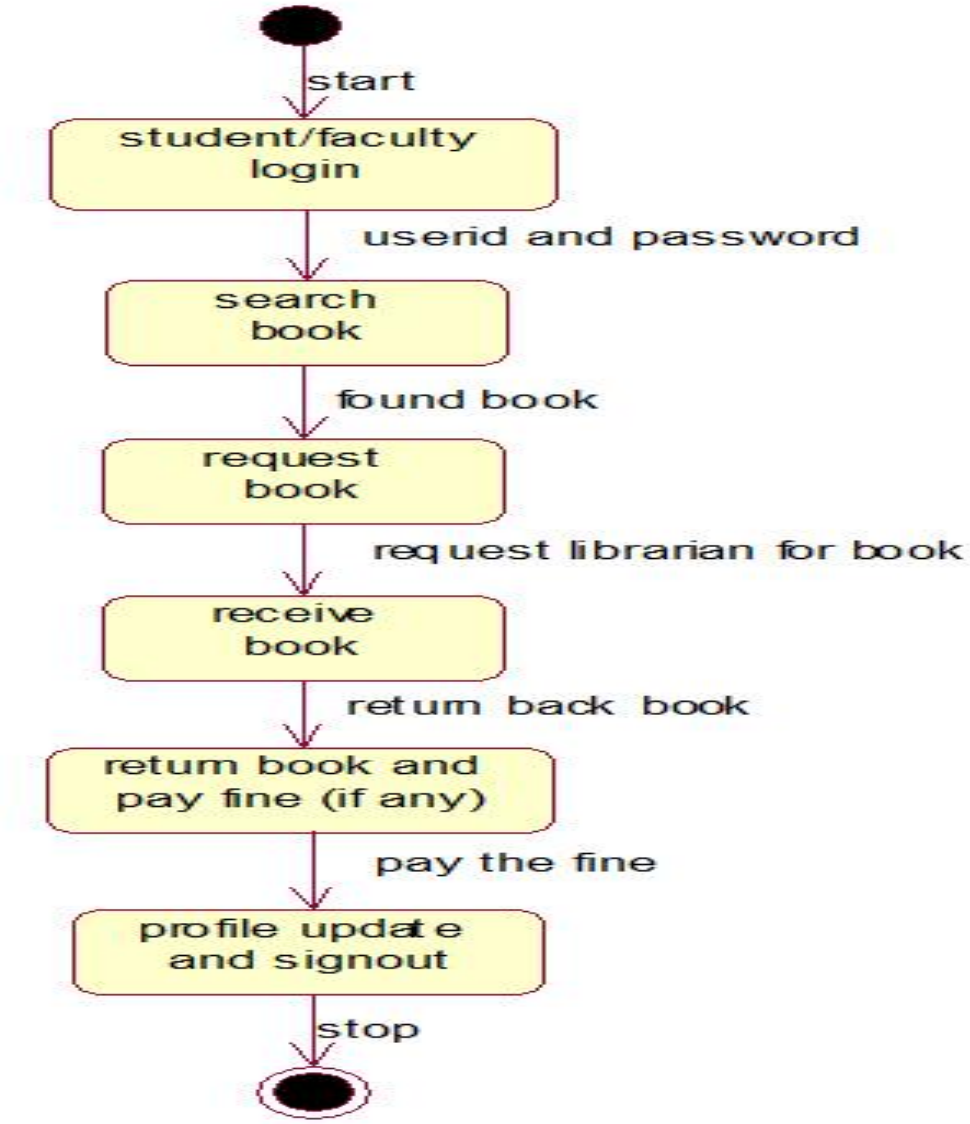
Web Application Design

Interaction Model - Sequence diagram



Web Application Design

Interaction Model - State diagram



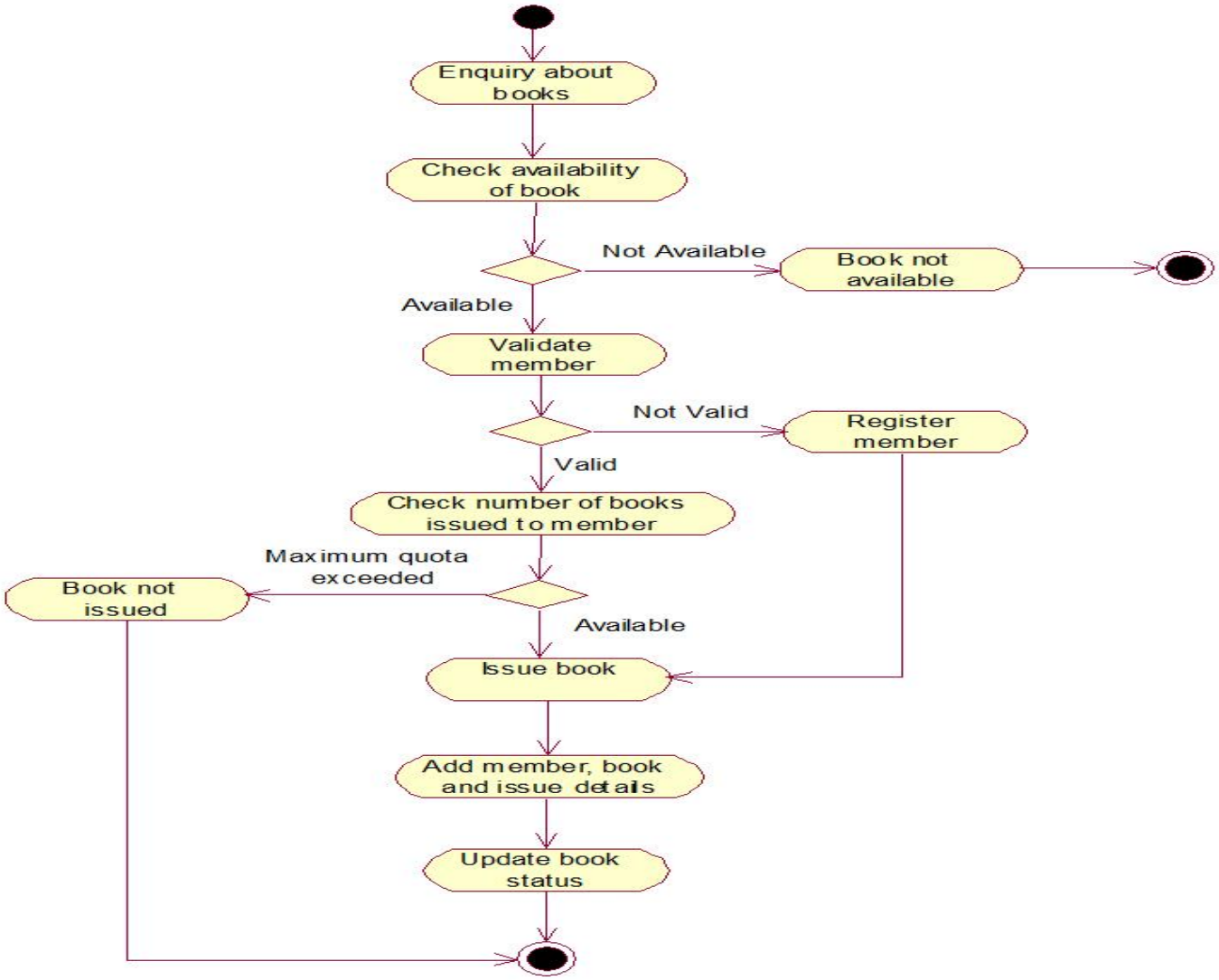
Web Application Design

Functional Model

- The functional model addresses two processing elements of the WebApp
  - **User observable functionality** that is delivered by the WebApp to end-users
  - The operations contained within analysis classes that implement **behaviors associated with the class**.
- An activity diagram can be used to represent processing flow

Web Application Design

Functional Model - Activity diagram



Web Application Design

Configuration Model

- Server-side
  - Server hardware and operating system environment must be specified
  - Interoperability considerations on the server-side must be considered
  - Appropriate interfaces, communication protocols and related collaborative information must be specified

- Client-side
  - Browser configuration issues must be identified
  - Testing requirements should be defined

Web Application Design

Navigation Modeling

- Should certain elements be easier to reach (require fewer navigation steps) than others? What is the priority for presentation?
- Should certain elements be emphasized to force users to navigate in their direction?
- How should navigation errors be handled?
- Should navigation to related groups of elements be given priority over navigation to a specific element.
- Should navigation be accomplished via links, via search-based access, or by some other means?
- Should certain elements be presented to users based on the context of previous navigation actions?
- Should a navigation log be maintained for users?

Web Application Design

Navigation Modeling

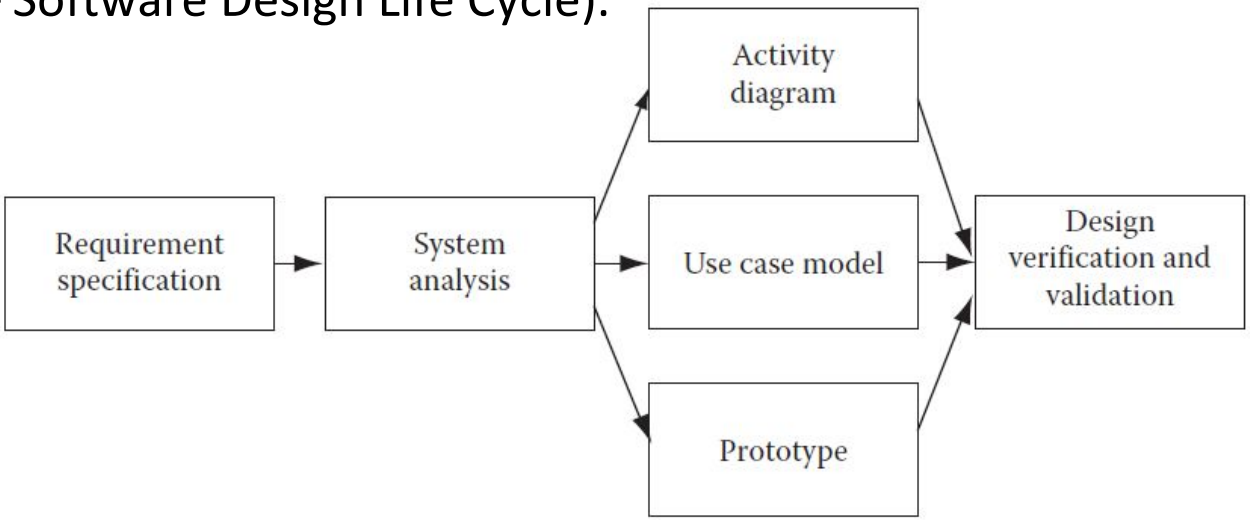
- Should a full navigation map or menu (as opposed to a single “back” link or directed pointer be available at every point in a user’s interaction?
- Should navigation design be driven by the most commonly expected user behaviors or by the perceived importance of the defined WebApp elements?
- Can a user “store” his previous navigation through the WebApp to expedite future usage?
- For which user category should optimal navigation be designed?
- How should links external to the WebApp be handled, whether it is by overlaying the existing browser window or as a new browser window or as a separate frame?

Concurrent Engineering in Software Design

- Concurrent engineering deals with taking advance information from an earlier stage for a later stage in project, so that both the stages can be performed simultaneously.
- Though project activities are planned ahead in time, most often there are dependencies between a previous task and the next task in line.
- So, the latter task cannot start until the previous task finishes.
- That is why it is significant to be aware that it is not feasible to initiate developing an application until its design is complete.
- Moreover, the development will depend on the design.
- Until all details about design are made, development cannot be started.
- So, the development team cannot start their job until they have a software design in their hands.
- Still some aspects about latter tasks can be done in advance.
- For instance, what development language will be used and how the application can be partitioned for development work can be decided at the design stage itself.
- Similarly, how maintenance and support functions will be done for the application can be determined at the design stage itself.
- Knowing in advance helps in taking care of issues that may arise in later stages.

Design Life-Cycle Management

- Software requirements go through design process steps to become a full-fledged software design.
- At the high level, system analysis is performed.
- System analysis includes a study of requirements and finding feasibility of converting them into software design.
- Once the feasibility is done, then the actual software design is made.
- The software design is in the form of activity diagrams, use cases, prototypes, etc.
- Once the design process is complete, these design documents are verified and validated through design reviews.
- Once the design is reviewed and approved, then the design phase is over (Figure below – Software Design Life Cycle).



REFERENCES

- Ashfaque Ahmed, Software Project Management: A Process-driven approach, Boca Raton, Fla: CRC Press, 2012

- Roger S. Pressman, Software Engineering – A Practitioner Approach, 6th ed., McGraw Hill, 2005

- Ian Somerville, Software Engineering, 8th ed., Pearson Education, 2010

THANK YOU