SOFTWARE ENGINEERING

UNIT - 2

Chapter-1: Principles that guide Practice
- Core Principles of Modeling

Chapter-2: Understanding Requirements
- Requirements Engineering
- Establishing the Groundwork
- Eliciting Requirements
- Developing Use Cases
- Building the Analysis Model

Chapter-3: Requirements Modeling: Scenario-based Methods:-
- Requirements Analysis
- Scenario Based Modeling
- UML Models That Supplement the Use Case

Chapter 4: Requirements Modeling: Class based methods
- Identifying Analysis Classes
- Specifying Attributes
- Defining Operations
- Class-Responsibility-Collaborator Modeling
- Associations and Dependencies
- Analysis Packages

Chapter 5 - Design Conce
- Design within the Conte
- Design Process
- Design Concepts
- Design Model

Chapter 6 - Architectural
- Software Architecture
- Architectural Styles
- Architectural Considera
- Architectural Design

Chapter 7 - Component -I
- What is a Component
- Designing Class-Based C
- Conducting Component
- Component-Based Deve

Chapter 8 - User Interfac
- User Interface Design R
  The Golden Rules

Chapter-1: Principles that guide Practi

- Core Principles of Modeling

Chapter-1: Principles that guide Practic

Software Engineering Knowledge:

- Many software practitioners think of software engineering knowled exclusively as knowledge of specific technologies: Java, Perl, html, C- Windows NT, and so on.
- Knowledge of technology is necessary to perform computer progra
- software engineering knowledge had evolved to a "stable core" that represented about "75 percent of the knowledge needed to develop system."

CORE PRINCIPLES

- Software engineering is guided by a collection of core principles the application of a software process and the execution of effective engineering methods.
- Core principles establish a collection of values and rules that serv as you analyze a problem, design a solution, implement, test the s ultimately deploy the software in the user community.
- A set of general principles of software engineering process and pra
  - (1) provide value to end users,
  - (2) keep it simple,
  - (3) maintain the vision (of the product and the project),
  - (4) recognize that others consume (and must understand) what you
  - (5) be open to the future,
  - (6) plan ahead for reuse, and
  - (7) think!

Principles That Guide Process

The following set of core principles can be applied to the framework, a extension, to every software process:

Principle 1. Be agile

Principle 2. Focus on quality at every step.

Principle 3. B e ready to adapt.

Principle 4. Build an effective team.

Principle 5. Establish mechanisms for communication and coor

Principle 6. Manage change.

Principle 7. Assess risk.

Principle 8. Create work products that provide value for others.

7.2.2 Principles That Guide Practice

- Software engineering practice has a single overriding goal—to deliv **high-quality**, **operational software** that contains functions and fe meet the needs of all stakeholders.
- To achieve this goal, a set of core principles that guide technica necessary
- The following set of core principles are fundamental to the practice engineering:

Principle 1. ***Divide and conquer*** **-** Stated in a more technical manner, a design should always emphasize *separation of concerns (SoCs).*

- *A large* problem is easier to solve if it is subdivided into a collection of
- *Ideally, each concern delivers distinct functionality that can be* develo some cases validated, independently of other concerns.

Principle 2. ***Understand the use of abstraction*** **-** *At its core, an abst* simplification of some complex element of a system, used to communica in a single phrase.

7.2.2 Principles That Guide Practice

Principle 3. Strive for consistency - Whether it's creating an ana
developing a software design, generating source code, or creating tes
principle of consistency suggests that a familiar context makes softwa
use.

Principle 4. Focus on the transfer of information - Softwar
information transfer—from a database to an end user, from a legacy sy
WebApp, from an end user into a graphic user interface (GUI), from an
system to an application, from one software component to another

Principle 5. Build software that exhibits effective modularity – S
concerns establishes a philosophy for software. Modularity provides a
for realizing the philosophy.

- Any complex system can be divided into modules (components
  software engineering practice demands more.
- Modularity must be effective .

7.2.2 Principles That Guide Practice

Principle 6. Look for patterns - The use of design patterns can b
wider systems engineering and systems integration problems, l
components in complex systems to evolve independently

Principle 7. When possible, represent the problem and its solutio
a number of different perspectives. When a problem and its solution a
from a number of different perspectives, it is more likely that greater
be achieved and that errors and omissions will be uncovered.

Principle 8. Remember that someone will maintain the softwa
long term, software will be corrected as defects are uncovered, ad
environment changes, and enhanced as stakeholders request more cap

Chapter-2: Understanding Requiremer

Chapter-2: Understanding Requirements:-

- Requirements Engineering
- Establishing the Groundwork
- Eliciting Requirements
- Developing Use Cases
- Building the Analysis Model

Chapter-2: Understanding Requirements:-

REQUIREMENTS ENGINEERING -

- The broad spectrum of tasks and techniques that l understanding of requirements is called ***requirements engi***
- RE must be adapted to the needs of the process, the p product, and the people doing the work.
- RE builds a bridge to design and construction.
- RE establishes a solid base for design and construction.
- RE encompasses seven distinct tasks:
  inception, elicitation, elaboration, negotiation, specification, and management.
- It is important to note that some of these tasks occur in para are adapted to the needs of the project.

Requirements Engineering

Inception. How does a software project get started

Elicitation. It certainly seems simple enough

- Ask the customer, the users, and others
  - what the objectives for the system or product are?
  - what is to be accomplished?
  - How the system or product fits into the needs of the business?
  - Finally, how the system or product is to be used on a day-to-day basis?
- An important part of elicitation is to establish **business goals.**

Goal-Oriented Requirements Engineering

- A ***goal*** *is a long-term aim that a system or* product must achieve.
- **Goals** may deal with either functional or nonfunctional concerns.
- Goals are often a good way to explain requirements to stakeholders
- *Elaboration is a good thing, but you have to know when to stop.*
- *The key is to describe the problem in a way that establishes a firm base for*
- *If you work beyond that point, you're doing design.*

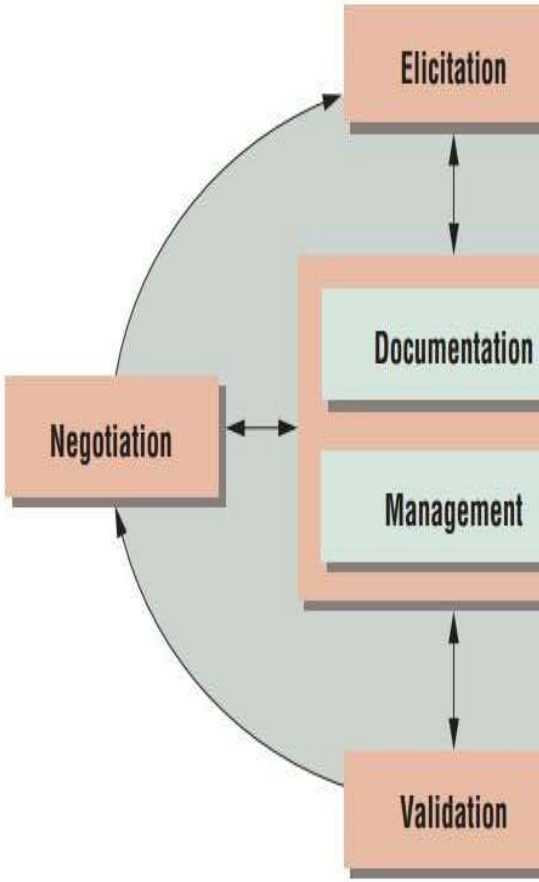
Requirements Engineering

Negotiation:

- It isn't unusual for customers and users to ask for more than can be achieved, given limited business resources.

Specification:

- In the context of computer-based systems (and software), the term ***specification*** *means different things to different people.*
- *A* ***specification*** *can* be a written document, a set of graphical models, a formal mathematical model, a collection of usage scenarios, a prototype, or any combination of these.
- A "**standard template**" should be developed and used for a specification to represent in understandable manner



Requirements Engineering

Table: Software Requirements Specification Template

Software Requirements Specification Template

A *software requirements specification* (SRS) is a work product that is created when a detailed description of all aspects of the software to be built must be specified before the project is to commence. It is important to note that a formal SRS is not always written. In fact, there are many instances in which effort expended on an SRS might be better spent in other software engineering activities. However, when software is to be developed by a third party, when a lack of specification would create severe business issues, or when a system is extremely complex or business critical, an SRS may be justified.

Karl Wiegers [Wie03] of Process Impact Inc. has developed a worthwhile template (available at **www.processimpact.com/process_assets/srs_template.doc**) that can serve as a guideline for those who must create a complete SRS. A topic outline follows:

Table of Contents

Revision History

1. **Introduction**
   - 1.1 Purpose
   - 1.2 Document Conventions
   - 1.3 Intended Audience and Reading Suggestions
   - 1.4 Project Scope
   - 1.5 References
2. **Overall Description**
   - 2.1 Product Perspective
   - 2.2 Product Features
   - 2.3 User Classes and Charac
   - 2.4 Operating Environment
   - 2.5 Design and Implementati
   - 2.6 User Documentation
   - 2.7 Assumptions and Depend
3. **System Features**
   - 3.1 System Feature 1
   - 3.2 System Feature 2 (and so
4. **External Interface Requir**
   - 4.1 User Interfaces
   - 4.2 Hardware Interfaces
   - 4.3 Software Interfaces
   - 4.4 Communications Interface
5. **Other Nonfunctional Req**
   - 5.1 Performance Requirement
   - 5.2 Safety Requirements
   - 5.3 Security Requirements
   - 5.4 Software Quality Attribut
6. **Other Requirements**

Appendix A: Glossary
Appendix B: Analysis Models
Appendix C: Issues List

A detailed description of each SRS tained by downloading the SRS tem noted in this sidebar.

Requirements Engineering

Validation.

- The work products produced as a consequence of requirements engine assessed for quality during a validation step.
- Requirements validation examines the specification to ensure that all so requirements have been stated unambiguously and the work products c the standards established for the process, the project, and the product.
- *A key concern during requirements validation is* ***consistency.***

Use the analysis *model to ensure that requirements have been consistentl*

- The primary requirements validation mechanism is the technical review.
- The review team that validates requirements includes software customers, users, and other stakeholders.

Requirements Engineering

Requirements management-

Objective:

Requirements engineering tools assist in requirements gathering, modeling, requirements management, and requirements validation.

Mechanics:/Tool mechanics

In general, requirements engineering tools build a variety of graphical (e.g., that depict the informational, functional, and behavioral aspects of a system. These models form the basis for all other activities in the software process.

Representative Tools:

A reasonably comprehensive (and up-to-date) listing of requirements engine are necessary

ESTABLISHING THE GROUNDWORK

The steps required for understanding of software requirements for project starting & toward a successful solution are:-

Identifying Stakeholders: *stakeholder as "anyone who benefits* in a direct or indire system, which is being developed"

Recognizing Multiple Viewpoints - the requirements of the system will be explo different points of view.

There are several things that can make it hard to elicit requirements :

- project goals are unclear,
- stakeholders' priorities differ,
- people have unspoken assumptions,
- stakeholders interpret meanings differently, and
- requirements are stated in a way that makes them difficult to verify.

The goal of effective requirements engineering is to eliminate or at least reduce these

8.2.3 Working toward Collaboration

If five stakeholders are involved in a software project, you may have five different op proper set of requirements.

ESTABLISHING THE GROUNDWORK

Asking the First Questions

Questions asked at the inception of the project should be "context free".

The context-free questions focuses on the customer and other stakeholder project goals and benefits. For ex:-

- Who is behind the request for this work?
- Who will use the solution?
- What will be the economic benefit of a successful solution?
- Is there another source for the solution that you need?

Nonfunctional Requirements

- A ***nonfunctional requirement (NFR)*** *can be described as a* ***quality*** **performance attribute**, **a security attribute**, or **a general constraint** on a s
- These are often not easy for stakeholders to articulate.
- Quality function deployment attempts to translate unspoken customer needs system requirements.
- Nonfunctional requirements are often listed separately in a software specification.

ESTABLISHING THE GROUNDWORK

- Table lists **NFRs as *column labels*** *and* ***software engineering guidelines as***
- *A* ***relationship*** **matrix** compares each guideline to all others, helping the te

whether each pair of guidelines is *complementary , overlapping , co* *independent .*

Traceability

- ***Traceability*** *is a software engineering term that refers to documented l* software engineering work products (e.g., requirements and test cases).
- **A *traceability matrix*** *allows a requirements engineer to represent the* between requirements and other software engineering work products.
- **Rows** of the traceability matrix are labeled using requirement names and be labeled with the name of a software engineering work product
- A **matrix cell** is marked to indicate the presence of a link between the two.
- The **traceability matrices** can support a variety of engineering developmen
- They can provide continuity for developers as a project moves from one pro another.

ELICITING REQUIREMENTS

- **Requirements elicitation** (also called requirements gathering) combines problem solving, elaboration, negotiation, and specification.
- In order to encourage a collaborative, team-oriented approach to requiremen stakeholders work together to identify the problem, propose elements of negotiate different approaches, and specify a preliminary set of solution Requi

Collaborative Requirements Gathering:

- Many different approaches to collaborative requirements gathering have been
- Each makes use of a slightly different scenario, but all apply some vari following basic guidelines:
- Meetings are conducted and attended by both software engineers and other st
- Rules for preparation and participation are established.
- An agenda is suggested that is formal enough to cover all important points enough to encourage the free flow of ideas.
- A "facilitator" controls the meeting.
- A "definition mechanism" is used.

Collaborative Requirements Gathering

- Joint Application Development (JAD) is a popular technique for r gathering.
- The goal is to identify the problem, propose elements of the soluti different approaches, and specify a preliminary set of solution requiremer
- If a system or product will serve many users, be absolutely certain that r are elicited from a representative cross section of users. If only one us requirements, acceptance risk is high.

Collaborative Requirements Gathering

- **QFD** defines requirements in a way that maximizes customer satisfac
- Everyone wants to implement lots of exciting requiremen requirements lead to a breakthrough product!
- **Quality function deployment** (QFD) is a quality management tec translates the needs of the customer into technical requirements for
- **QFD** "concentrates on maximizing customer satisfaction from th engineering process".
- **QFD** emphasizes an understanding of what is valuable to the cu then deploys these values throughout the engineering process.
- If these requirements are present, the customer is satisfied.
- Their absence will be a cause for significant dissatisfaction.
- **Exciting requirements** go beyond the customer's expectations and very satisfying when present.

Collaborative Requirements Gathering

Usage Scenarios

- As requirements are gathered, an overall vision of system functions and f to appear.
- Developers and users can create a set of scenarios that identify a thread of system to be constructed.
- The scenarios, often called ***use cases,*** *provide a* description of how the s used.

Elicitation Work Products

The work products produced as a consequence of requirements elicitat depending on the size of the system or product to be built.

For most systems, the work products include:

(1) a statement of need and feasibility,

(2) a bounded statement of scope for the system or product,

(3) a list of customers, users, and other stakeholders who participated in elicitation,

Collaborative Requirements Gathering

(4) a description of the system's technical environment,

(5) a list of requirements (preferably organized by function) and the domain co applies to each,

(6) a set of usage scenarios that provide insight into the use of the system or p different operating conditions, and

(7) any prototypes developed to better define requirements. Each of these wor reviewed by all people who have participated in requirements elicitation.

Collaborative Requirements Gathering

Agile Requirements Elicitation

- Within the context of an agile process, requirements are elicited b stakeholders to create user stories .
- Each user story describes a simple system requirement written fron perspective.
- User stories can be written on small note cards, making it easy for develop and manage a subset of requirements to implement for the next product in

Service-Oriented Methods

- **Service-oriented development** views a system as an aggregation of servic
- A **service** can be "as simple as providing a single function, for example **response-based mechanism** that provides a series of random numbers, aggregation of complex elements, such as the **Web service API**".
- **Requirements elicitation** for service-oriented methods fines services n app.
- A **touch point** represents an opportunity for the user to interact with t receive a desired service.

DEVELOPING USE CASES

- **Use cases** are defined from an actor's point of view.
- An **actor** is a role that people or devices play as they interact with the so
- The first step in writing a use case is to define the set of "actors".
- ***Actors are the different people (or devices)*** *that use the* system or pro the context of the function and behavior that is to be described.
- **Actors** represent the roles that people play as the system operates.
- An **actor** is anything that communicates with the system or product external to the system itself.
- Every **actor** has one or more goals when using the system.
- The software for the control computer requires **four different modes interaction:** programming mode, test mode, monitoring m troubleshooting mode.
- Therefore, **four actors can be defined**: programmer, tester, m troubleshooter

DEVELOPING USE CASES

- Primary actors interact to achieve required system function and derive the inter
  from the system.
- Secondary actors support the system so that primary actors can do their work.
- Once actors have been identified, use cases can be developed.
- Jacobson suggests a number of questions that should be answered by a use cas
  - Who is the primary actor, the secondary actor(s)?
  - What are the actor's goals?
  - What preconditions should exist before the story begins?
  - What main tasks or functions are performed by the actor?
  - What exceptions might be considered as the story is described?
  - What variations in the actor's interaction are possible?
  - What system information will the actor acquire, produce, or change?
  - Will the actor have to inform the system about changes in the external environment?
  - What information does the actor desire from the system?
  - Does the actor wish to be informed about unexpected changes?

Assignment: Refer to Page 150 – 154 for Case study on Use case

BUILDING THE ANALYSIS MODEL

- The intent of the **analysis model** is to provide a description of t **informational, functional,** and **behavioral domains** for a com system.
- **Elements of the Analysis Model**
- It is always a good idea to get stakeholders involved.
- One of the best ways to do this is to have each stakeholder write us describe how the software will be used.

BUILDING THE ANALYSIS MODEL

Chapter-1: Principles that guide Practice:-

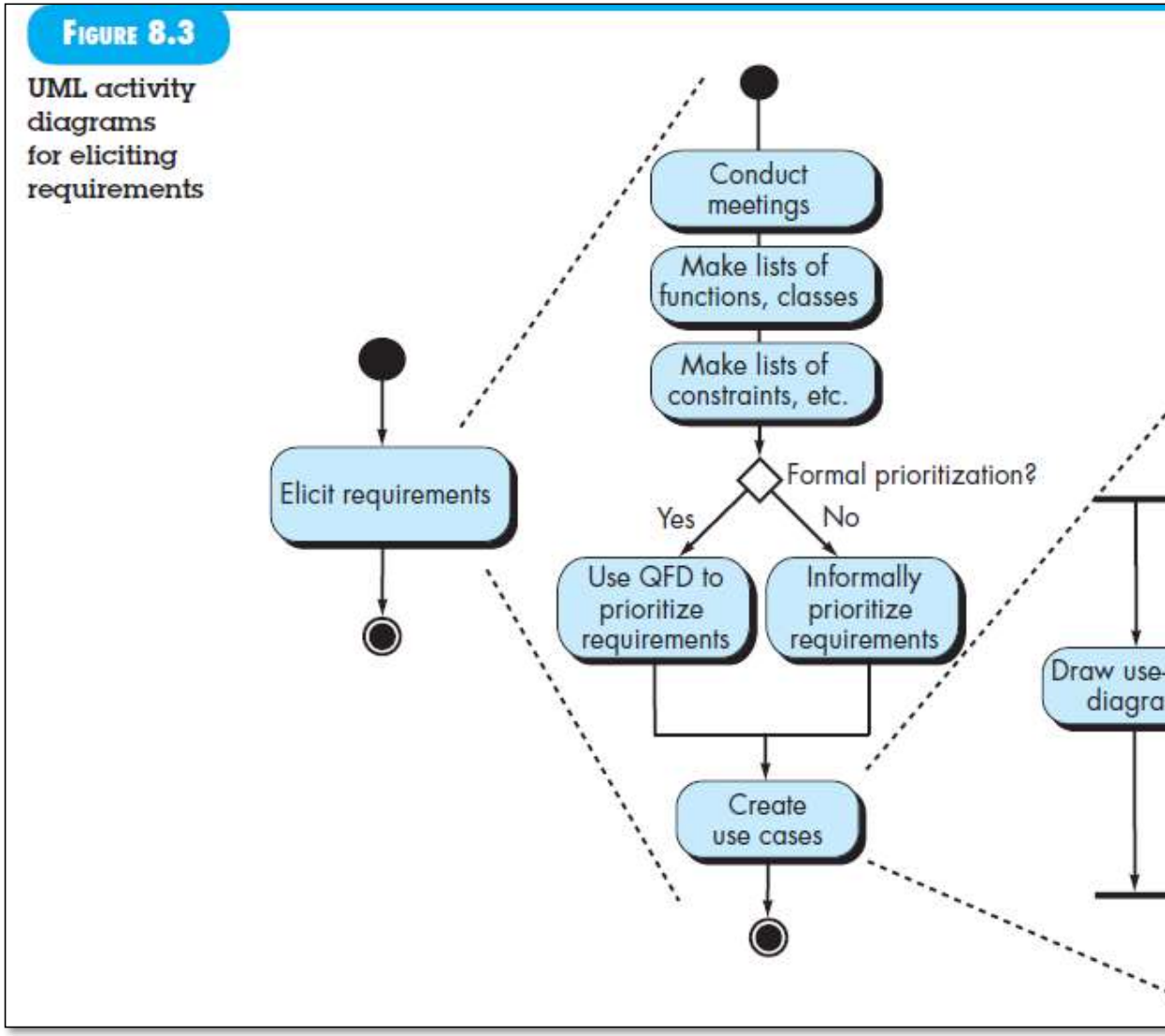


FIGURE 8.3 UML activity diagrams for eliciting requirements

BUILDING THE ANALYSIS MODEL

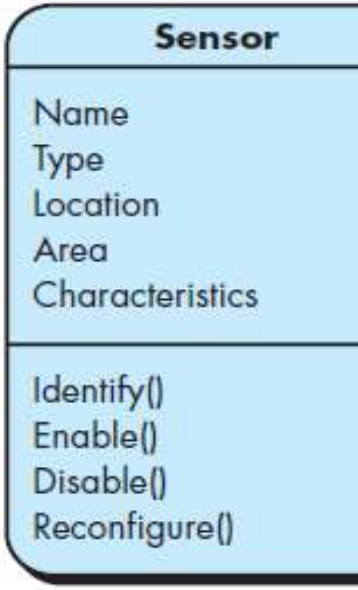
Scenario-based elements –

- Scenario-based elements of the requirements model are often the first part of the model that is developed.
- They serve as input for the creation of other modeling elements.

Class-based elements -

- Each usage scenario implies a set of objects that are manipulated as an actor interacts with the system.
- These objects are categorized into classes—a collection of things that have similar attributes and common behaviors.

Fig: Class diagram fo

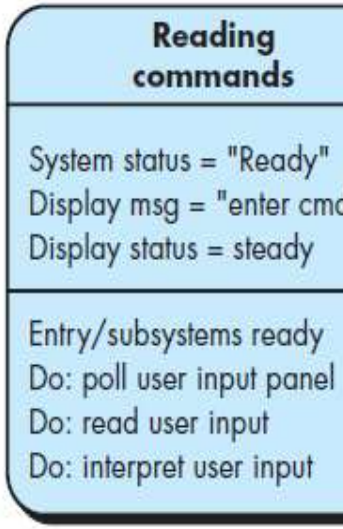


BUILDING THE ANALYSIS MODEL

Behavioral elements.

- The behavior of a computer-based system can have a deep effect on the design.
- **A state** is an externally observable mode of behavior.
- The **state diagram** is one method for representing the behavior of a system by depicting its states and the events that cause the system to change state.
- In addition, the **state diagram** indicates what actions (e.g., process activation) are taken as a consequence of a particular event.

Fig: UML State diagram



BUILDING THE ANALYSIS MODEL

Analysis Patterns

- If you want to obtain solutions to customer requirements more rapidly a your team with proven approaches, use **analysis patterns**.
- Analysis patterns suggest solutions (e.g., a class, a function, a behavior application domain that can be reused when modeling many applicatio
- Geyer-Schulz and Hahsler suggest two benefits that can be associated w of analysis patterns:-
- **First, analysis patterns** speed up the development of abstract analysis
- **Second, analysis patterns** facilitate the transformation of the analysis a design model by suggesting design patterns and reliable solutions f problems.

BUILDING THE ANALYSIS MODEL

Agile Requirements Engineering

- The intent of **agile requirements** engineering is to transfer stakeholders to the software team rather than create extensive ar products.
- The **agile process** encourages the early identification and implemen highest priority product features.
- **Agile requirements engineering** addresses important issues:
  - high requirements volatility
  - incomplete knowledge of development technology, and
  - customers not able to articulate their visions .

Requirements for Self-Adaptive Systems

- Self-adaptive systems can reconfigure themselves, augment their fu
  protect themselves, recover from failure.
- Self-adaptive systems hide most of their internal complexity from the
- Adaptive requirements document the variability needed for s
  systems.

Topics

Chapter-3: Requirements Modeling: Scenario-based Methods:-

Requirements Analysis,

Scenario Based Modeling

UML Models That Supplement the Use Case

REQUIREMENTS ANALYSIS

- Requirements analysis allows you to elaborate on basic requirements
of requirements engineering .
- The requirements modeling action results in one or more of the follow
models:
  - ✓ **Scenario-based models -** actors, use cases
  - ✓ **Class-oriented models -** object-oriented classes
  - ✓ **Behavioral and patterns-based models -** events
  - ✓ **Data models** that depict the information domain for the problem.
  - ✓ **Flow-oriented models -** functional elements, flow of data

REQUIREMENTS ANALYSIS

-> Requirements model (and the software requirements specification) pro
developer and the customer with the means to assess quality once softwa

Overall Objectives and Philosophy

The requirements model must achieve **three primary objectives**:

(1)to describe what the customer requires,

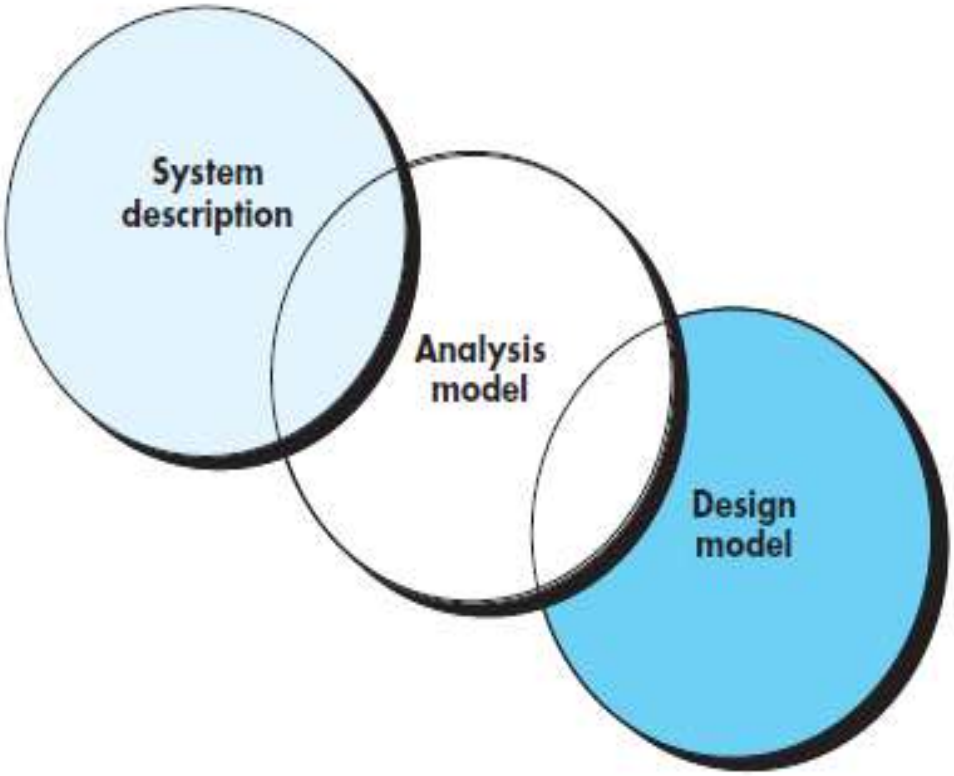
(2)to establish a basis for the creation of a software design, and

(3)to define a set of requirements that can be validated once the software

REQUIREMENTS ANALYSIS

FIGURE 9.1

The require-ments model as a bridge between the system description and the design model



Analysis Rules of Thumb

Arlow and Neustadt suggest **rules of thumb** that should be followed wh
the analysis model:

- ❖ The model should **focus on requirements** that are visible within the p
- ❖ The **level of abstraction** should be relatively **high.**
- ❖ **Each element of the requirements model** should provide insig

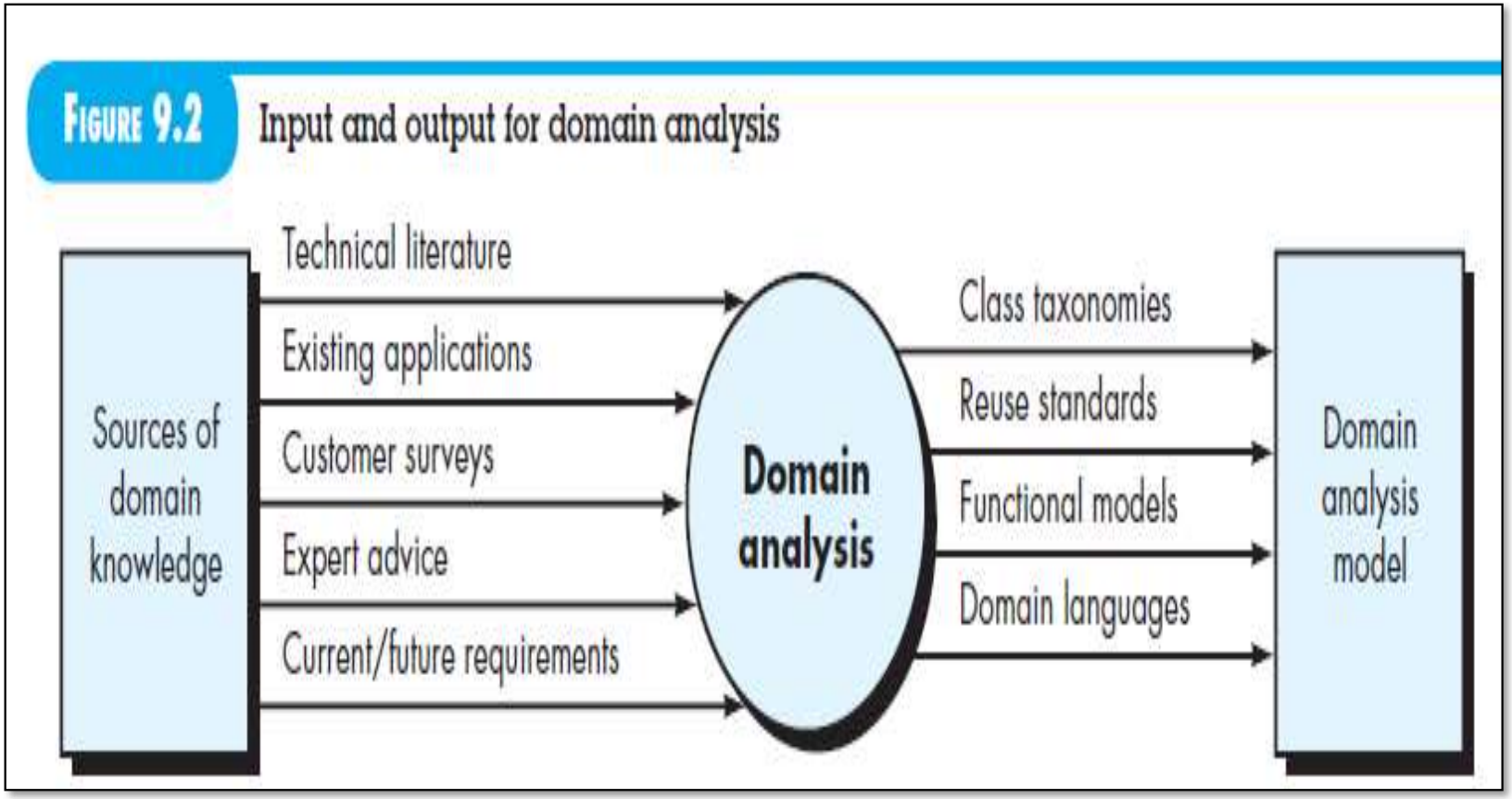
information domain, function, and **behavior** of the system.

- ❖**Delay consideration of infrastructure** until design.
- ❖**Minimize coupling** throughout the system.
- ❖Be certain that the **requirements model provides value to all stakeh**
- ❖**Keep the model as simple** as it can be.

Domain Analysis

- The intent of **Domain Analysis** is **to identify common problem elements for** applications within the domain.
- **Software domain analysis** is the **identification, analysis, and spe of common requirements** from a specific application domain.
- **Object-oriented domain analysis** is the **identification, analy specification of common, reusable capabilities** in terms of **objects, classes, subassemblies, and frameworks.**

Domain Analysis



Requirements Modeling Approaches

Structured analysis, considers data and the processes that transform separate entities.

- **Data objects -** defines their attributes and relationships.
- **Processes -** that manipulate data objects, as data objects flow thro system.

Object-oriented analysis, focuses on the definition of classes and th which they collaborate with one another to effect customer requireme

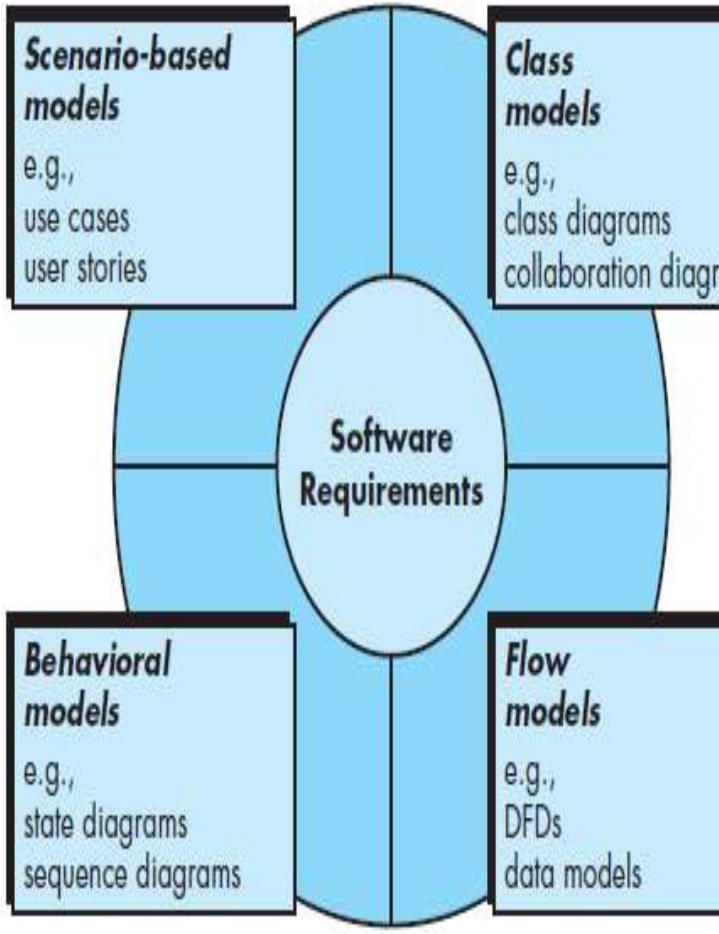
- **Scenario-based elements** depict how the user interacts with the sy the specific sequence of activities
- **Class-based elements** model the objects that the system will mani

Requirements Modeling Approaches

- **Behavioral elements** depict how external events change the state of the system or the classes that reside within it.

FIGURE 9.3

Elements of the analysis model



SCENARIO-BASED MODELING

Creating a Preliminary Use Case:

A use case captures the interactions that occur between producers and consumers of information and the system itself.

(1) What to write about?

(2) How much to write about it?

(3) How detailed to make your description? and

(4) How to organize the description?

These are the questions that must be answered if use cases are to provi as a requirements modeling tool.

SCENARIO-BASED MODELING

Refining a Preliminary Use Case:-

A description of alternative interactions is essential for understanding of the function that is being described by a use

Therefore, each step in the primary scenario is evaluated by following questions:

- Can the actor take some other action at this point?
- Is it possible that the actor will encounter some error cond point? If so, what might it be?

SCENARIO-BASED MODELING

Writing a Formal Use Case

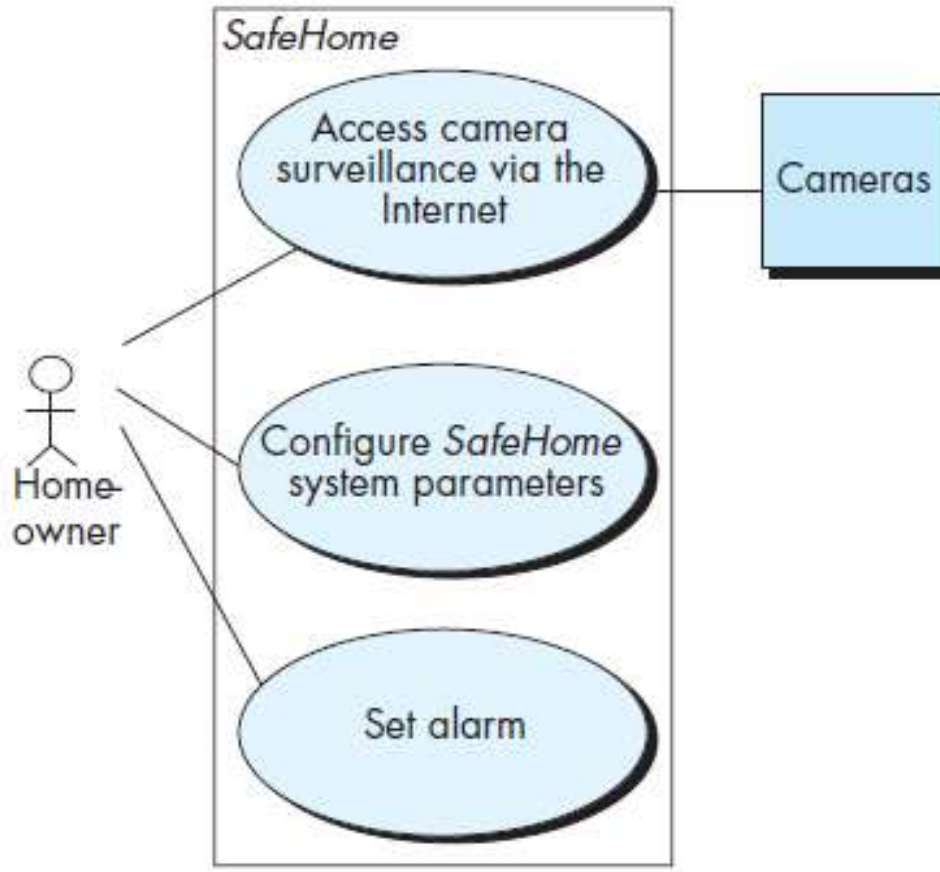
- The **informal use cases** are sometimes sufficient for req modeling.

For a Formal use case, we should consider

- ✓ The **goal** in context
- ✓ The **precondition**
- ✓ The **trigger** that identifies the event or condition
- ✓ The **scenario** that lists the specific actions
- ✓ **Exceptions**

FIGURE 9.4

Preliminary use case diagram for the *SafeHome* system



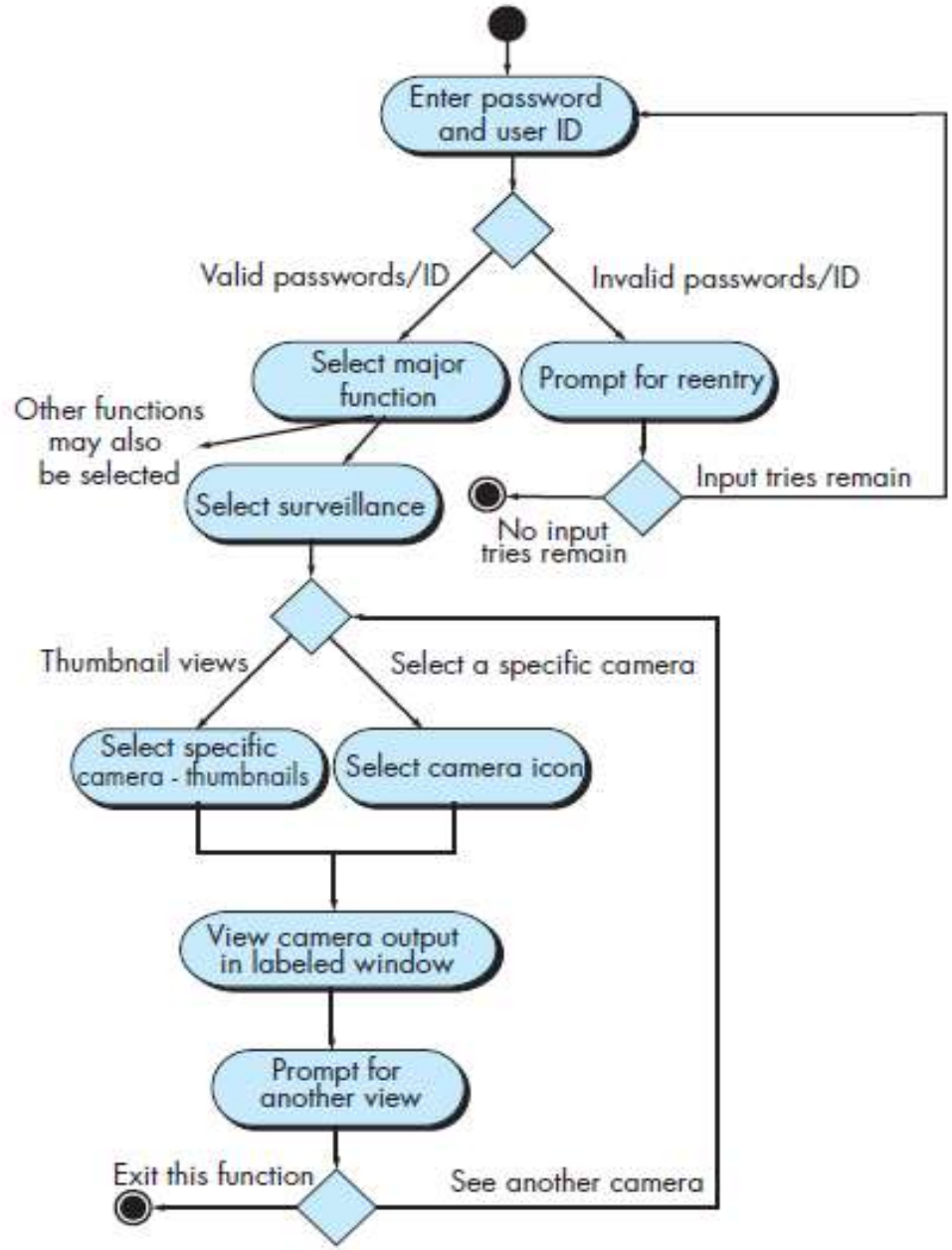
UML MODELS THAT SUPPLEMENT THE USE CASE

Developing an Activity Diagram

- The **UML activity diagram** supplements the use case by providing a representation of the flow of interaction within a specific scenario.
  - ❖ An **activity diagram** uses
    - **Rounded rectangles** to imply a specific system function,
    - **Arrows** to represent flow through the system,
    - **Decision diamonds** to depict a branching decision and
    - **Solid horizontal lines** to indicate that parallel activities are

FIGURE 9.5

Activity diagram for Access camera surveillance via the Internet—display camera views function.

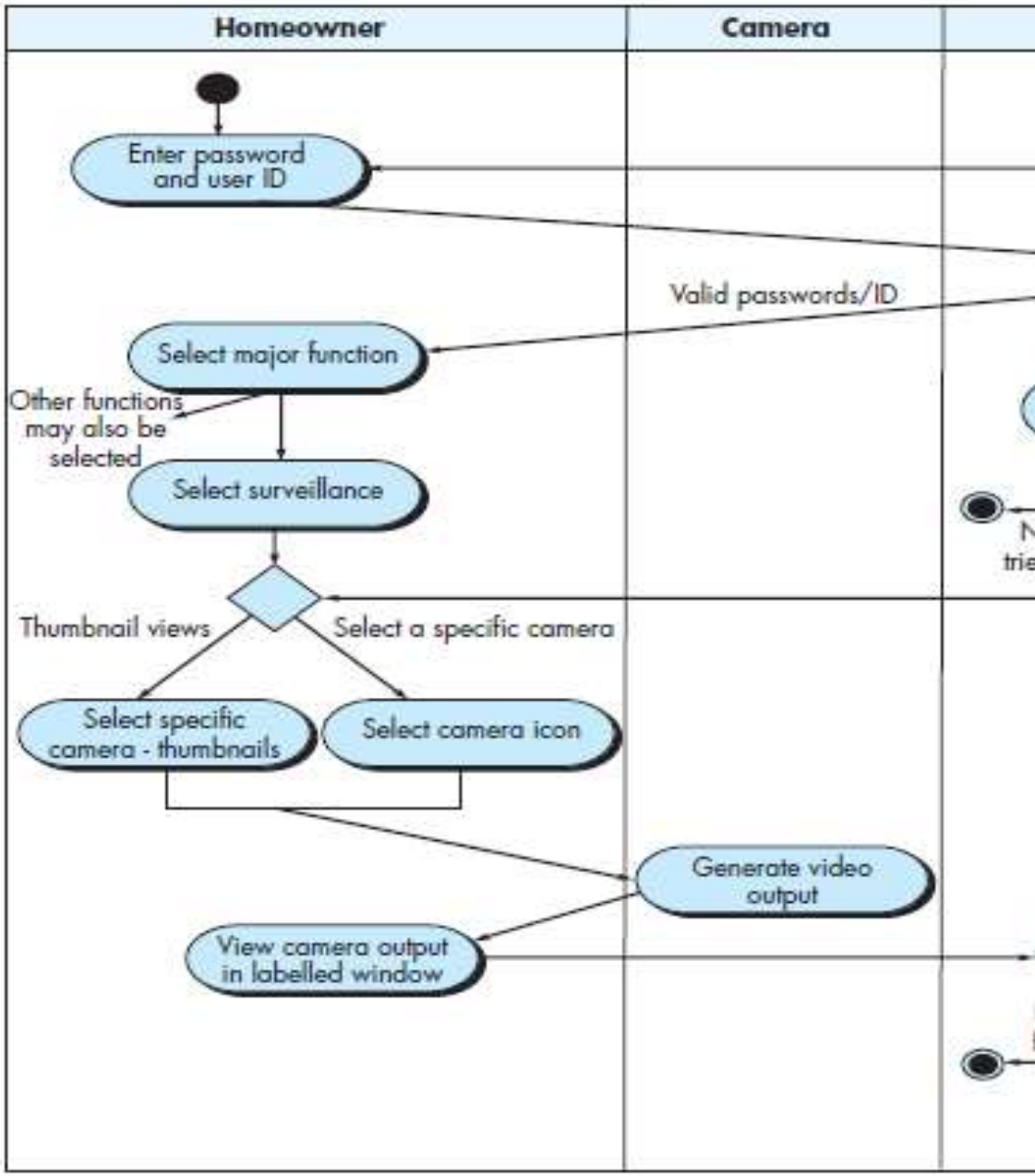


Swimlane Diagrams

- A **UML swimlane** diagram represents the flow of actions and dec indicates which actors perform each.
- The **UML *swimlane diagram*** *is a useful variation of the activity di allows* you to represent the flow of activities described by the use ca
- **Responsibilities** are represented as parallel segments that diagram vertically, like the **lanes** in a swimming pool.

Referring to Figure 9.6 , the activity diagram is rearranged so that activities associated with a particular analysis class fall inside the swimlane for that class.

FIGURE 9.6 Swimlane diagram for Access camera surveillance via the Int
views function.



Chapter 4: Requirements Modeling: Class based mo

1. Identifying Analysis Classes
2. Specifying Attributes
3. Defining Operations
4. Class-Responsibility-Collaborator Modeling
5. Associations and Dependencies
6. Analysis Packages

IDENTIFYING ANALYSIS CLASSES

- The really hard problem is discovering what are the **rig [classes].**
- **Classes**
  - **External entities** (e.g., other systems, devices, people)
  - **Things** (e.g., reports, displays, letters, signals)
  - **Occurrences or events** (e.g., a property transfer or the com series of robot movements).
  - **Roles** (e.g., manager, engineer, salesperson).
  - **Organizational units** (e.g., division, group, team)
  - **Places** (e.g., manufacturing floor or loading dock)
  - **Structures** (e.g., sensors, four-wheeled vehicles, or compute

Assignment: Refer to the table "Identifying Analysis Classes"

IDENTIFYING ANALYSIS CLASSES

- Coad and Yourdon suggest **six selection characteristics** that shou

as you consider each potential class for inclusion in the analysis mo

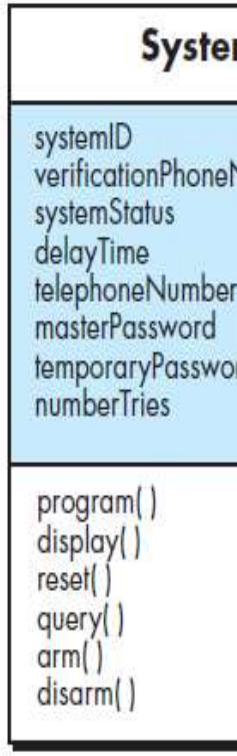
- Retained information
- Needed services
- Multiple attributes
- Common attributes
- Common operations
- Essential requirements

SPECIFYING ATTRIBUTES

- ***Attributes*** *describe a class that has been selected for inclusion in the analysis* model.
- **Attributes** are the set of data objects that fully define the class within the context of the problem.

FIGURE 10.1

Class diagram for the system class



DEFINING OPERATIONS

• **Operation**s define the behavior of an object. Operations are generall
into **four** broad categories:

(1) Operations that manipulate data in some way (e.g., adding, deletin
reformatting, selecting),

(2) Operations that perform a computation,

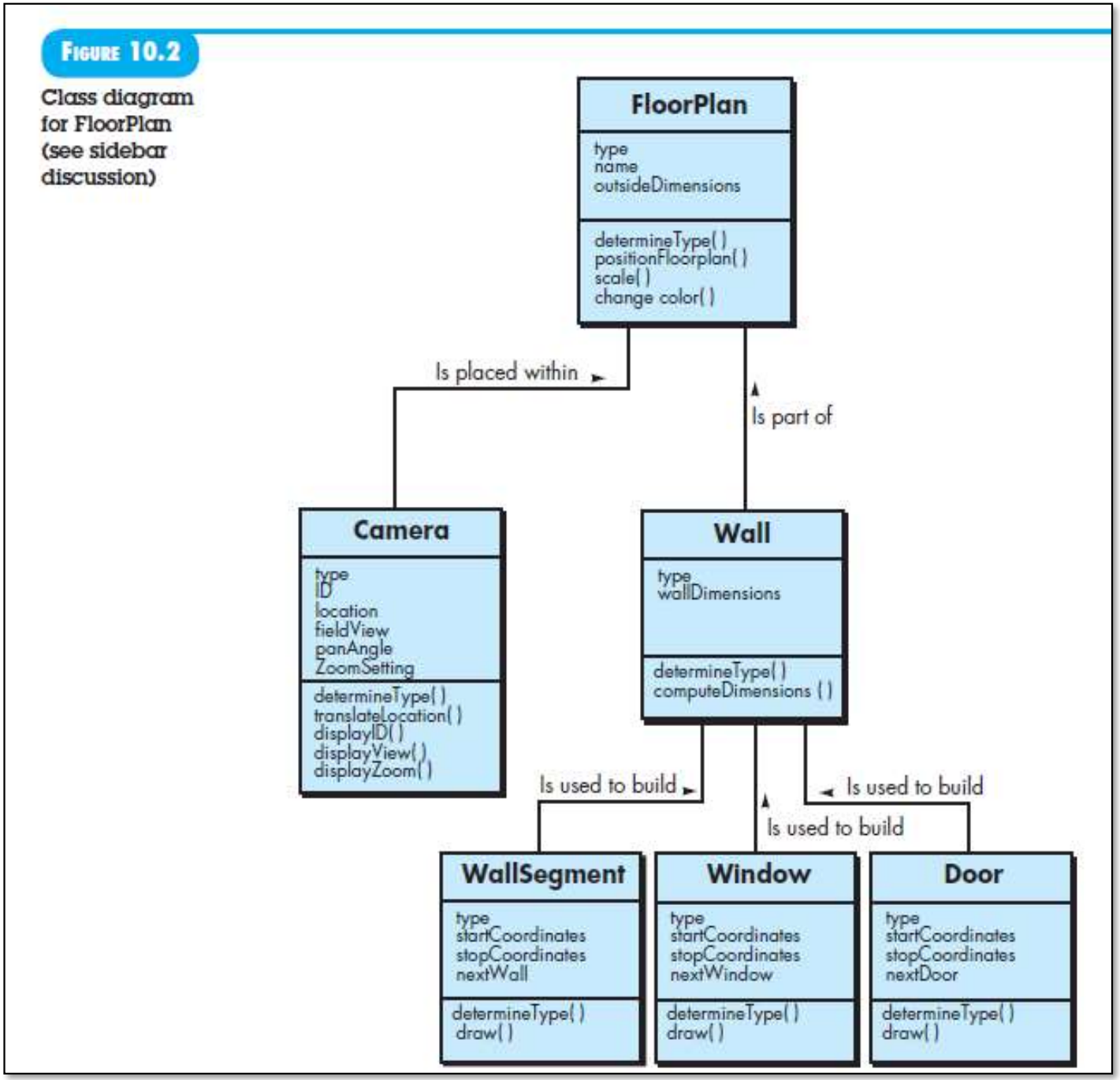
(3) Operations that inquire about the state of an object, and

(4) Operations that monitor an object for the occurrence of a controlli

Ex: sensor is assigned a number and type. Here, assign() is an operat

FIGURE 10.2

Class diagram for FloorPlan (see sidebar discussion)



CLASS-RESPONSIBILITY-COLLABORATOR MODELING

- ***Class-responsibility-collaborator (CRC) modeling*** *provides a simple* identifying and organizing the classes that are relevant to system requirements.
- A **CRC model** is really a collection of standard index cards that repres
- **Responsibilities** are the attributes and operations
- A responsibility is "**anything the class knows or does**".
- **Collaborators** are those classes that are required to provide a cla information needed to complete a responsibility.
- **Entity classes , *also called model or business classes, are extract*** from the statement of the problem.
- **Boundary classes** *are used to create the interface* that the use interacts with as the software is used.

CLASS-RESPONSIBILITY-COLLABORATOR MODELING

Controller classes (CC) manage a "**unit of work**" from start to finish. C
designed to manage :
(1) The creation or update of entity objects,
(2) The instantiation of boundary objects as they obtain information fr
objects,
(3) Complex communication between sets of objects,
(4) Validation of data communicated between objects or between the u
application.

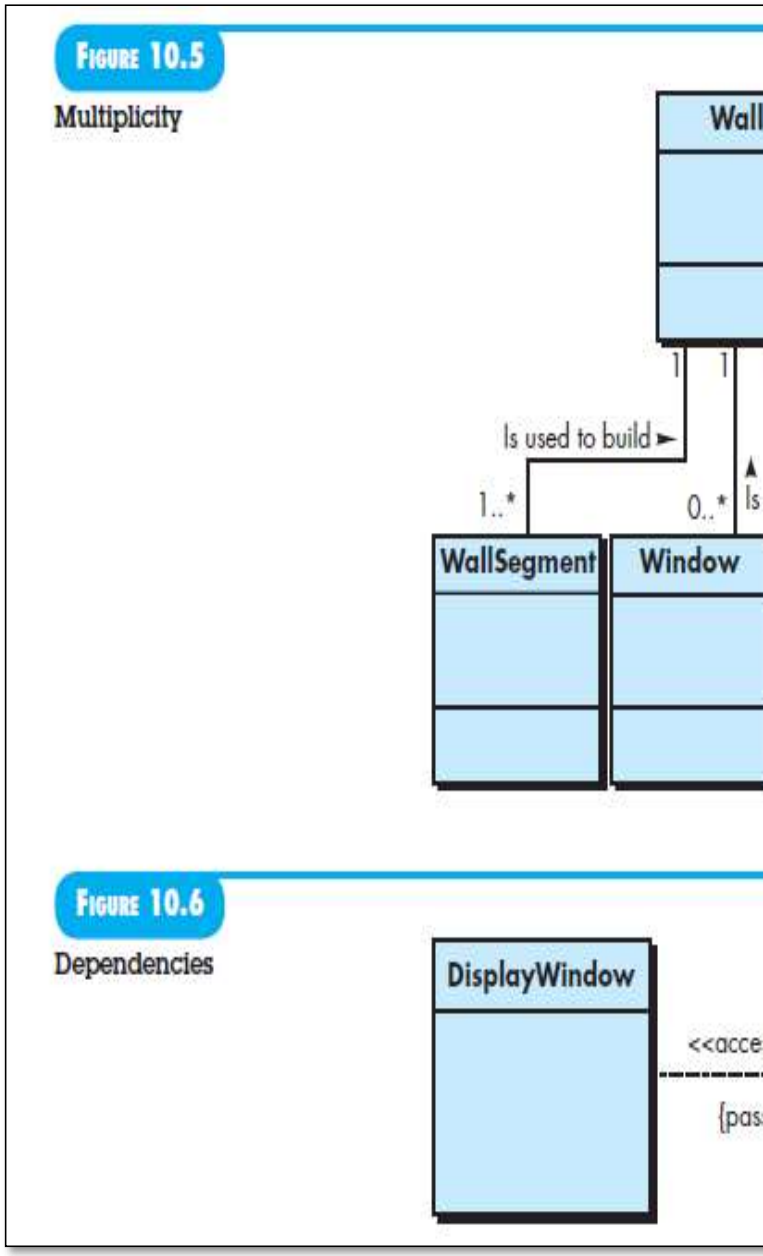
Collaborations:

Classes fulfill their responsibilities in one of two ways:

(1) A class can use its own operations to manipulate its own attributes
(2) A class can collaborate with other classes.

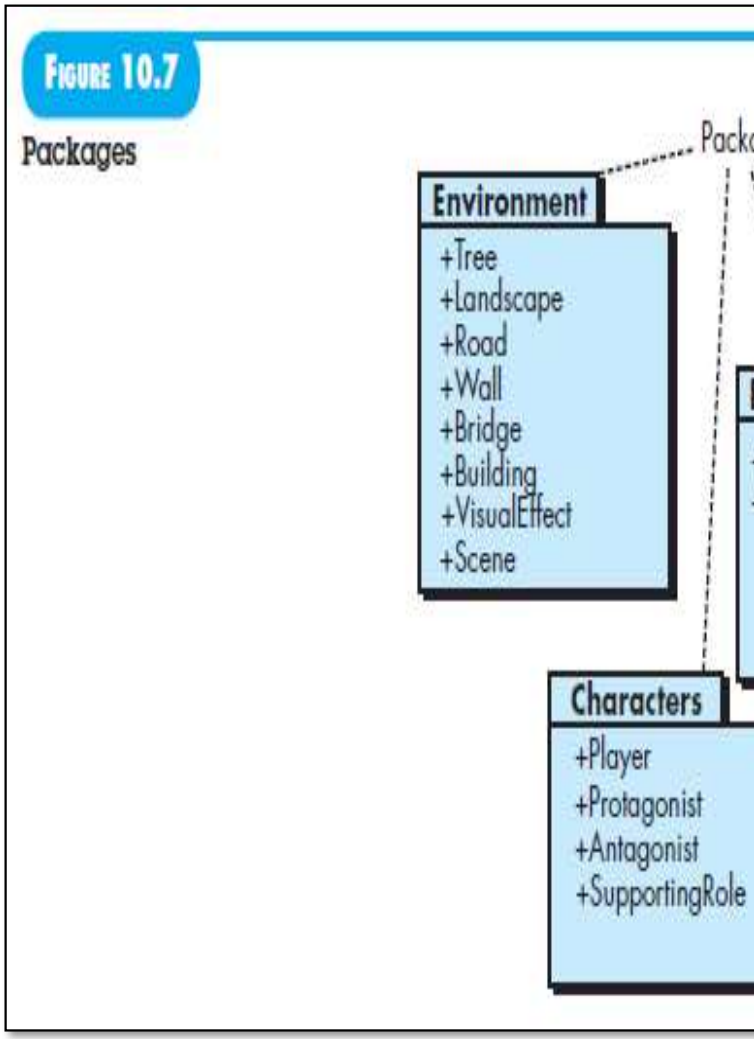
ASSOCIATIONS AND DEPENDENCIES

- An **association** defines a relationship between classes.
- **Multiplicity** defines how many of one class are related to how many of another class.
- Dependencies are defined by a **stereotype.**
- A ***stereotype*** *is an "extensibility mechanism" within UML*
- In **UML, stereotypes** are represented in double angle brackets (e.g., <<stereotype>>).



ANALYSIS PACKAGES

- A package is used to assemble a collection of related classes.
- An important part of analysis modeling is categorization.



Chapter 5 - Design Concepts

- Design within the Context of SE
- Design Process
- Design Concepts
- Design Model

DESIGN WITHIN THE CONTEXT OF SOFTWARE ENGINEERING

- "The miracle of software engineering is the transition from analysis t design to code."
- **Richard Due -** Software design changes continually as new methods, analysis, and broader understanding evolve.
- **Software design** sits at the technical kernel of software engineering
- ***Software design*** *should always begin with a consideration of data*
- "There are **two ways** of constructing a **software design.**
  - One way is to make it so simple
  - The other way is to make it so complicated

DESIGN WITHIN THE CONTEXT OF SOFTWARE ENGINEERING

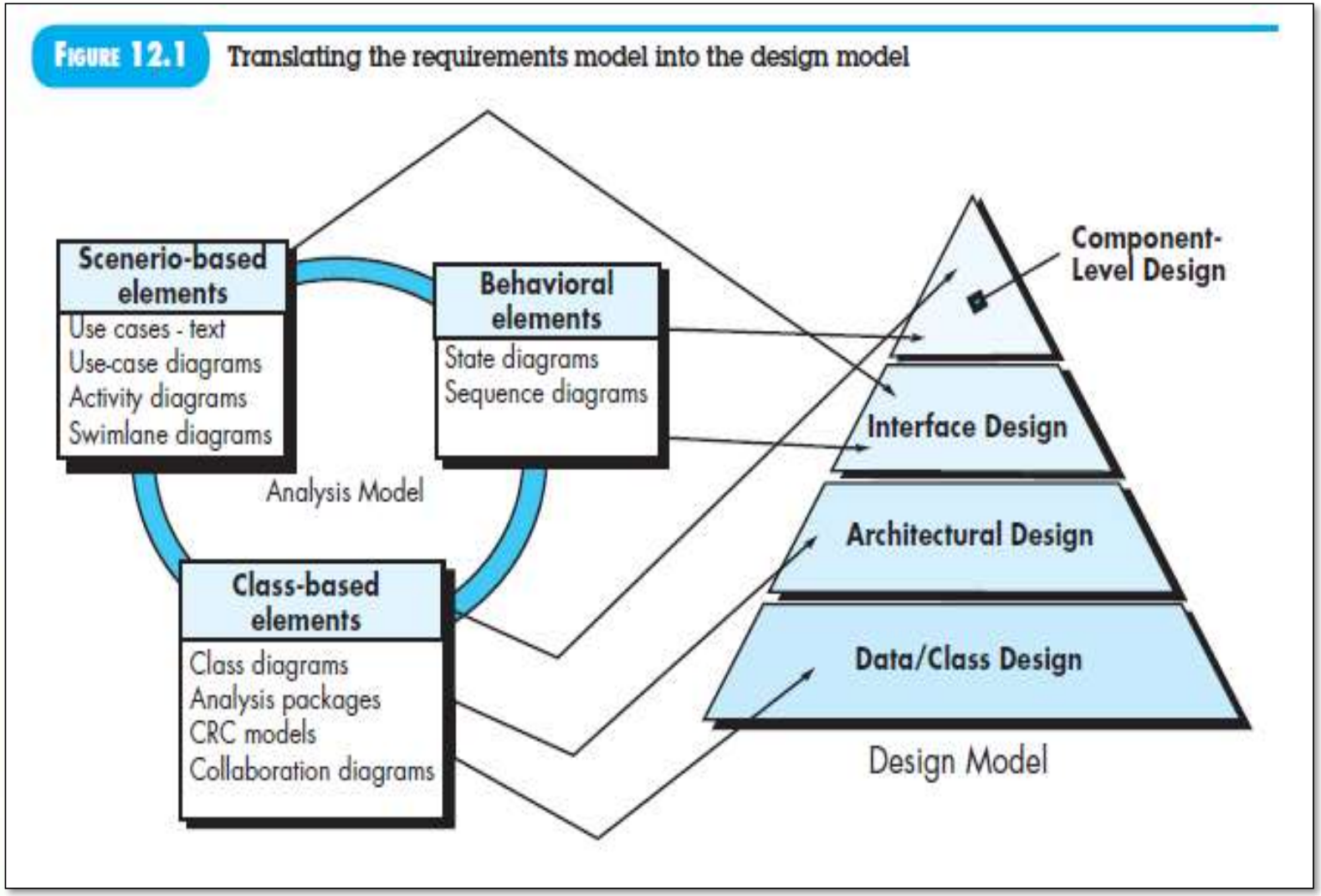


FIGURE 12.1 Translating the requirements model into the design model

- The data/cl transforms models into class realiza
- The requisi structures r implement software.
- Detailed cla occurs as ea software co designed

THE DESIGN PROCESS

- Software design is an iterative process through which requirements a into a "blueprint" for constructing the software.
- The design is represented at a high level of abstraction

Software Quality Guidelines and Attributes

- In the design process, the quality of the evolving design is assessed of technical reviews

McGlaughlin suggests **three characteristics** for the evaluation of a good d

- The design **should implement all of the explicit requireme accommodate all of the implicit requirements** desired by stakeh
- The **design should be a readable & understandable**
- The **design should provide a complete picture of the software**

Quality Guidelines .

- In order to evaluate the **quality of a design representation,** me software team must establish technical criteria for good design.

THE DESIGN PROCESS

consider the following guidelines:

1. **A design should exhibit an architecture that**
   - has been created using **recognizable architectural styles or patterns**,
   - is **composed of components that exhibit good design characteristics** and
   - can be implemented in an **evolutionary fashion**,
   - thereby facilitating **implementation and testing.**
2. **A design should be modular**
3. **A design should contain distinct representations of data, architecture, interfac** components.
4. **A design should lead to data structures that are appropriate for the** classes
5. **A design should lead to components that exhibit independent functional** chara
6. **A design should lead to interfaces that reduce the complexity of connections** be components.
7. **A design should be derived using a repeatable method that is driven by** informa during software requirements analysis.
8. **A design should be represented using a notation that effectively communicates**

THE DESIGN PROCESS

Quality Attributes: -

Hewlett-Packard developed a set of software quality attributes t
given the acronym FURPS—**f**unctionality, **u**sability, **r**eliability, **p**erfo
supportability.

The FURPS quality attributes represent a target for all software designs

- ***Functionality*** *is assessed by evaluating the feature set and capal*
  program
- ***Usability*** *is assessed by considering human factors,*
- ***Reliability*** *is evaluated by measuring the frequency and severity of fail*
- ***Performance*** *is measured using processing speed, response ti*
  consumption, throughput, and efficiency.
- ***Supportability*** *combines extensibility, adaptability, and serviceability.*

THE DESIGN PROCESS

The Evolution of Software Design

- "A designer knows that he has achieved perfection not when there is
  to add, but when there is nothing left to take away."
- These methods have a number of common characteristics:

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

  (2) a notation for representing functional components and their inter

  (3) heuristics for refinement and partitioning, and

  (4) guidelines for quality assessment.

DESIGN CONCEPTS

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

Abstraction

- "Abstraction is one of the fundamental ways that we as humans cope with
- *Designer had to derive both procedural and data abstractions that serve th*
- A ***procedural abstraction*** *refers to a sequence of* instructions that have limited function.
- A ***data abstraction*** *is a named collection of data that describes a data obje*

Architecture

- *Software architecture refers to "the overall structure of the software an* which that structure provides conceptual integrity for a system".
- In its simplest form, architecture is the **structure or organization components (modules)**

DESIGN CONCEPTS

- Shaw and Garlan describe a **set of properties for** an architectural design.
- *Structural properties define "the components* of a system (e.g., modules, obje
- **Models**
  - Structural models
  - Framework models
  - Dynamic models
  - Process models.
  - Functional models

THE DESIGN PROCESS

Patterns:

- "Each pattern describes a problem which occurs over and over environment, and then describes the core of the solution to that problem
- Brad Appleton defines a design pattern as: "A pattern is a named nug which conveys the essence of a proven solution to a recurring prob certain context amidst competing concerns"
- The intent of each design pattern is to provide a description to determine
  - (1) whether the pattern is applicable to the current work,
  - (2) whether the pattern can be reused (hence, saving design time), a
  - (3) whether the pattern can serve as a guide for developing a functionally or structurally different pattern

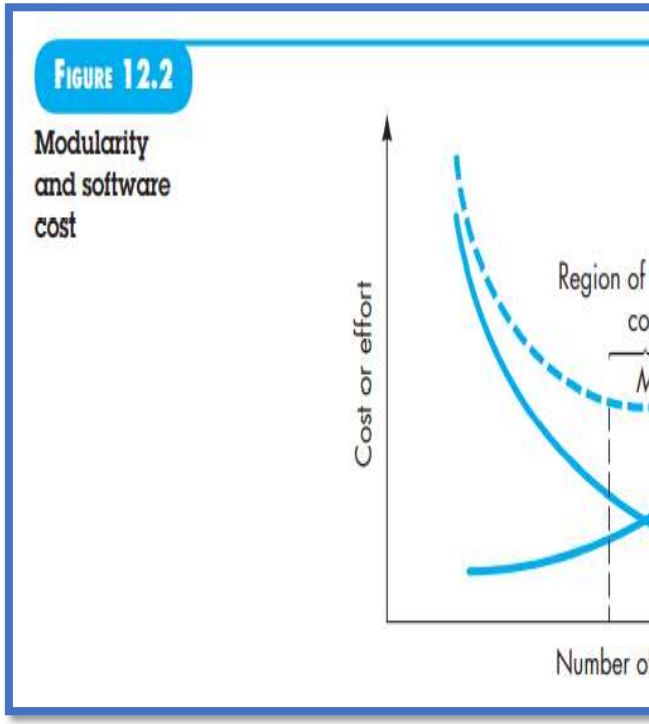
THE DESIGN PROCESS

Separation of Concerns - is a design concept that suggests that any complex problem can be more easily handled if it is subdivided into pieces that can each be solved and/or optimized independently.

A concern is a feature or behavior that is specified ed as part of the requirements model for the software.

Modularity - is the most common manifestation of separation of concerns.

Software is divided into separately named and addressable components, sometimes called modules, that are integrated to satisfy problem requirements.



THE DESIGN PROCESS

Information Hiding: The principle of information hiding [Par72] suggests tha "characterized by design decisions that (each) hides from all others."

Functional Independence: The concept of functional independence is a direct separation of concerns, modularity, and the concepts of abstraction and infor Functional independence is achieved by developing modules with "single minded an "aversion" to excessive interaction with other modules.

THE DESIGN PROCESS

- Independence is assessed using two qualitative criteria: cohesion and c
  - Cohesion is an indication of the relative functional strength of a mo
  - Coupling is an indication of the relative interdependence among mo
  - In software design, you should strive for the lowest possible couplin
  - Simple connectivity among modules results in software that understand and less prone to a “ripple effect”, caused when errors location and propagate throughout a system.

THE DESIGN PROCESS

- **Refinement:** Stepwise refinement is a top-down design strategy original by Niklaus Wirth
- **Aspects As requirements analysis** occurs, a set of "concerns" is uncover
- These concerns "include requirements, use cases, features, data structu of- service issues, variants, intellectual property boundaries, collaboratio and contracts".
- An **aspect** is a representation of a crosscutting concern.

THE DESIGN PROCESS

"**Refactoring** is the process of changing a software system in such a wa not alter the external behavior of the code [design] yet improves structure."

- Object-Oriented Design Concepts
- Design Classes

THE DESIGN MODEL

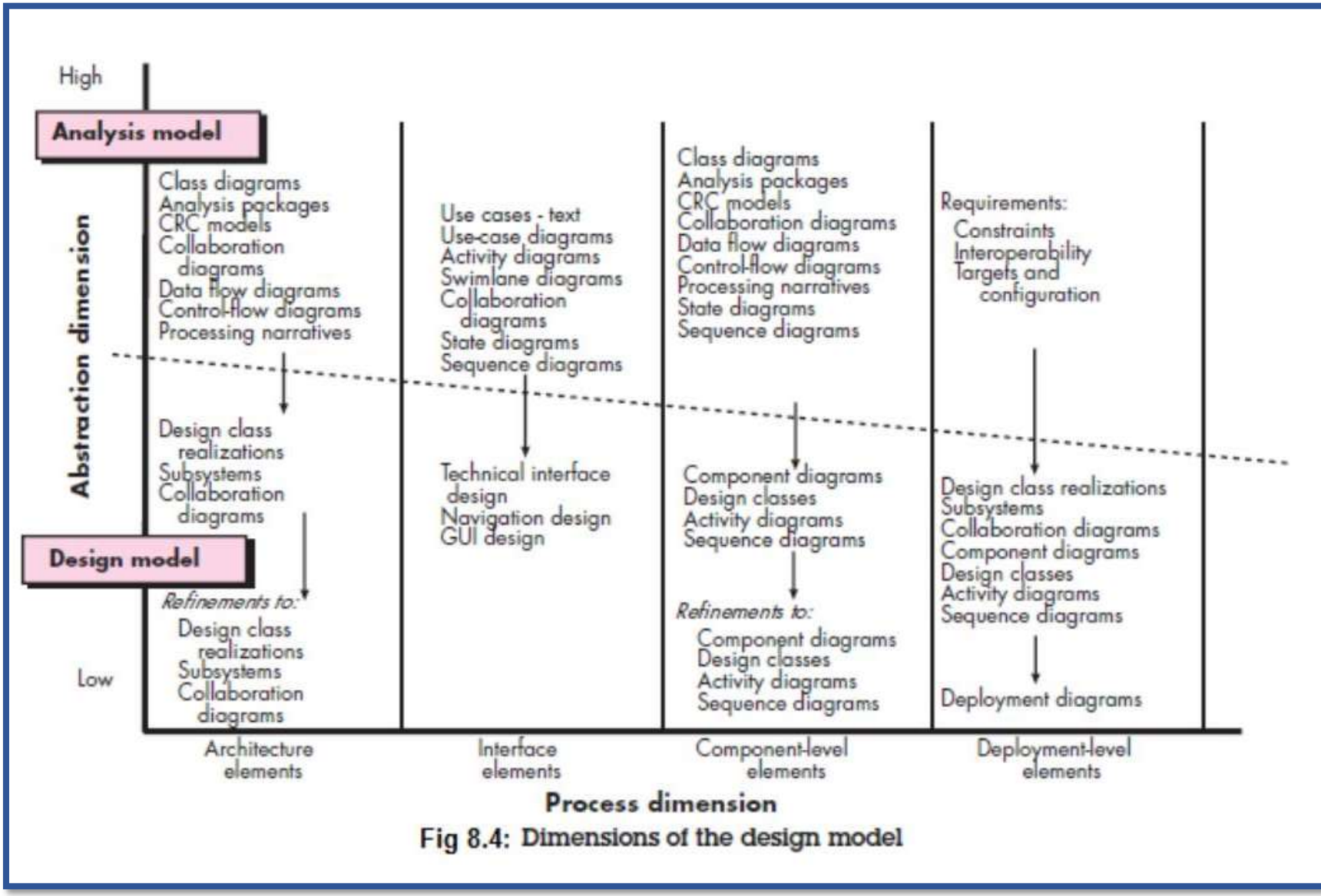


Fig 8.4: Dimensions of the design model

THE DESIGN MODEL

Data Design Elements: Like other software engineering activities, data desig
referred to as data architecting) creates a model of data and/or inform
represented at a high level of abstraction.

Architectural Design Elements: The architectural design for software is the
the floor plan of a house.

Interface Design Elements: The interface design elements for software depi
flows into and out of the system and how it is communicated among the
defined as part of the architecture.

Component-Level Design Elements: The component-level design for so
equivalent to a set of detailed drawings (and specifications) for each room in
component-level design for software fully describes the internal detail of
component.

Deployment-Level Design Elements: Deployment-level design elements indicate
functionality and subsystems will be allocated within the physical computing env
will support the software.

THE DESIGN MODEL

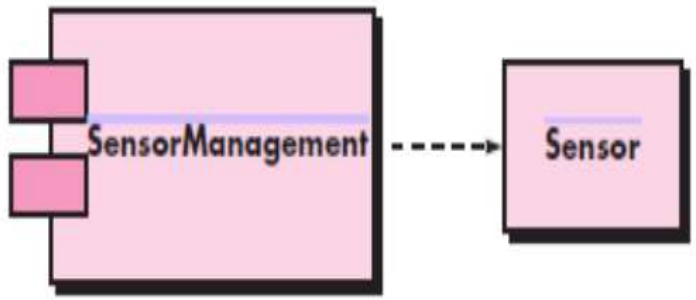


Fig 8.6: A UML component diagram

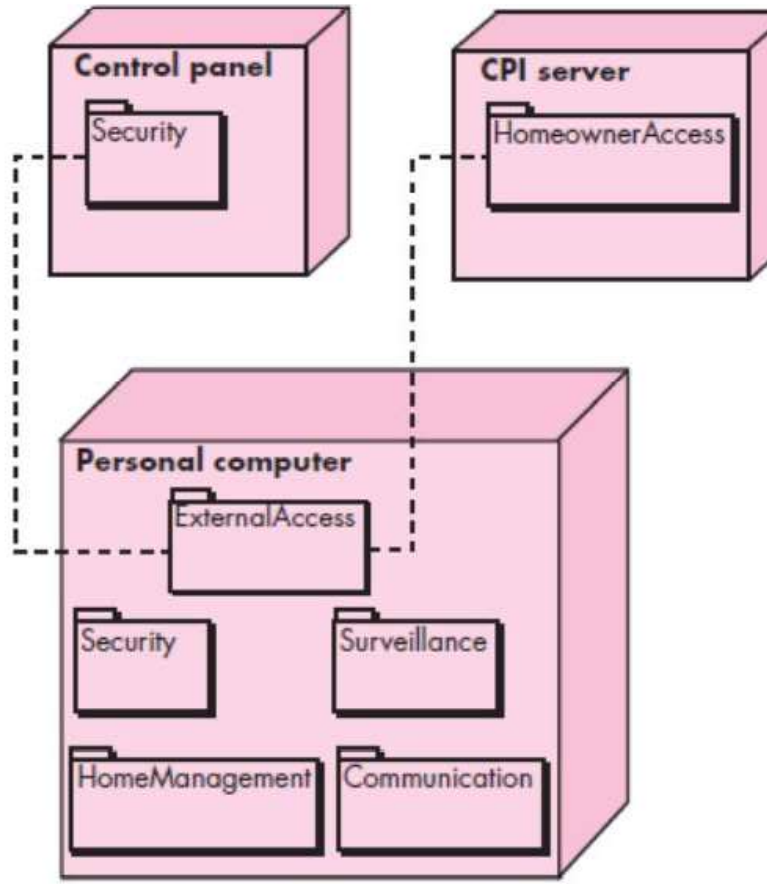


Fig 8.7: A UML deployment diagram

Chapter 6 - Architectural Design:

1. Software Architecture
2. Architectural Styles
3. Architectural Considerations
4. Architectural Design

SOFTWARE ARCHITECTURE

Architectural design represents the structure of data and program compo required to build a computer-based system.

The architecture is not the operational software. Rather, it is a representatio you to

(1) Analyze the **effectiveness of the design**

(2) Architectural alternatives at a stage when making design

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

(4) In the context of architectural design, a **software component** can be simple as a program module or an object-oriented class.

SOFTWARE ARCHITECTURE

Why Is Architecture Important?:

Three key reasons that software architecture is important:

- Representations of software architecture are an enabler for communicatio parties (stakeholders)
- The architecture highlights early design decisions
- Architecture "constitutes a relatively small, intellectually graspable model of hov structured and how its components work together".

Architectural Descriptions:

An architectural description is actually a set of work products that reflect differe system.

The IEEE standard defines an architectural description (AD) as "a collection document an architecture."

Architectural Decisions: Each view developed as part of an architectural descripti specific stakeholder concern.

ARCHITECTURAL GENRES

- In the context of architectural design, **genre** implies a specific category with software domain.
- Within each category, you encounter a number of subcategories.
- For example, within the genre of buildings, you would encounter the following houses, condos, apartment, office buildings, industrial building, warehouses, an
- Grady Booch suggests the following architectural genres for software-based sys

- Artificial intelligence
- Commercial and non-profit systems
- Communications
- Content authoring
- Devices
- Entertainment and sports
- Financial
- Games

- Government
- Industrial.
- Medical
- Military
- Operating systems
- Platforms
- Scientific
- Transportation
- Utilities

ARCHITECTURAL STYLES

- Architectural style describes a system category that encompasses :-
  (1) ***a set of components*** (e.g., a database, computational modules) th function required by a system;
  (2) ***a set of connectors*** that enable "communication, coordination and among components;
  (3) ***constraints*** that define how components can be integrated to form the
  (4) ***semantic models*** that enable a designer to understand the overall p system by analyzing the known properties of its constituent parts.
- An architectural style is a transformation that is imposed on the entire system. The intent is to establish a structure for all compo system.
- A **pattern** differs from a style in a number of fundamental ways
- Patterns can be used in conjunction with an architectural style t overall structure of a system.

ARCHITECTURAL STYLES

A Brief Taxonomy of Architectural Styles: The majority architectures can be categorized into one of a relatively small number of architectural styles:

Data-centered architectures: A data store (e.g., a file or database) resides at the center of this architecture and is accessed frequently by other components that update, add, delete, or otherwise modify data within the store.

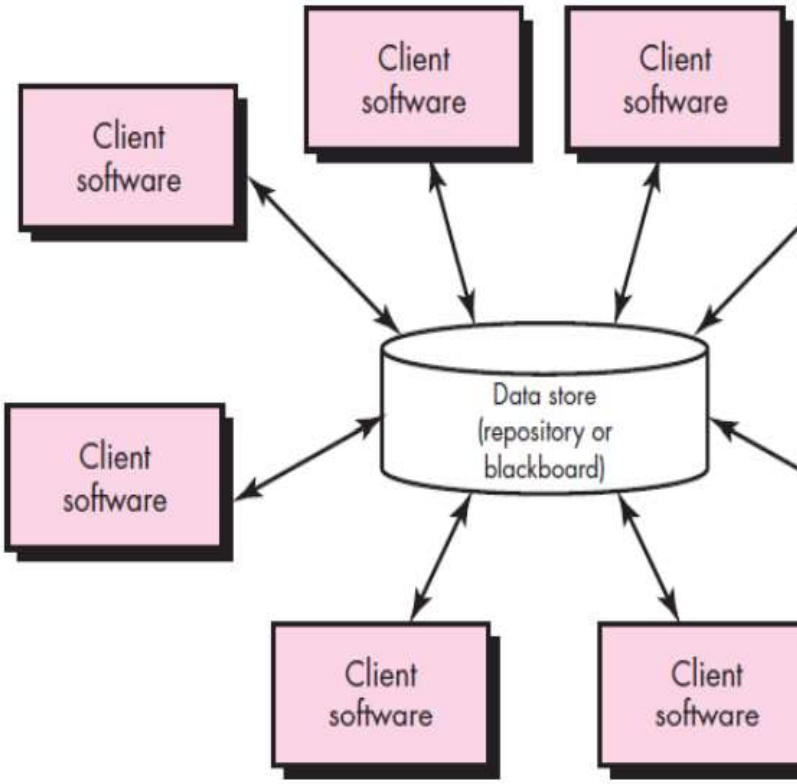


Fig 9.1: Data-centered architecture

ARCHITECTURAL STYLES

- **Data-flow architectures.** This architecture is applied when input data are to be transformed through a series of computational or manipulative components into output data.
- A pipe-and-filter pattern (Figure 9.2) has a set of components, called filters, connected by pipes that transmit data from one component to the next.
- Each filter works independently of those components upstream and downstream

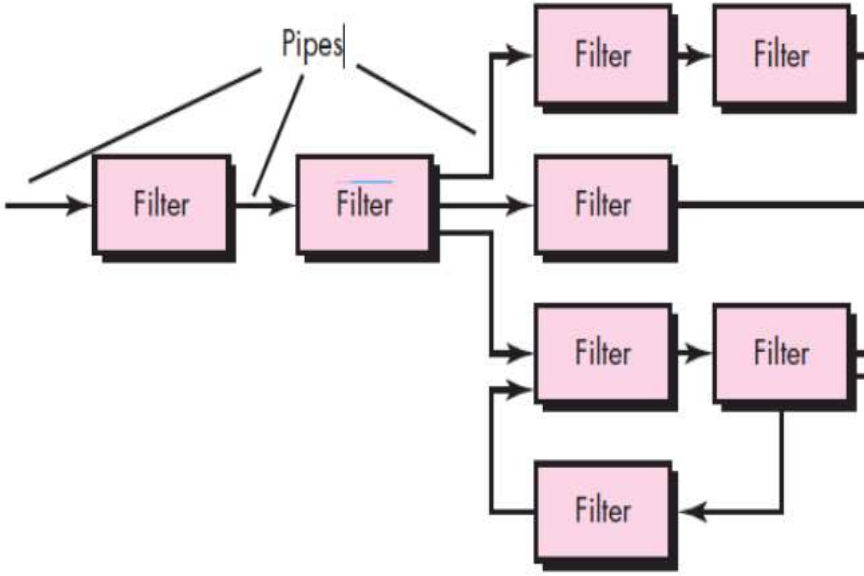


Fig 9.2: Data-flow architecture

ARCHITECTURAL STYLES

- **xCall and return architectures.** This architectural style enables you to achieve a program structure that is relatively easy to modify and scale.
- **Main program/subprogram architectures.** This classic program structure decomposes function into a control hierarchy where a "main" program invokes a number of program components that in turn may invoke still other components.
- **Remote procedure call architectures.** The components of a main program/subprogram architecture are distributed across multiple computers on a network.

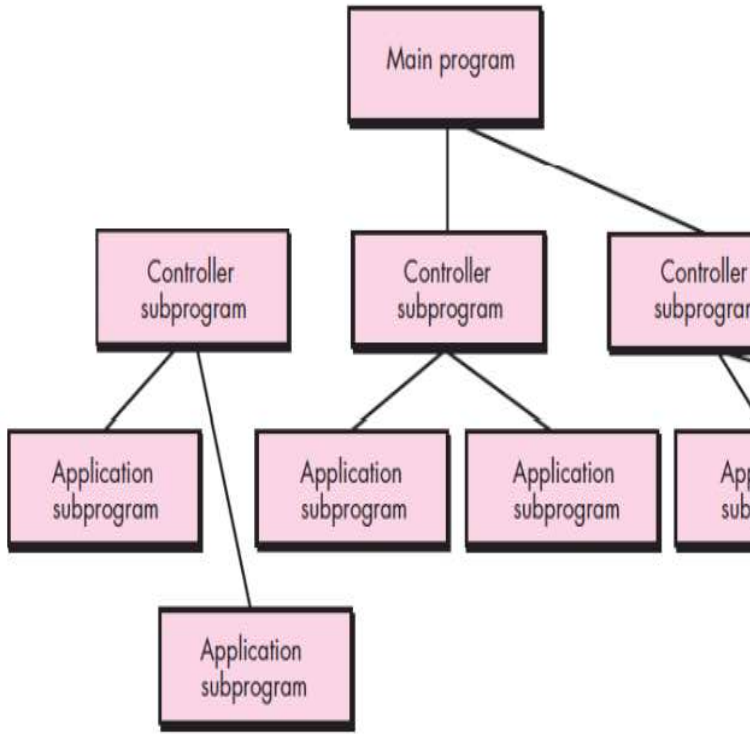


Fig 9.3: Main program / subprogram architectu

ARCHITECTURAL STYLES

Object-oriented architectures. The components of a system encapsulate data and the operations that must be applied to manipulate the data.

- Communication and coordination between components are accomplished via message passing.

Layered architectures. The basic structure of a layered architecture is illustrated in Figure 9.4. A number of different layers are defined, each accomplishing operations that progressively become closer to the machine instruction set.

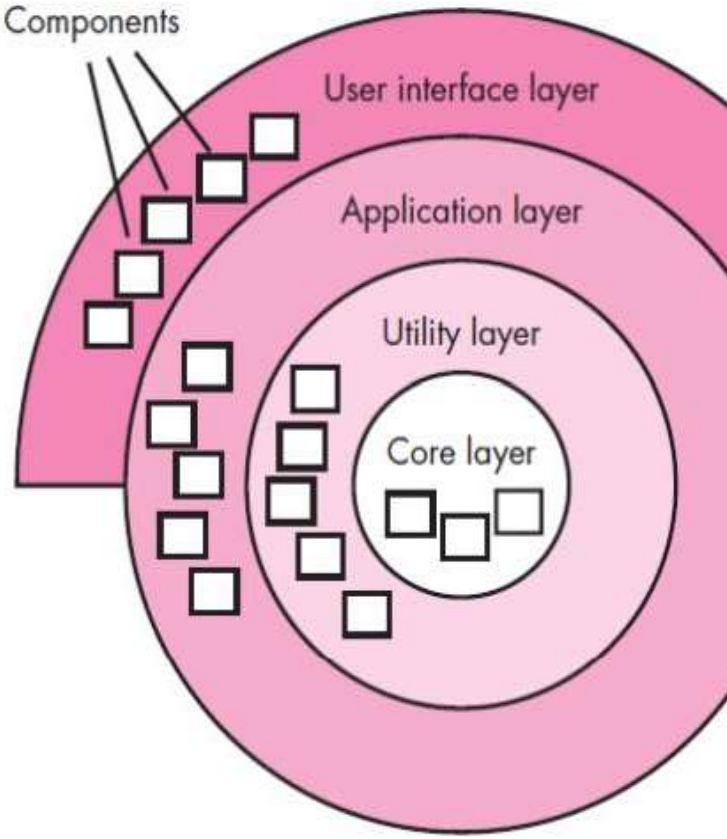


Fig 9.4: Layered architecture

ARCHITECTURAL STYLES

- **Architectural Patterns:** Architectural patterns address an applica problem within a specific context and under a set of limitations and c
- The pattern proposes an architectural solution that can serve as t architectural design.
- **For example,** the overall architectural style for an application might return or object oriented.
- **Organization and Refinement:** Because the design process often leave number of architectural alternatives, it is important to establish a s criteria that can be used to assess an architectural design that is derived.

ARCHITECTURAL DESIGN

Representing the System in Context: Architectural context represents how the softwar entities external to its boundaries.

- The generic structure of the architectural context diagram (ACD) is illustrated in Figure
- Referring to the figure, systems that interoperate with the target system are represente
  - **Superordinate systems—**those systems that use the target system as part of so processing scheme.
  - **Subordinate systems—**those systems that are used by the target system and processing that are necessary to complete target system functionality.

Peer-level systems—those systems that interact on a peer-to-peer basis

- **Actors—**entities (people, devices) that interact with the target system.
- Each of these external entities communicates with the target system through an interface.

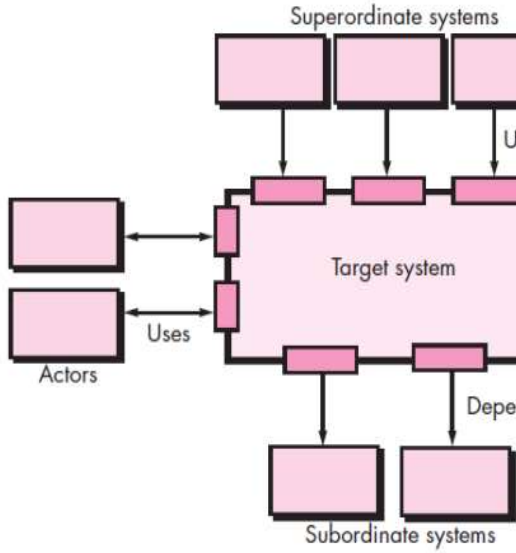


Fig 9.5: Architectural context d

ARCHITECTURAL DESIGN

Defining Archetypes:

- Archetypes are the abstract building blocks of an architectural design.
- An archetype is a class or pattern that represents a core abstraction that is critical to the design of an architecture for the target system.
- A relatively small set of archetypes is required to design even relatively complex systems.
- The following are the archetypes for safeHome: Node, Detector, Indicator., Controller.

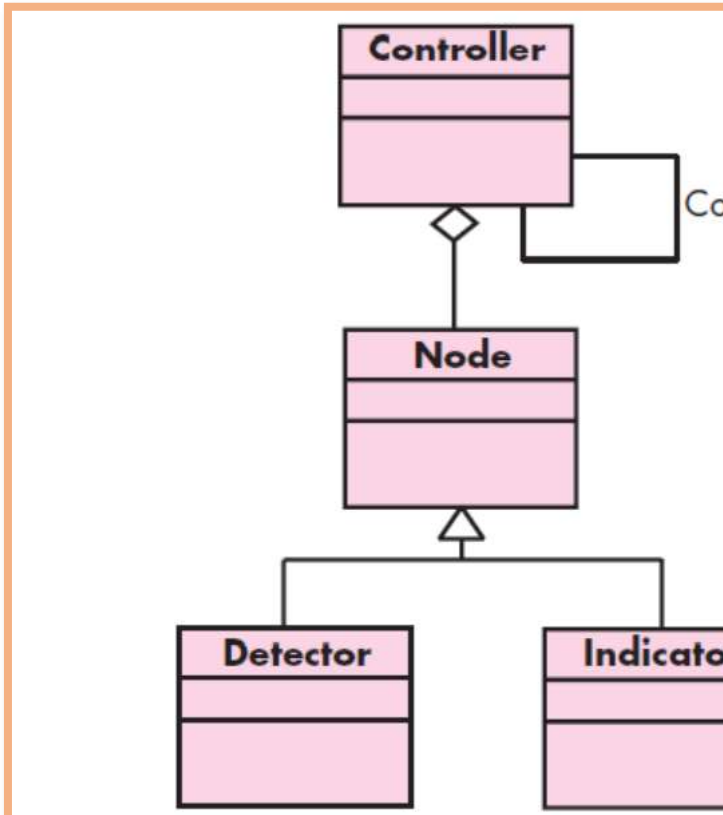


Fig 9.7: UML relationships for safehome function

ARCHITECTURAL DESIGN

- **Refining the Architecture into Components:** As the software ar refined into components, the structure of the system begins to emerge.
- For example, memory management components, communication database components, and task management components are often in the software architecture.
- As an example for SafeHome home security, the set of top-level com address the following functionality:
  - External communication management
  - Control panel processing
  - Detector management
  - Alarm processing
- **Describing Instantiations of the System:** The architectural design th modeled to this point is still relatively high level.

ARCHITECTURAL DESIGN

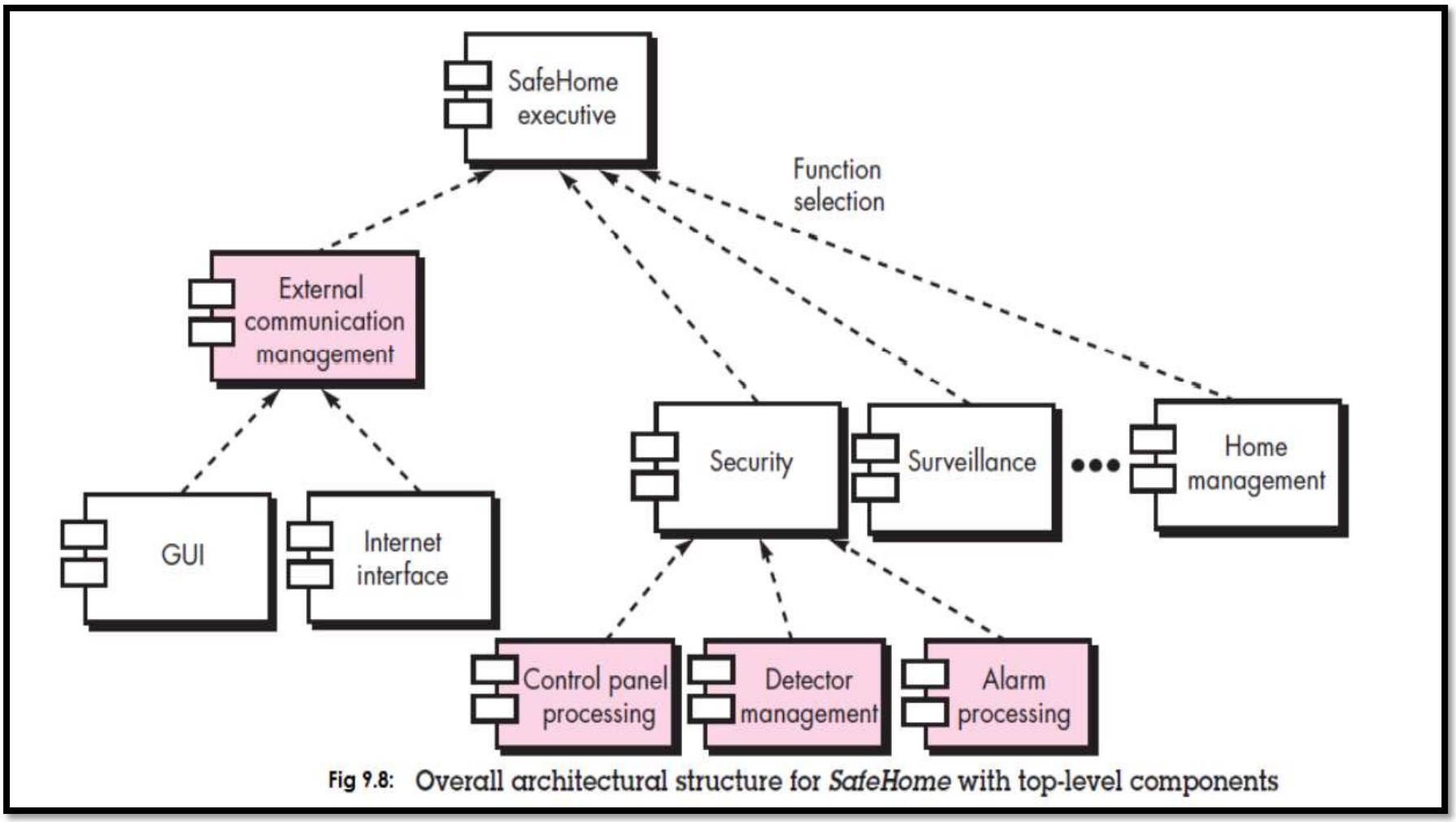


Fig 9.8: Overall architectural structure for *SafeHome* with top-level components

Chapter 7 - Component -Level Design:

1. What is a Component
2. Designing Class-Based Components
3. Conducting Component-Level Design
4. Component-Based Development

WHAT IS A COMPONENT?

- A component is a modular building block for computer software.
- More formally, component is "a modular, deployable, and replaceal system that encapsulates implementation and exposes a set of interfa
- **An Object-Oriented View:** In the context of object-oriente engineering, a component contains a set of collaborating classes.
- Each class within a component has been fully elaborated to include and operations that are relevant to its implementation.

WHAT IS A COMPONENT?

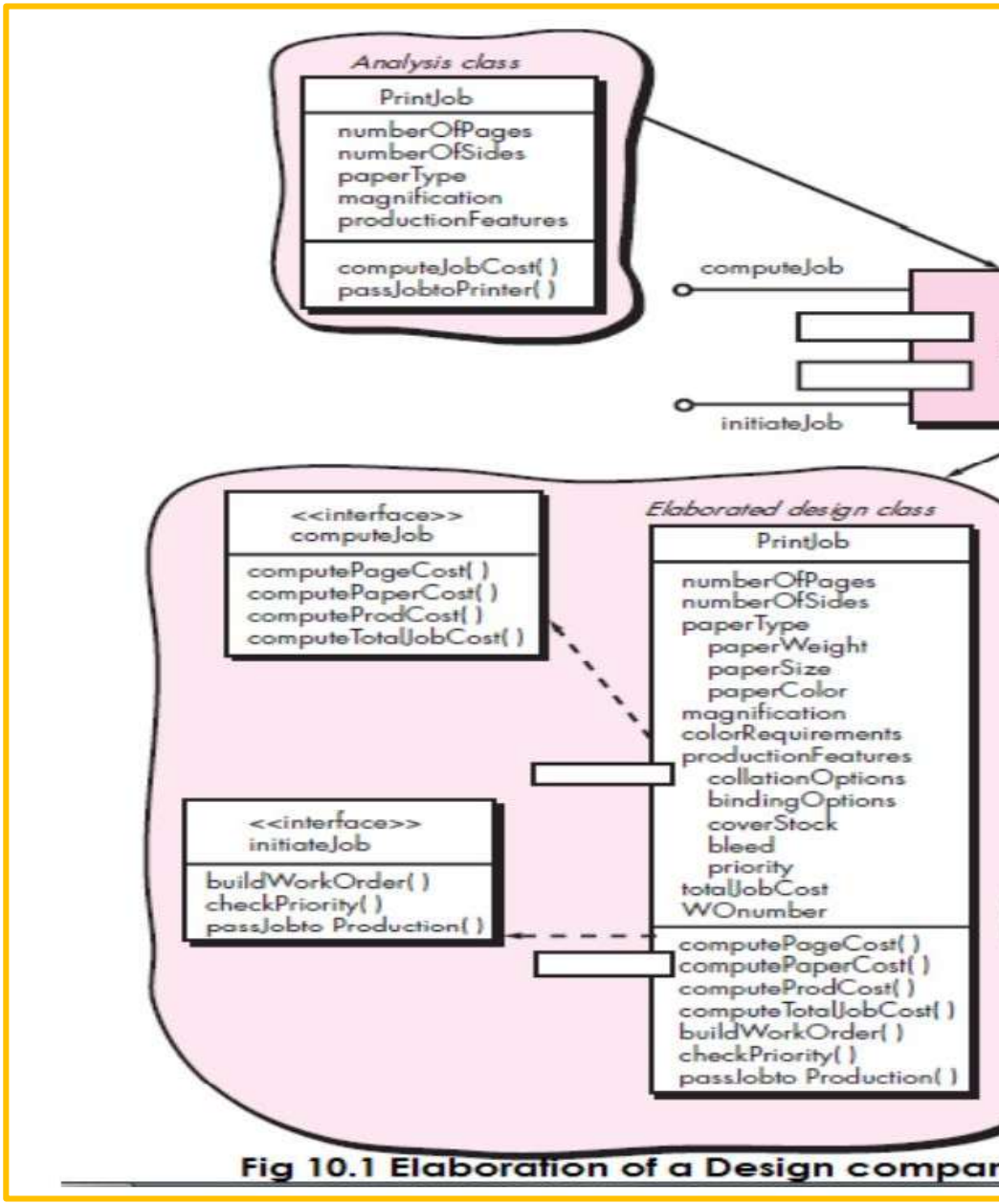


Fig 10.1 Elaboration of a Design compar

WHAT IS A COMPONENT?

The Traditional View: In the context of traditional software engineerin component is a functional element of a program that incorporates proce the internal data structures and an interface that enables the componen invoked and data to be passed to it.

A traditional component, also called a module, resides within the softwa architecture and serves one of three important roles:
(1) a control component
(2) a problem domain component
(3) an infrastructure component

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

WHAT IS A COMPONENT?

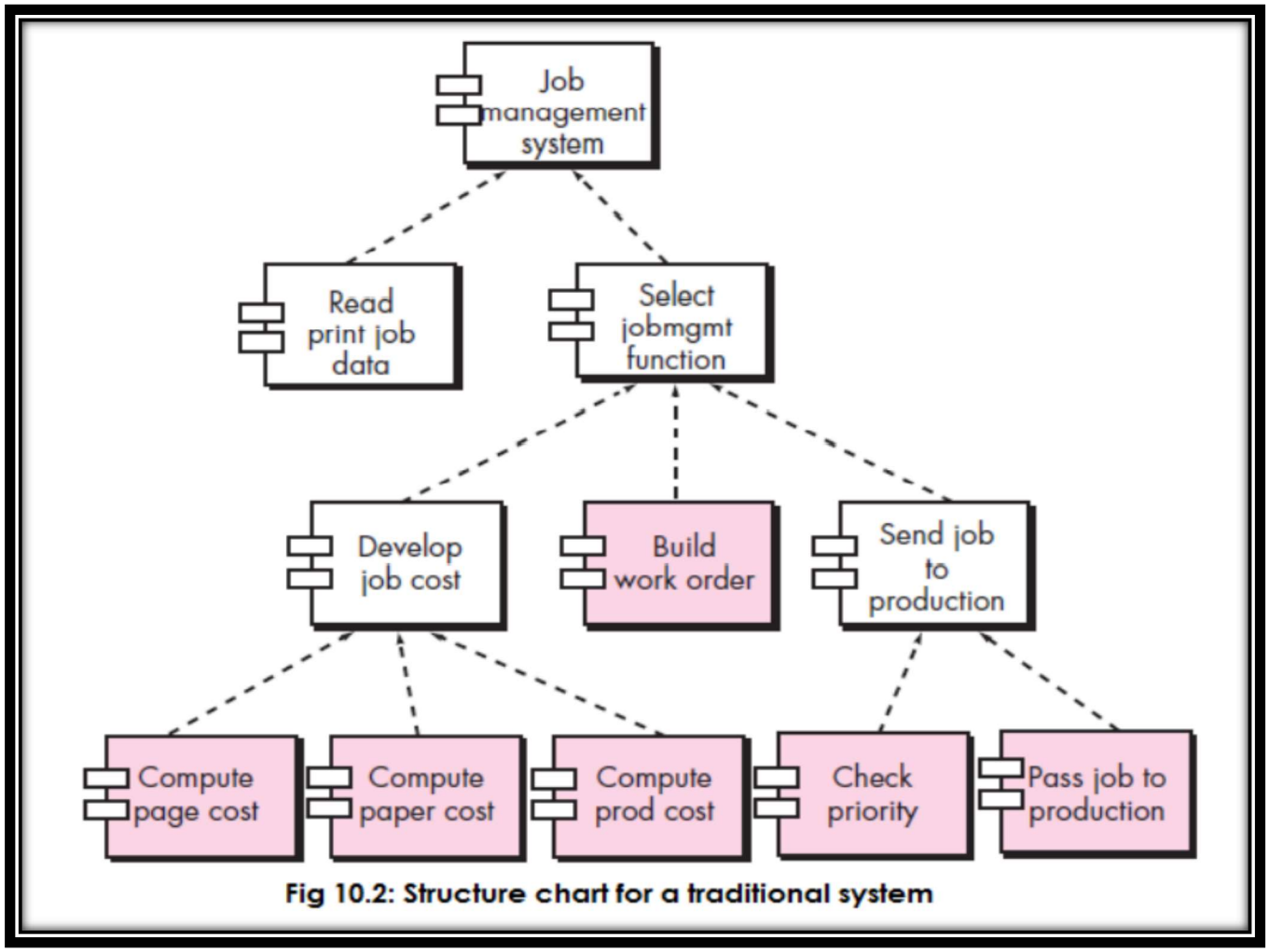


Fig 10.2: Structure chart for a traditional system

WHAT IS A COMPONENT?

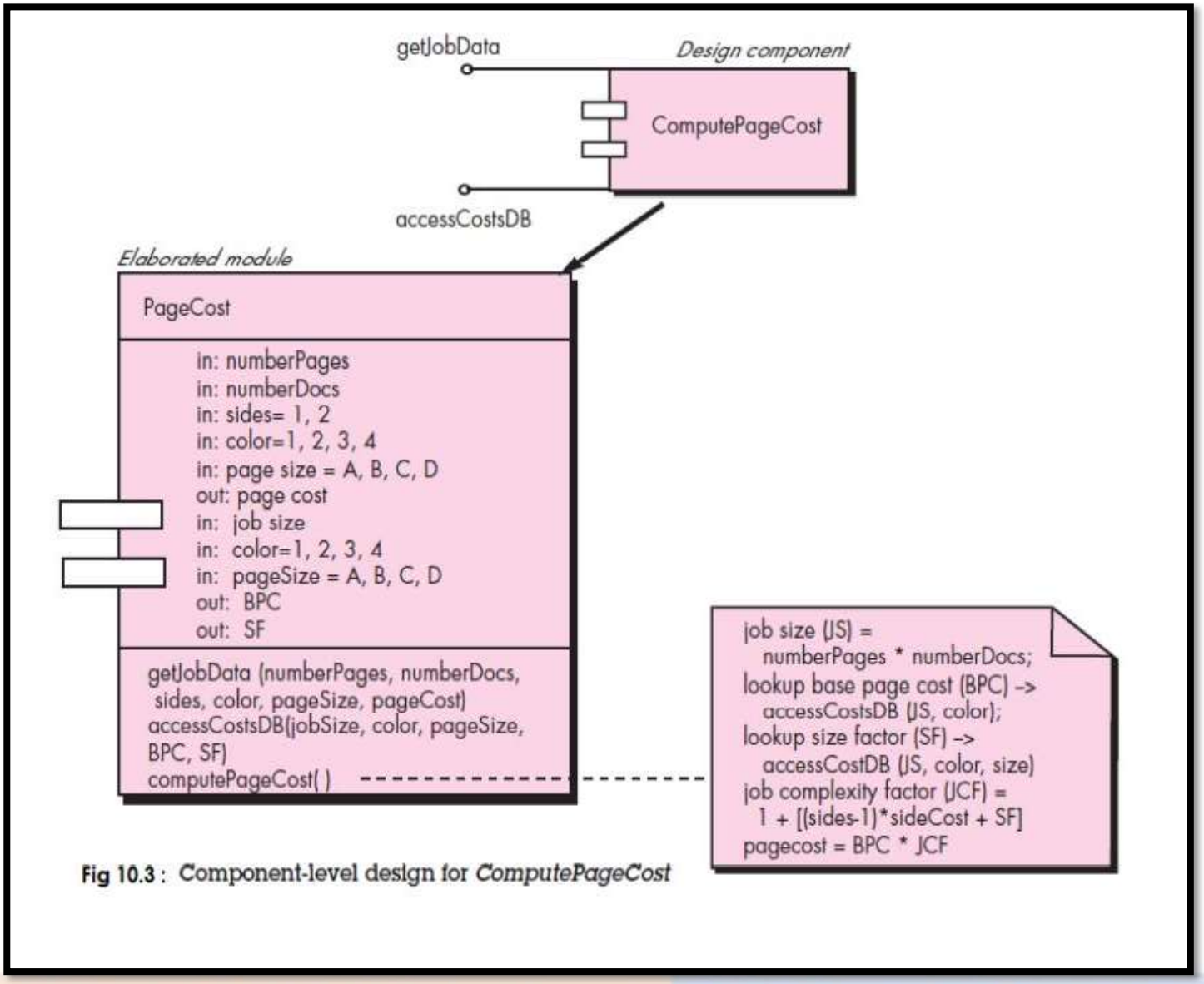


Fig 10.3 : Component-level design for *ComputePageCost*

DESIGNING CLASS-BASED COMPONENTS

- The detailed description of the attributes, operations, and interfaces used by design detail required as a precursor to the construction activity.
- **Basic Design Principles:** Four basic design principles are applicable to compone and have been widely adopted when object-oriented software engineering is ap
- The Open-Closed Principle (OCP):
- "A module [component] should be open for extension but closed for modificati

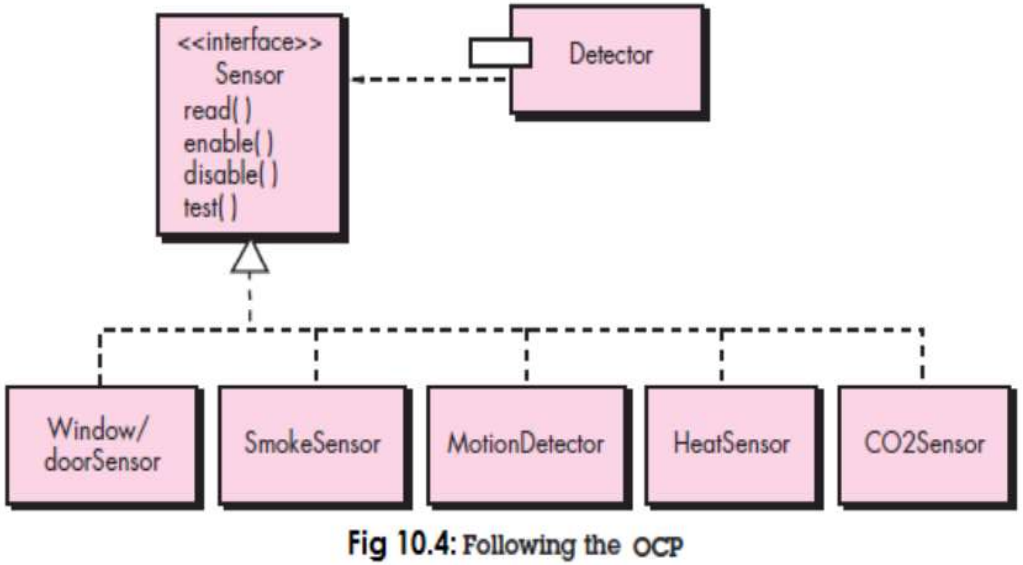


Fig 10.4: Following the OCP

DESIGNING CLASS-BASED COMPONENTS

Design Principles:

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 int

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"

DESIGNING CLASS-BASED COMPONENTS

Component-Level Design Guidelines: Ambler suggests the following g components, their interfaces, and the dependencies and inheritance cl Components.

Ambler recommends that:

(1) lollipop representation of an interface should be used when dia complex;

(2) for consistency, interfaces should flow from the left-hand side of th box;

(3) only those interfaces that are relevant to the component under conside be shown, even if other interfaces are available.

Dependencies and Inheritance:

For improved readability, it is a good to model dependencies from left inheritance from bottom (derived classes) to top (base classes).

DESIGNING CLASS-BASED COMPONENTS

Cohesion: Cohesion is the "single-mindedness" of a component. Lethbridge and Laganiére define a number of different types of cohesion.

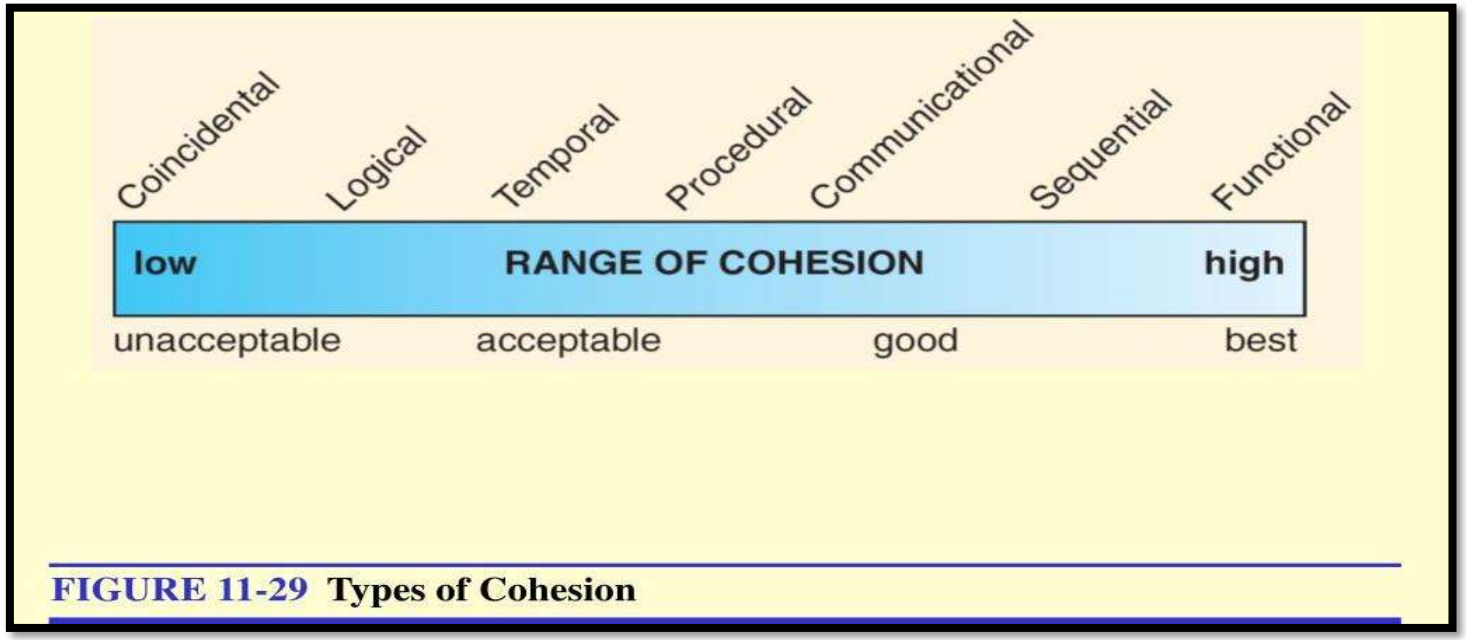


FIGURE 11-29 Types of Cohesion

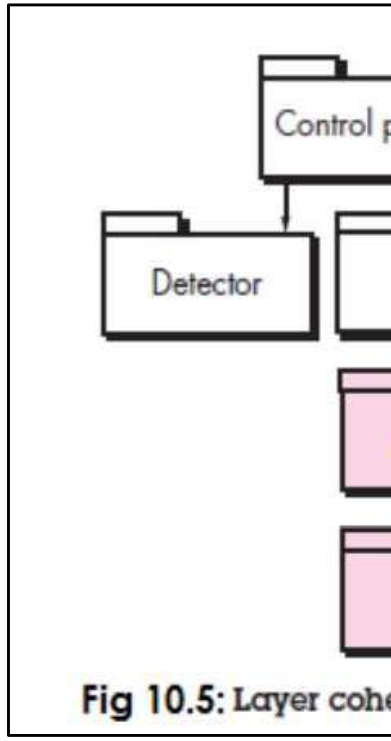
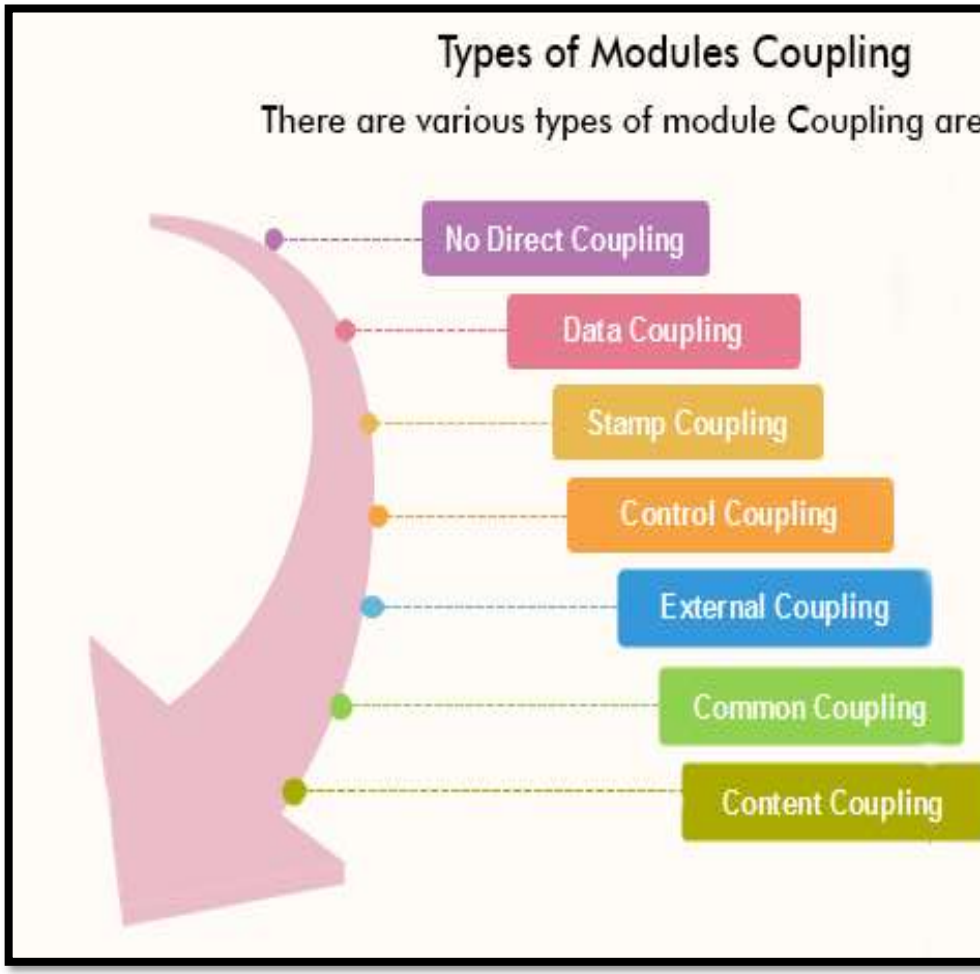


Fig 10.5: Layer coh

DESIGNING CLASS-BASED COMPONENTS

- **Coupling:** Communication and collaboration are essential elements of any object-oriented system. Coupling is a qualitative measure of the degree to which classes are connected to one another.
- An important objective in component-level design is to keep coupling as low as is possible.



CONDUCTING COMPONENT-LEVEL DESIGN

The following steps represent a typical task set for component-level desig applied for an object-oriented system.

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 compone

- Step 3a. Specify message details when classes or components collaborate.
- Step 3b. Identify appropriate interfaces for each component.
- Step 3c. Elaborate attributes and define data types and data structures require them.

Step 3d. Describe processing flow within each operation in detail.

Figure 10.8 depicts a UML activity diagram for computePaperCost().

Step 4. Describe persistent data sources (databases and files) and identif required to manage them. Databases and files normally transcend the design an individual component.

CONDUCTING COMPONENT-LEVEL DESIGN

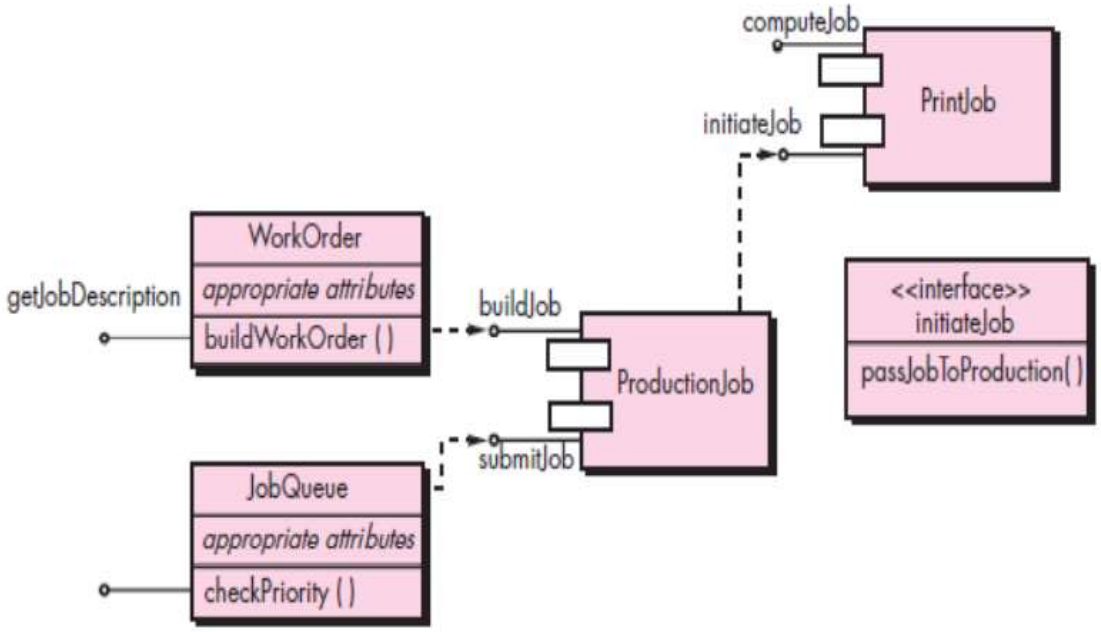


Fig 10.7: Refactoring interfaces and class definitions for PrintJob

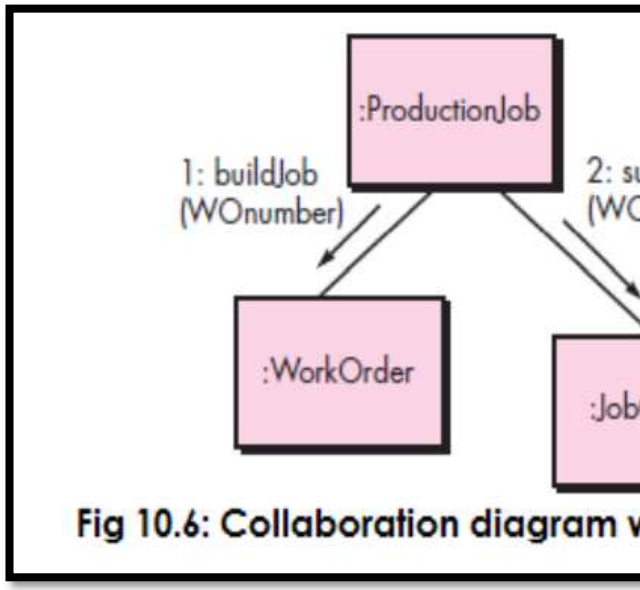


Fig 10.6: Collaboration diagram

CONDUCTING COMPONENT-LEVEL DESIGN

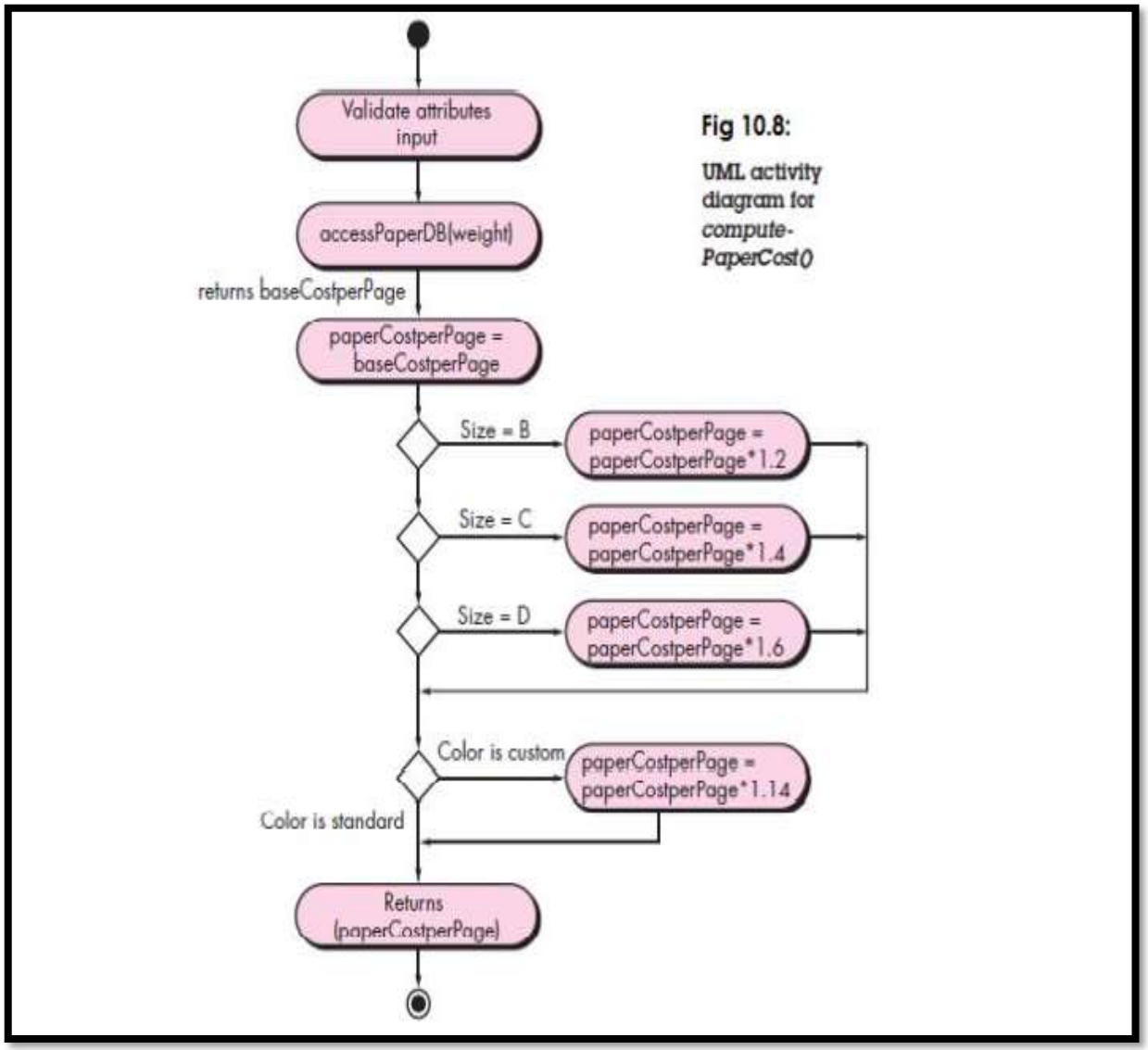
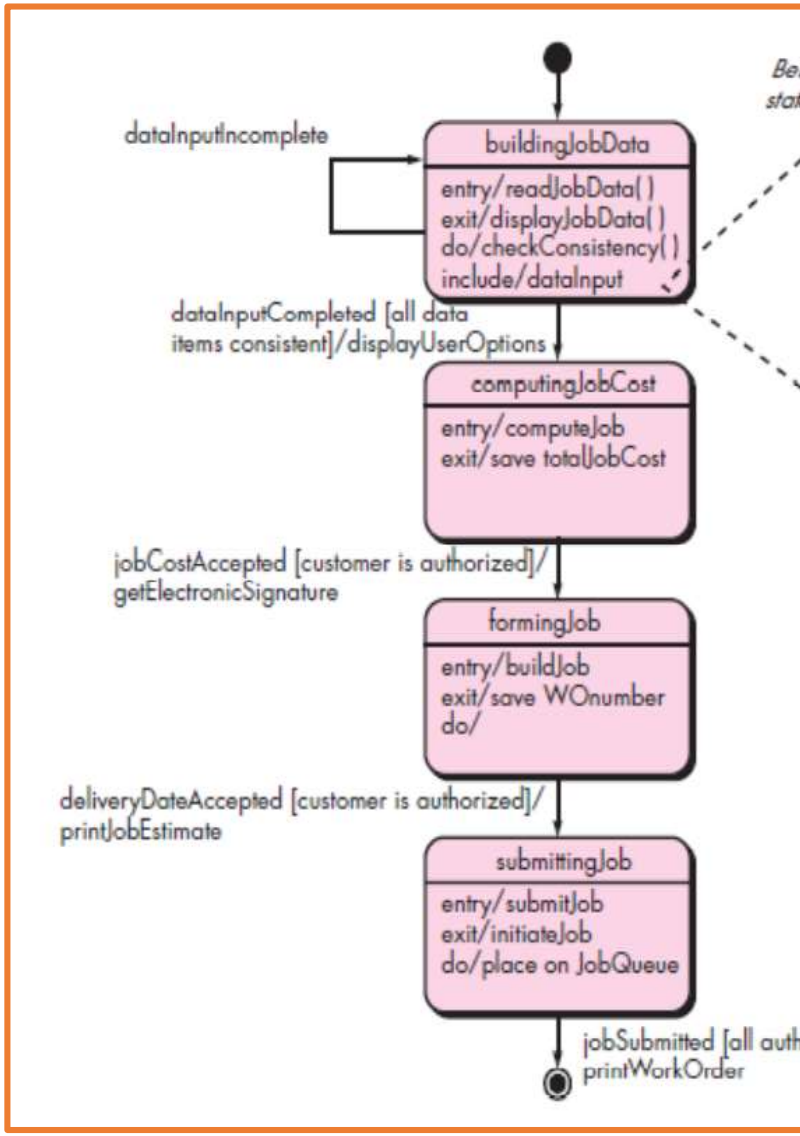


Fig 10.8:

UML activity diagram for *compute-PaperCost()*

CONDUCTING COMPONENT-LEVEL DESIGN

- **Step 5.** Develop and elaborate behavioral representations for a class or component.
- **Step 6.** Elaborate deployment diagrams to provide additional implementation detail.
- Deployment diagrams are used as part of architectural design and are represented in descriptor form.
- **Step 7.** Refactor every component-level design representation and always consider alternatives.
- The design is an iterative process.



COMPONENT-LEVEL DESIGN FOR WEBAPPS

WebApp component is

(1) a well-defined cohesive function that manipulates content computational or data processing for an end user or

(2) a cohesive package of content and functionality that provides the e some required capability. Therefore, componentlevel design for We incorporates elements of content design and functional design.

Design at the Component Level: Content design at the component leve content objects and the manner in which they may be packaged for pres WebApp end user.

- The formality of content design at the component level should be characteristics of the WebApp to be built.

COMPONENT-LEVEL DESIGN FOR WEBAPPS

Functional Design at the Component Level: Modern Web applica
increasingly sophisticated processing functions that

(1) perform localized processing to generate content and navigation ca
dynamic fashion,

(2) provide computation or data processing capability that is approp
WebApp's business domain,

(3) provide sophisticated database query and access, or

(4) establish data interfaces with external corporate systems.

During architectural design, WebApp content and functionality are combin
functional architecture.

COMPONENT-LEVEL DESIGN FOR WEBAPPS

Designing Traditional Components: The foundations of component-lev
traditional software components were formed in the early 1960s and w
with the work of Edsger Dijkstra and his colleagues.

- The structured constructs are logical chunks that allow a reader
  procedural elements of a module, rather than reading the design or code

Graphical Design Notation: "A picture is worth a thousand words".

- The activity diagram allows you to represent sequence, condition, and re
  elements of structured programming
- A box is used to indicate a processing step.
- A diamond represents a logical condition, and
- arrows show the flow of control.
- Figure 10.10 illustrates three structured constructs.

COMPONENT-LEVEL DESIGN FOR WEBAPPS

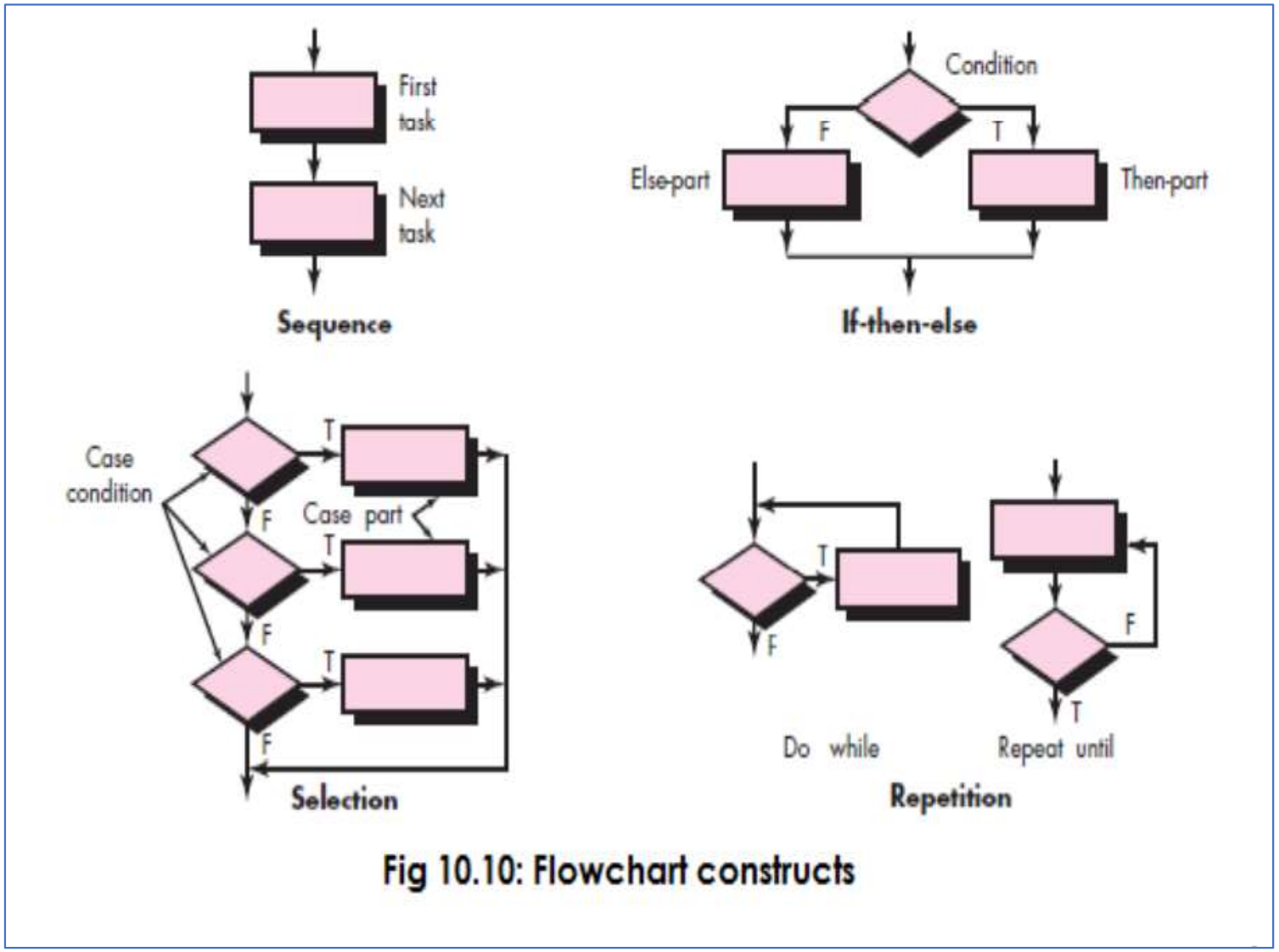


Fig 10.10: Flowchart constructs

COMPONENT-LEVEL DESIGN FOR WEBAPPS

Tabular Design Notation: Decision tables provide a notation that translates actions (described in a processing narrative or a use case) into a tabular form.

	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
Actions						
No discount	✓					
Apply 8 percent discount			✓	✓		
Apply 15 percent discount					✓	✓
Apply additional x percent discount		✓		✓		✓

Fig 10.11: Decision table nomenclature

COMPONENT-LEVEL DESIGN FOR WEBAPPS

- **Program Design Language:** Program design language (PDL), also called structured English or pseudocode, incorporates the logical structure of a programming language with the free-form expressive ability of a natural language (e.g., English).
- Narrative text (e.g., English) is embedded within a programming language-like syntax.

```
do for all sensors
    invoke checkSensor procedure returning
    if signalValue > bound [alarmType]
        then phoneMessage = message [alar
            set alarmBell to "on" for alarmTim
            set system status = "alarmCondi
            parbegin
                invoke alarm procedure with "
                invoke phone procedure set to
            endpar
        else skip
    endif
enddofor
end alarmManagement
```

COMPONENT-BASED DEVELOPMENT

- Component-based software engineering (CBSE) is a process that emphasizes t construction of computer-based systems using reusable software "components

Domain Engineering:

- The intent of domain engineering is to identify, construct, catalog, and disser software components that have applicability to existing and future software application domain.
- Domain engineering includes three major activities—analysis, const dissemination.

Component Qualification, Adaptation, and Composition:

- Domain engineering provides the library of reusable components that are component-based software engineering.
- Some of these reusable components are developed in-house, others can be e existing applications, and still others may be acquired from third parties.

COMPONENT-BASED DEVELOPMENT

Component Qualification.

Component qualification ensures that a candidate component will perform required, will properly "fit" into the architectural style specified for the sys exhibit the quality characteristics that are required for the application.

Among the many factors considered during component qualification are:

- Application programming interface (API).
- Development and integration tools required by the component.
- Run-time requirements
- Service requirements
- Security features
- Embedded design assumptions
- Exception handling

COMPONENT-BASED DEVELOPMENT

Component Adaptation.

Conflicts may occur in one or more of the areas in selection of compone these conflicts, an adaptation technique called component wrapping i used.

When a software team has full access to the internal design and component **white-box wrapping** is applied.

Gray-box wrapping is applied when the component library provides a extension language or API that enables conflicts to be removed or maske

Black-box wrapping requires the introduction of pre- and postproce component interface to remove or mask conflicts.

COMPONENT-BASED DEVELOPMENT

Component Composition.

- The component composition task assembles qualified, adapted, an components to populate the architecture established for an application.
- To accomplish this, an infrastructure and coordination must be establish components into an operational system.
- **OMG/CORBA.** The Object Management Group has published a common broker architecture (OMG/CORBA).

Analysis and Design for Reuse:

- Design concepts such as abstraction, hiding, functional independence, re structured programming, along with object-oriented methods, testing, so assurance (SQA), and correctness verification methods all contribute to t software components that are reusable.
- The requirements model is analyzed to determine elements that point to ex components.

COMPONENT-BASED DEVELOPMENT

Classifying and Retrieving Components:

- An ideal description of a resusable component encompasses the concept, content, and context.
- The concept of a software component is "a description of what the does".
- The interface to the component is fully described and the represented within the context of pre- and post conditions— is identi
- The content of a component describes how the concept is realized. places a reusable software component within its domain of applicabil

Assignment: A reuse environment exhibits the what characteristics?

SOFTWARE ENGINEERING

Chapter 8 - User Interface Design

User Interface Design Rules./ The Golden Rules

Background

- Interface design focuses on the following
- The design of interfaces between software components
- The design of interfaces between the software and other nonhuman produc

consumers of information

- The design of the interface between a human and the computer
- Graphical user interfaces (GUIs) have helped to eliminate many of the most

interface problems

- However, some are still difficult to learn, hard to use, confusing, counterintu

unforgiving, and frustrating

- User interface analysis and design has to do with the study of people and ho

to technology

https://www.youtube.com/watch?v=M8VIkjtKCZk

Summary: Golden Rules

- *Place User in Control*
  - Define interaction in such a way that the user is not forced into performing unnecessary or unde actions
  - Provide for flexible interaction (users have varying preferences)
  - Allow user interaction to be interruptible and reversible
  - Streamline interaction as skill level increases and allow customization of interaction
  - Hide technical internals from the casual user
  - Design for direct interaction with objects that appear on the screen
- *Reduce User Cognitive (Memory) Load*
  - Reduce demands on user's short-term memory
  - Establish meaningful defaults
  - Define intuitive short-cuts
  - Visual layout of user interface should be based on a familiar real world metaphor
  - Disclose information in a progressive fashion
- *Make Interface Consistent*
  - Allow user to put the current task into a meaningful context
  - Maintain consistency across a family of applications
  - If past interaction models have created user expectations, do not make changes unless there is good reason to do so

END OF THE UNIT