

# MODULE 2

# Syllabus

- ▶ **Requirement Analysis and Design** - Functional and non-functional requirements, Requirements engineering processes, Requirements elicitation, Requirements validation, Requirements change, **Traceability matrix**, Developing use cases, **Software Requirements Specification Template**, Personas, Scenarios, User stories, Feature identification.
- ▶ **Design concepts** - Design within the context of software engineering, Design Process, Design concepts, Design Model.

- ▶ **Architectural Design** - Software Architecture, Architectural Styles, Architectural considerations, Architectural Design.
- ▶ **Component level design** - What is a component?, Designing Class-Based Components, Conducting Component level design, Component level design for web-apps.
- ▶ Template of a Design Document as per “IEEE Std 1016-2009 IEEE Standard for Information Technology Systems Design Software Design Descriptions”.
- ▶ Case study: The Arienne 5 launcher failure.

# Requirement Engineering

- ▶ The process of establishing the services that the customer requires from a system and the constraints under which it operates and is developed.
- ▶ The requirements themselves are the descriptions of the system services and constraints that are generated during the requirements engineering process.
- ▶ It may range from a high-level abstract statement of a service or of a system constraint to a detailed mathematical functional specification

# Types of requirement

## User requirements

- ▶ Statements in natural language plus diagrams of the services the system provides and its operational constraints. Written for customers.

## System requirements

- ▶ A structured document setting out detailed descriptions of the **system's functions, services and operational constraints**. Defines **what should be implemented** so may be part of a contract between client and contractor.

# Functional and non-functional requirements

## Functional requirements

- ▶ Statements of services the system should provide, **how the system should react to particular inputs and how the system should behave in particular situations.**
- ▶ May state what the system should not do.

## Non-functional requirements

- ▶ Constraints on the services or functions offered by the system such as timing constraints, Security constraints on the development process, standards, etc. Often apply to the system as a whole rather than individual features or services.

# Functional requirements

- Describe functionality or system services.
- Depend on the type of software, expected users and the type of system where the software is used.
- Functional user requirements may be high-level statements of what the system should do.
- Functional system requirements should describe the system services in detail.

# Non-functional requirements

- ▶ These define system properties and constraints e.g. reliability, response time and storage requirements etc.
- ▶ Constraints are I/O device capability, system representations, etc.
- ▶ Process requirements may also be specified mandating a particular IDE, programming language or development method.
- ▶ Non-functional requirements may be more critical than functional requirements.



# Non-functional classifications

## Product requirements

- ▶ Requirements which specify that the delivered product must **behave in a particular way**  
e.g. execution speed, reliability, etc.

## Organizational requirements

- ▶ Requirements which are a consequence of **organizational policies and procedures**  
e.g. process standards used, implementation requirements, etc.

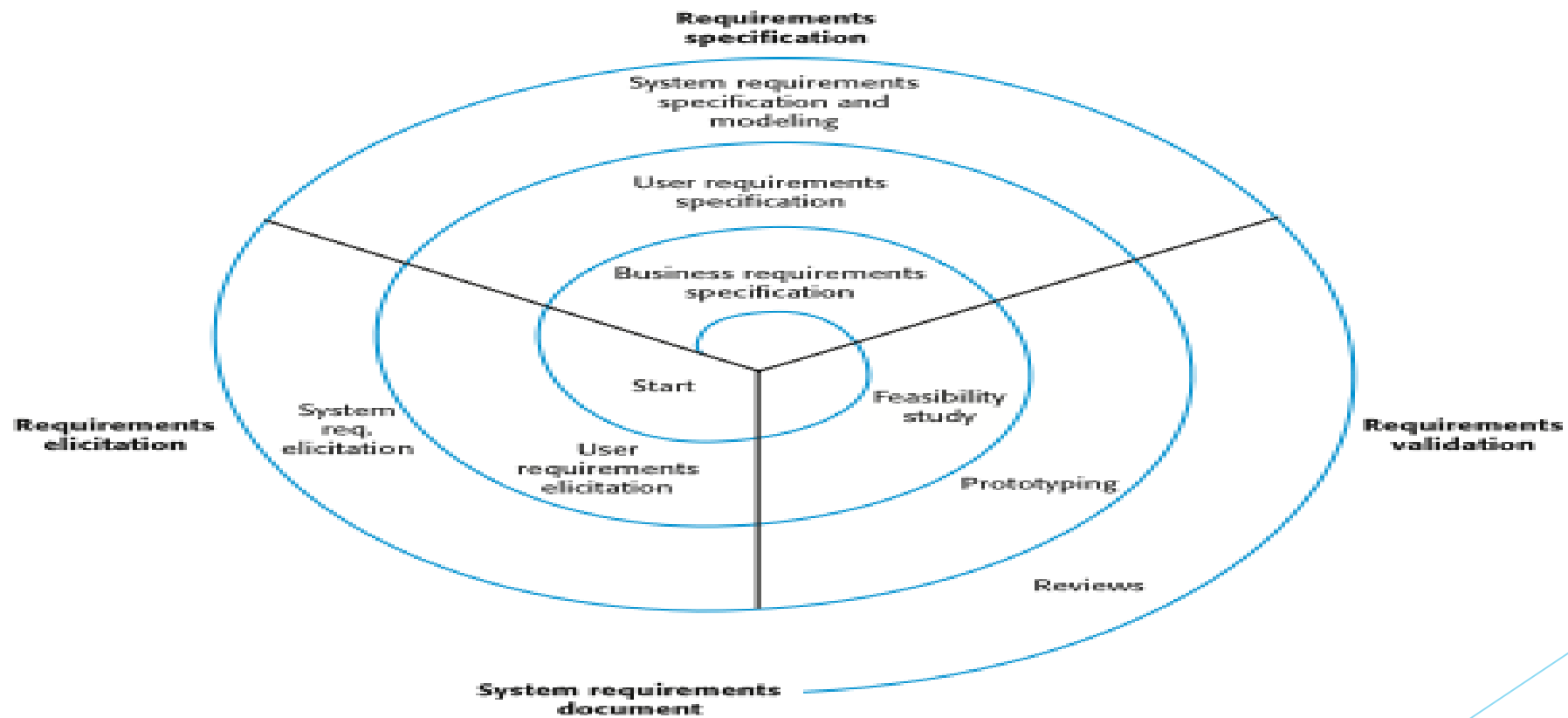
## External requirements

- ▶ Requirements which arise from factors which are external to the system development process e.g. interoperability requirements, legislative requirements etc.
- ▶ *legislative requirements- Laws, regulations, and standards imposed by government bodies or industry authorities may dictate certain features or behaviors that the system must adhere to. This includes data privacy regulations, accessibility standards, and industry-specific compliance requirements*

## **Usability requirements**

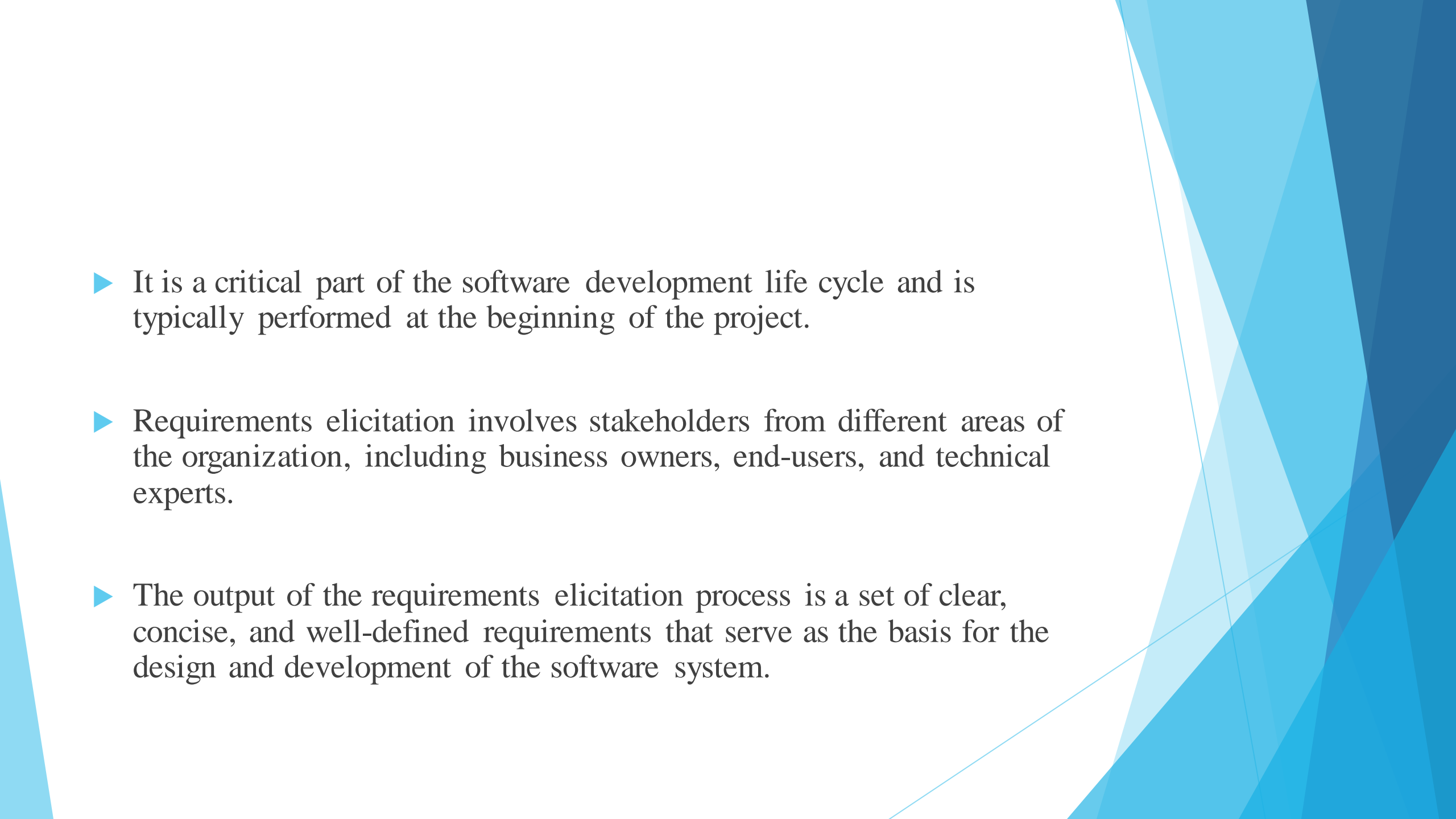
The system should be easy to use and should be organized in such a way that user errors are minimized. (Goal).

# Requirement Engineering process

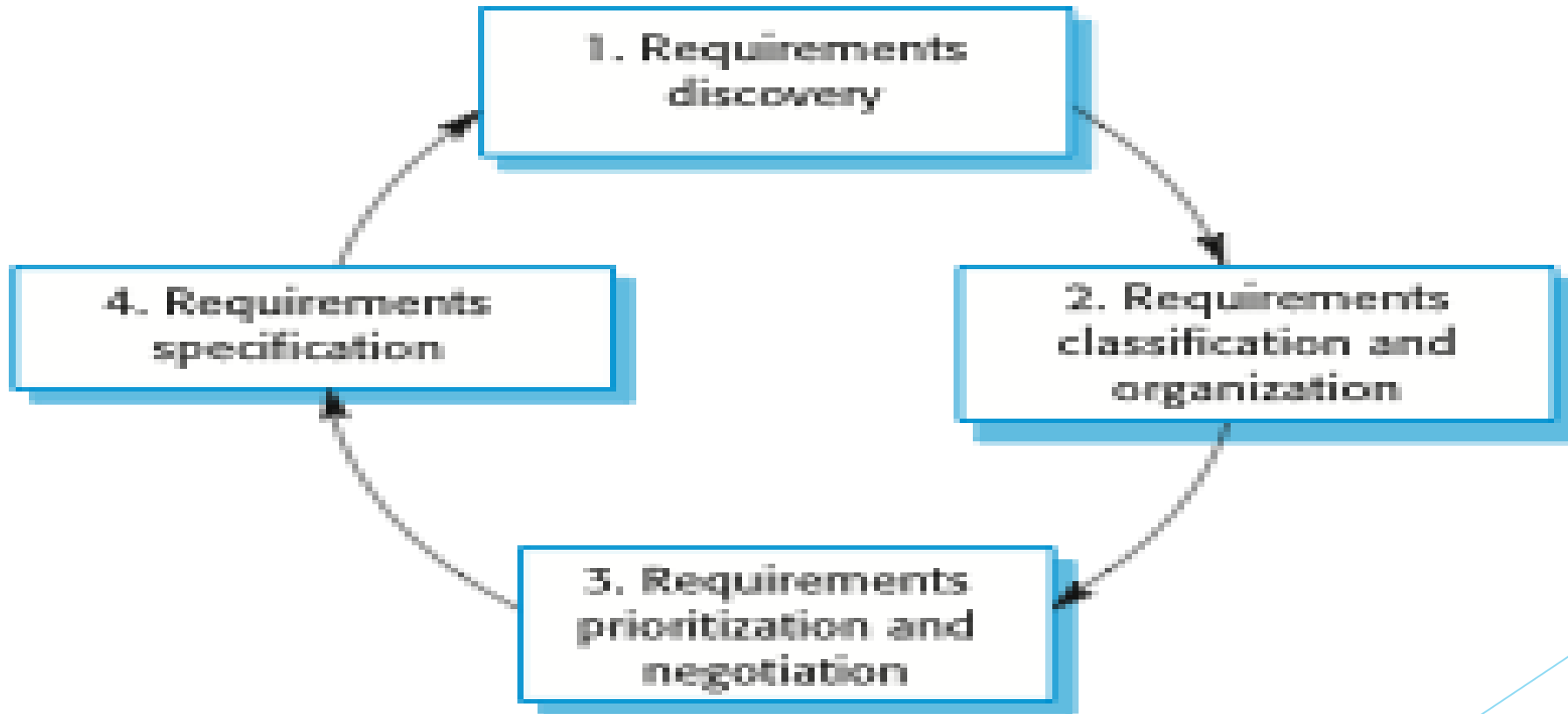


# Requirement Elicitation and Analysis

- **Requirements elicitation** is the process of gathering and defining the requirements for a software system.
- Goal: software development process is based on a clear and comprehensive understanding of the customer's needs and requirements.
- Requirements elicitation involves the **identification, collection, analysis, and refinement of the requirements** for a software system.
-

- 
- The background of the slide features an abstract design with overlapping geometric shapes in various shades of blue, ranging from light sky blue to deep navy blue. These shapes are primarily located on the right side of the slide, creating a modern, layered effect.
- ▶ It is a critical part of the software development life cycle and is typically performed at the beginning of the project.
  - ▶ Requirements elicitation involves stakeholders from different areas of the organization, including business owners, end-users, and technical experts.
  - ▶ The output of the requirements elicitation process is a set of clear, concise, and well-defined requirements that serve as the basis for the design and development of the software system.

# Elicitation process



# Requirement Discovery

- ▶ The process of gathering information about the required and existing systems and distilling the user and system requirements from this information.

## Interviewing

- ▶ **Types of interview**
  - **Closed interviews : stakeholders answers based on pre-determined list of questions**
  - **Open interviews : in which there is no predefined agenda, where various issues are explored with stakeholders**



## ► Effective interviewing

- Be open-minded, avoid pre-conceived ideas about the requirements and are willing to listen to stakeholders.
- Prompt the interviewee to get discussions going using a springboard question, a requirements proposal, or by working together on a prototype system
- **Springboard question example:** What do you believe are the most significant challenges our project faces right now?

► **Scenarios** are **real-life examples of how a system can be used.**

They should include

- A description of the starting situation;
- A description of the normal flow of events;
- A description of what can go wrong;
- A description of the state when the scenario finishes.

## ✓ **Ethnography**

- A social scientist spends a considerable time observing and analysing how people actually work.
- Social and organisational factors of importance may be observed.
- Requirements that are derived from cooperation and awareness of other people's activities.
- Awareness of what other people are doing leads to changes in the ways in which we do things.

# Requirement Specification

- The process of writing the user and system requirements in a requirements document.
- User requirements have to be understandable by end-users and customers who do not have a technical background.
- System requirements are more detailed requirements and may include more technical information.
- The requirements may be part of a contract for the system development
- **It is therefore important that these are as complete as possible.**

Notation	Description
Natural language	The requirements are written using numbered sentences in natural language. Each sentence should express one requirement.
Structured natural language	The requirements are written in natural language on a standard form or template. Each field provides information about an aspect of the requirement.
Design description languages	This approach uses a language like a programming language, but with more abstract features to specify the requirements by defining an operational model of the system. This approach is now rarely used although it can be useful for interface specifications.
Graphical notations	Graphical models, supplemented by text annotations, are used to define the functional requirements for the system; UML use case and sequence diagrams are commonly used.
Mathematical specifications	These notations are based on mathematical concepts such as finite-state machines or sets. Although these unambiguous specifications can reduce the ambiguity in a requirements document, most customers don't understand a formal specification. They cannot check that it represents what they want and are reluctant to accept it as a system contract

# Requirement Validation

- ▶ Requirements Validation Techniques are used to ensure that the **software requirements are complete, consistent, and correct.**
- ▶ **Common Techniques used in Software Engineering**
- ▶ **Inspection:** This technique involves reviewing the requirements document with a group of experts, looking for errors, inconsistencies, and missing information.
- ▶ **Walkthrough:** technique involves a group of experts reviewing the requirements document and walking through it line by line, discussing any issues or concerns that arise.
- ▶ **Formal Verification:** This technique involves mathematically proving that the requirements are complete and consistent and that the system will meet the requirements.
- ▶ **Model-Based Verification:** Model-Based Verification involves creating a model of the system and simulating it to see if it meets the requirements.

- ▶ **Prototyping:** This technique involves creating a working prototype of the system and testing it to see if it meets the requirements.
- ▶ **Black-box Testing:** technique involves testing the system without any knowledge of its internal structure or implementation, to see if it meets the requirements.
- ▶ **Acceptance Testing:** technique involves testing the system with real users to see if it meets their needs and requirements.
- ▶ **User Feedback:** This technique involves gathering feedback from the users and incorporating their suggestions and feedback into the requirements.

# Software Requirements Document(SRS document)

- The software requirements document is the official statement of what is required of the system developers.
- Should include both a definition of user requirements and a specification of the system requirements.
- It is NOT a design document. As far as possible, it should set of **WHAT the system should do rather than HOW it should do it.**

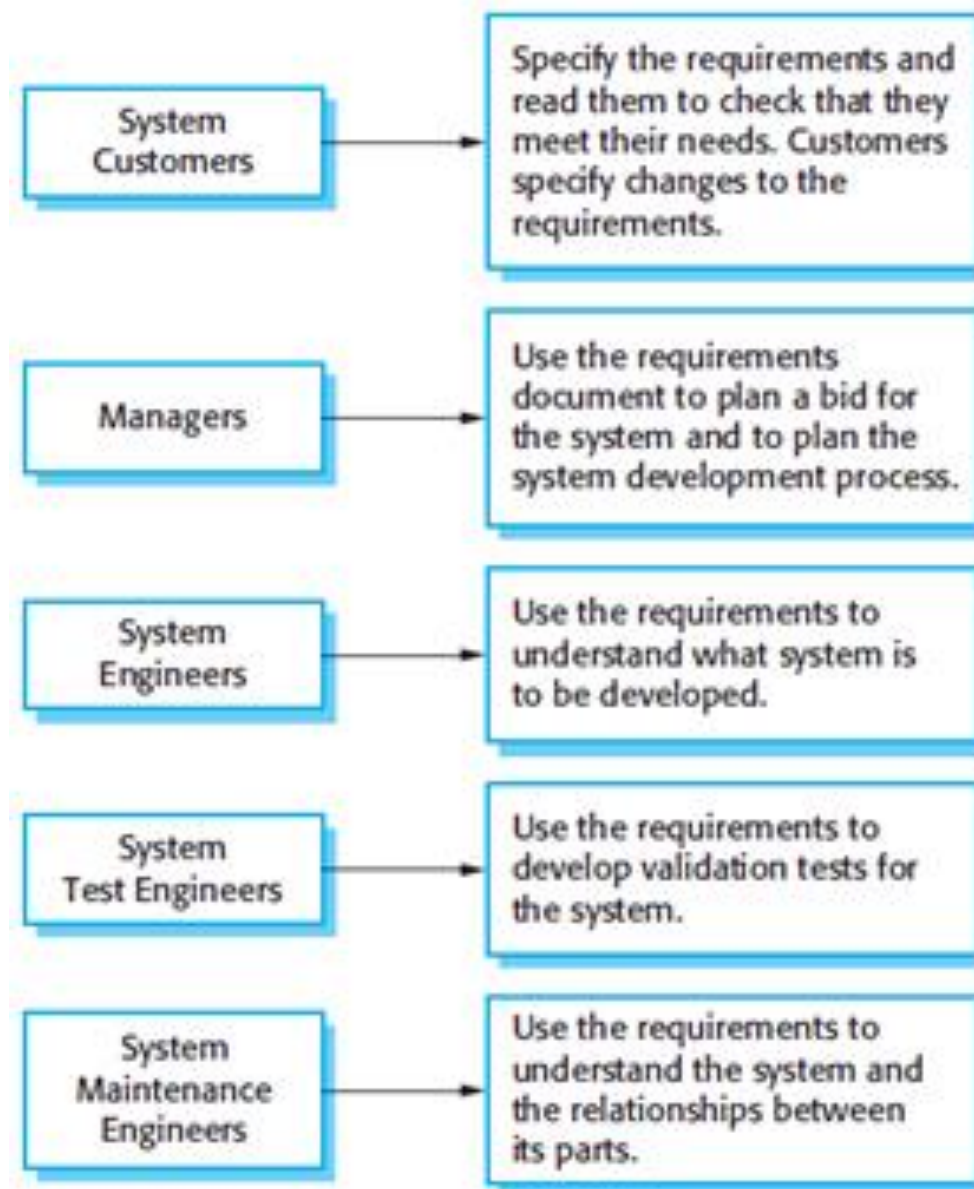


# The structure of a requirements document

Chapter	Description
Preface	This should define the expected readership of the document and describe its version history, including a rationale for the creation of a new version and a summary of the changes made in each version.
Introduction	This should describe the need for the system. It should briefly describe the system's functions and explain how it will work with other systems. It should also describe how the system fits into the overall business or strategic objectives of the organization commissioning the software.
Glossary	This should define the technical terms used in the document. You should not make assumptions about the experience or expertise of the reader.
User requirements definition	Here, you describe the services provided for the user. The nonfunctional system requirements should also be described in this section. This description may use natural language, diagrams, or other notations that are understandable to customers. Product and process standards that must be followed should be specified.
System architecture	This chapter should present a high-level overview of the anticipated system architecture, showing the distribution of functions across system modules. Architectural components that are reused should be highlighted.

Chapter	Description
System requirements specification	This should describe the functional and nonfunctional requirements in more detail. If necessary, further detail may also be added to the nonfunctional requirements. Interfaces to other systems may be defined.
System models	This might include graphical system models showing the relationships between the system components and the system and its environment. Examples of possible models are object models, data-flow models, or semantic data models.
System evolution	This should describe the fundamental assumptions on which the system is based, and any anticipated changes due to hardware evolution, changing user needs, and so on. This section is useful for system designers as it may help them avoid design decisions that would constrain likely future changes to the system.
Appendices	These should provide detailed, specific information that is related to the application being developed; for example, hardware and database descriptions. Hardware requirements define the minimal and optimal configurations for the system. Database requirements define the logical organization of the data used by the system and the relationships between data.
Index	Several indexes to the document may be included. As well as a normal alphabetic index, there may be an index of diagrams, an index of functions, and so on.

# Users of SRS



# SRS-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](http://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

## INFO

2. **Overall Description**
    - 2.1 Product Perspective
    - 2.2 Product Features
    - 2.3 User Classes and Characteristics
    - 2.4 Operating Environment
    - 2.5 Design and Implementation Constraints
    - 2.6 User Documentation
    - 2.7 Assumptions and Dependencies
  3. **System Features**
    - 3.1 System Feature 1
    - 3.2 System Feature 2 (and so on)
  4. **External Interface Requirements**
    - 4.1 User Interfaces
    - 4.2 Hardware Interfaces
    - 4.3 Software Interfaces
    - 4.4 Communications Interfaces
  5. **Other Nonfunctional Requirements**
    - 5.1 Performance Requirements
    - 5.2 Safety Requirements
    - 5.3 Security Requirements
    - 5.4 Software Quality Attributes
  6. **Other Requirements**
- Appendix A: Glossary**  
**Appendix B: Analysis Models**  
**Appendix C: Issues List**

A detailed description of each SRS topic can be obtained by downloading the SRS template at the URL noted in this sidebar.



# Traceability Matrix

- Traceability matrix allows a requirement engineer to represent the **relationship between requirements and other work products**.
- Rows of the matrix are labelled using requirement names and columns can be labelled with the name of Software Engg work product.

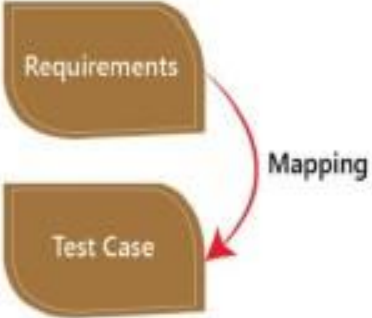
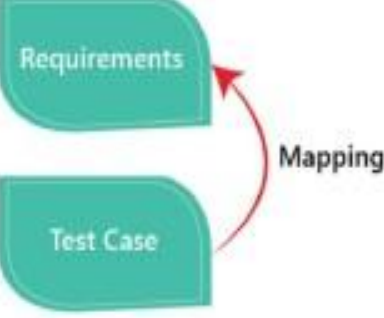
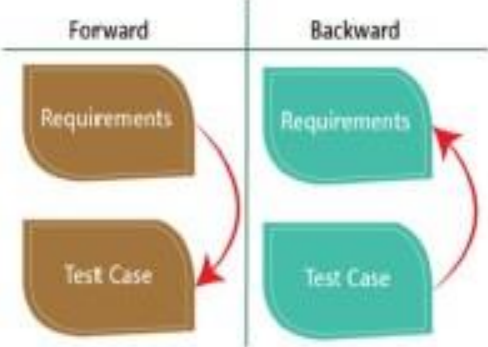
- A matrix cell is marked to indicate the presence of link between the two.
- A table type document that is used in the development of software application to trace requirements.
- It can be used for both forward (from Requirements to Design or Coding) and backward (from Coding to Requirements) tracing.

- It is also known as **Requirement Traceability Matrix (RTM)** or **Cross Reference Matrix (CRM)**.
- Map all the requirements and corresponding test cases to ensure that we have written all the test cases for each condition.

- It can be used to ensure the Engg work products have taken all requirements into account.
- As the no: of req and the number of work products grows.it become increasingly difficult to keep the traceability up to date.



Req No	Req Desc	Testcase ID	Status
123	Login to the application	TC01,TC02,TC03	TC01-Pass TC02-Pass
345	Ticket Creation	TC04,TC05,TC06, TC07,TC08,TC09 TC010	TC04-Pass TC05-Pass TC06-Pass TC06-Fail TC07-No Run
456	Search Ticket	TC011,TC012, TC013,TC014	TC011-Pass TC012-Fail TC013-Pass TC014-No Run

Forward Traceability	Backward Traceability	Bi-directional Traceability
<ul style="list-style-type: none"> <li>Used to ensure that every business's needs or requirements are executed correctly in the application and also tested rigorously.</li> <li>Requirements are mapped into the forward direction to the test cases.</li> </ul>	<ul style="list-style-type: none"> <li>Used to check that we are not increasing the space of the product by enhancing the design elements, code, test other things which are not mentioned in the business needs.</li> <li>Requirements are mapped into the backward direction to the test cases.</li> </ul>	<ul style="list-style-type: none"> <li>A combination of forwarding and backward traceability matrix.</li> <li>Used to make sure that all the business needs are executed in the test cases.</li> <li>Also evaluates the modification in the requirement which is occurring due to the bugs in the application.</li> </ul>
		

# Goals of Traceability Matrix:

- ◀ It ensures that the software completely meets the customer's requirements.
- ◀ It helps in detecting the root cause of any bug.

# Advantages of RTM:

- With the help of the RTM document, we can **display the complete test execution and bugs status based on requirements.**
- ◀ It is used to show the **missing requirements** or conflicts in documents.
- ◀ We can ensure the complete test coverage, which means all the **modules are tested.**
- ◀ It will also consider the **efforts of the testing teamwork towards reworking or reconsidering on the test cases.**

# Personas, Scenarios, User Stories , & Feature Identification

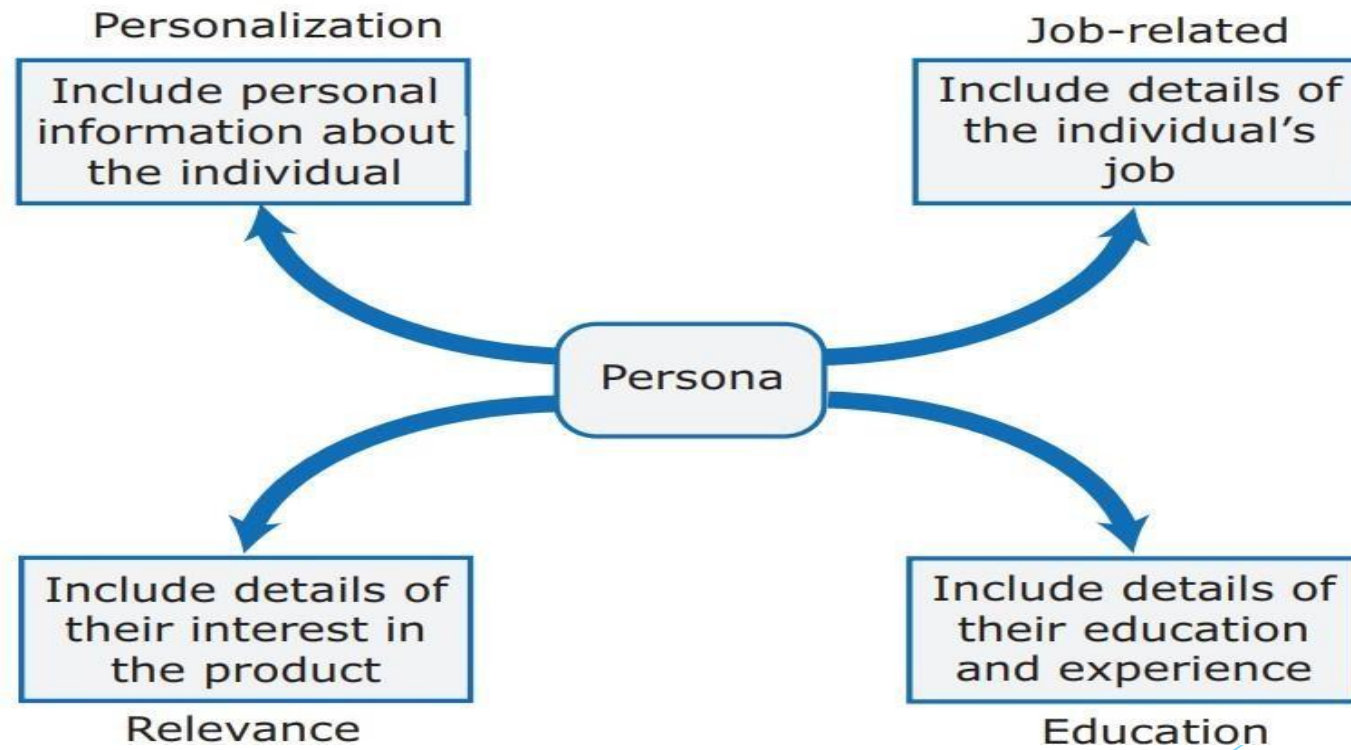


# PERSONAS

- Personas are about “imagined users,” character portraits of types of user that you think might adopt your product.
- Ex: if your product is aimed at managing appointments for dentists, you might create a dentist persona, a receptionist persona, and a patient persona.

- Personas of different types of users help to **imagine what these users may want to do with your software and how they might use it.**
- They also help you to investigate difficulties that users might have in understanding and using product features.
- There is no standard way to represent personas

**Figure 3.4** Persona descriptions

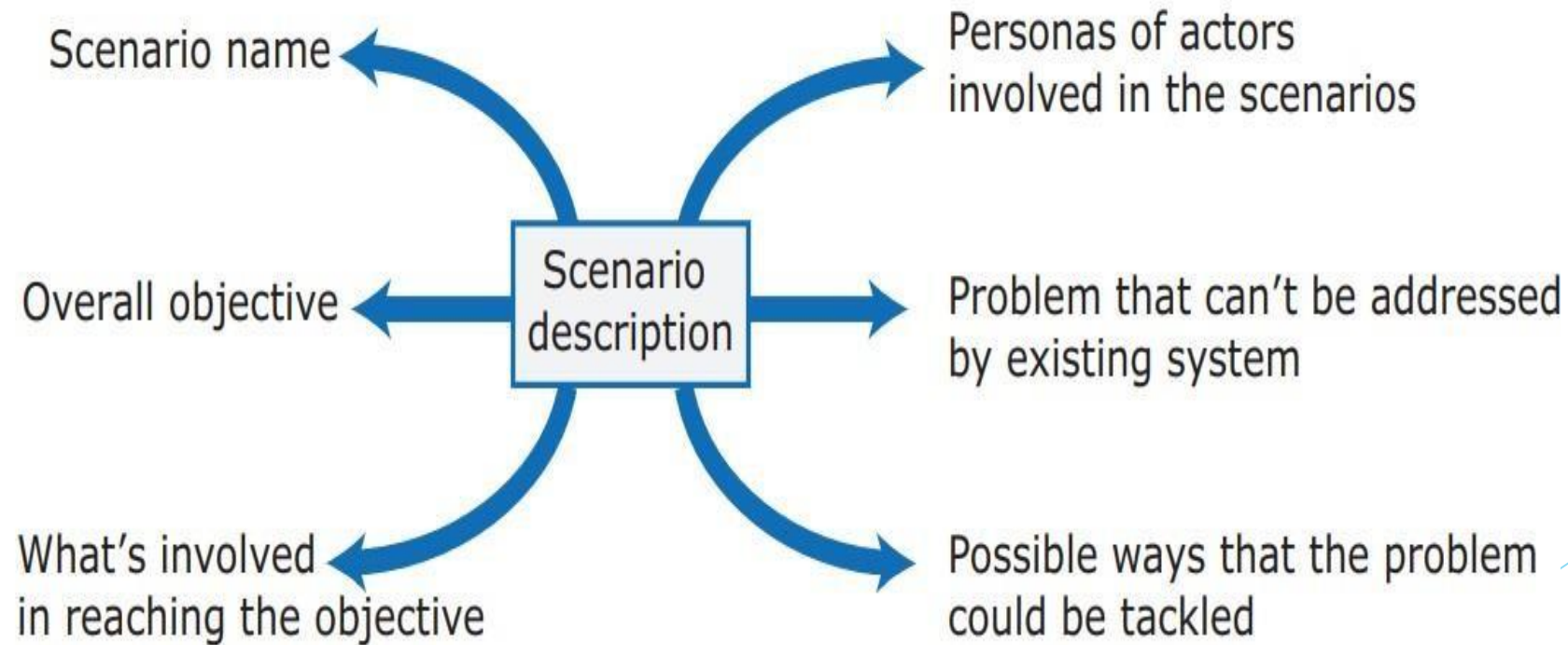




# SCENARIOS

- A scenario is a narration that describes **a situation in which a user is using your product's features to do something that they want to do.**
- Scenarios are used in the design of requirements and system features, in system testing, and in user interface design
- It should briefly explain **the user's problem and present an imagined way that the problem might be solved.**

**Figure 3.5** Elements of a scenario description



# User Stories

- ✓ These are finer-grain narratives that set out in a more **detailed and structured way a single thing that a user wants from a software system.**
- ✓ User stories are not intended for planning but for helping with feature identification.

## User Story for Withdrawing Cash

### User Story

As a **regular fintech app user**, I want to **withdraw cash** from my bank account using the fintech mobile app, so that I can **easily draw out cash even without using my card**.

### Acceptance Criteria

- User can login to the mobile app.
- User can enter the desired amount to withdraw.
- Desired amount is within the user's current balance.
- Accounts are updated concurrently after validation.
- Email and in-app confirmation of transactions are sent to the user.

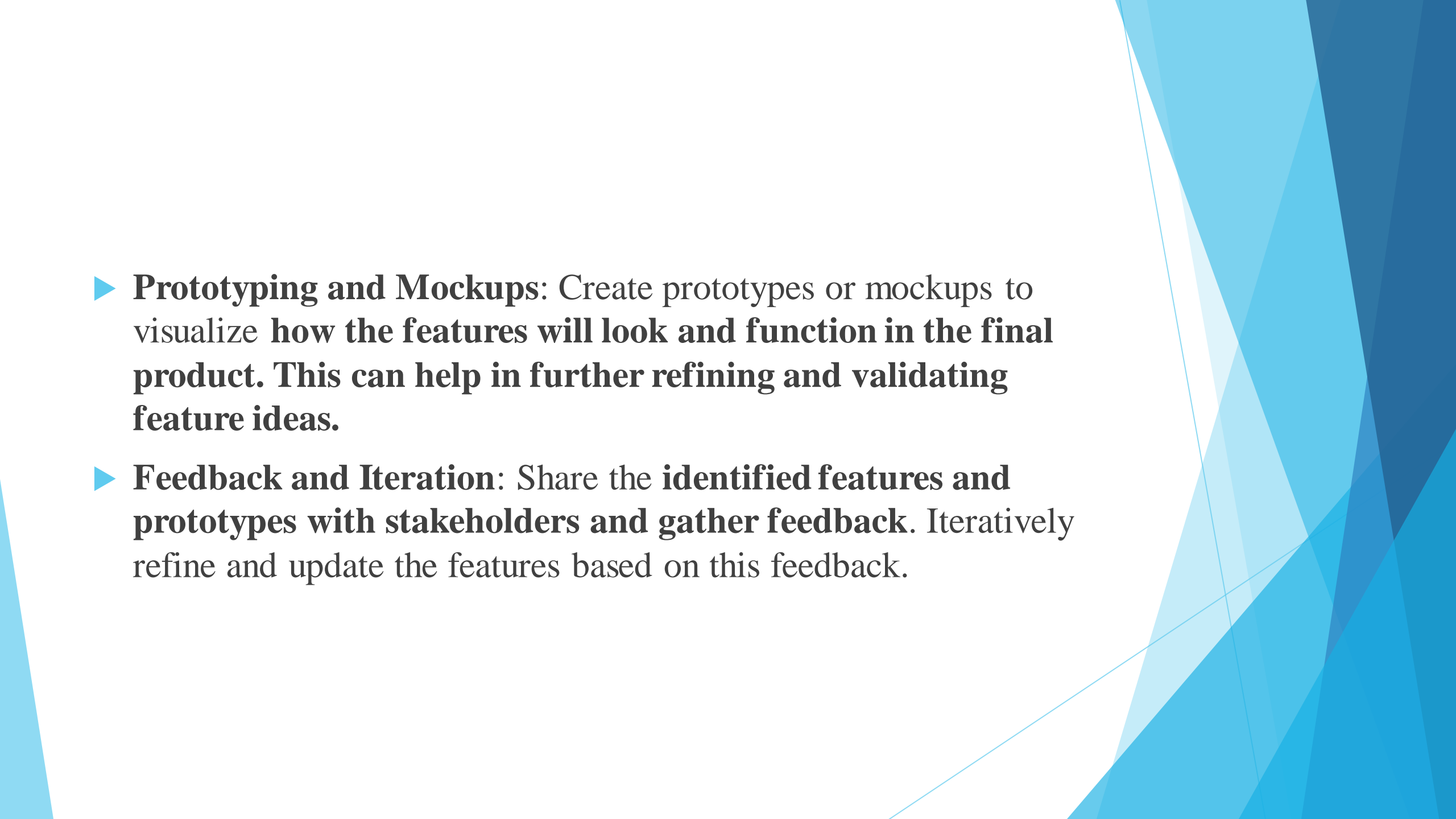
# Features

- ✓ Feature defines the functionality of the system.
- ✓ Feature is a fragment of functionality that implements some user or system need.
- ✓ **Feature is something that the user needs or wants.**

# Feature identification

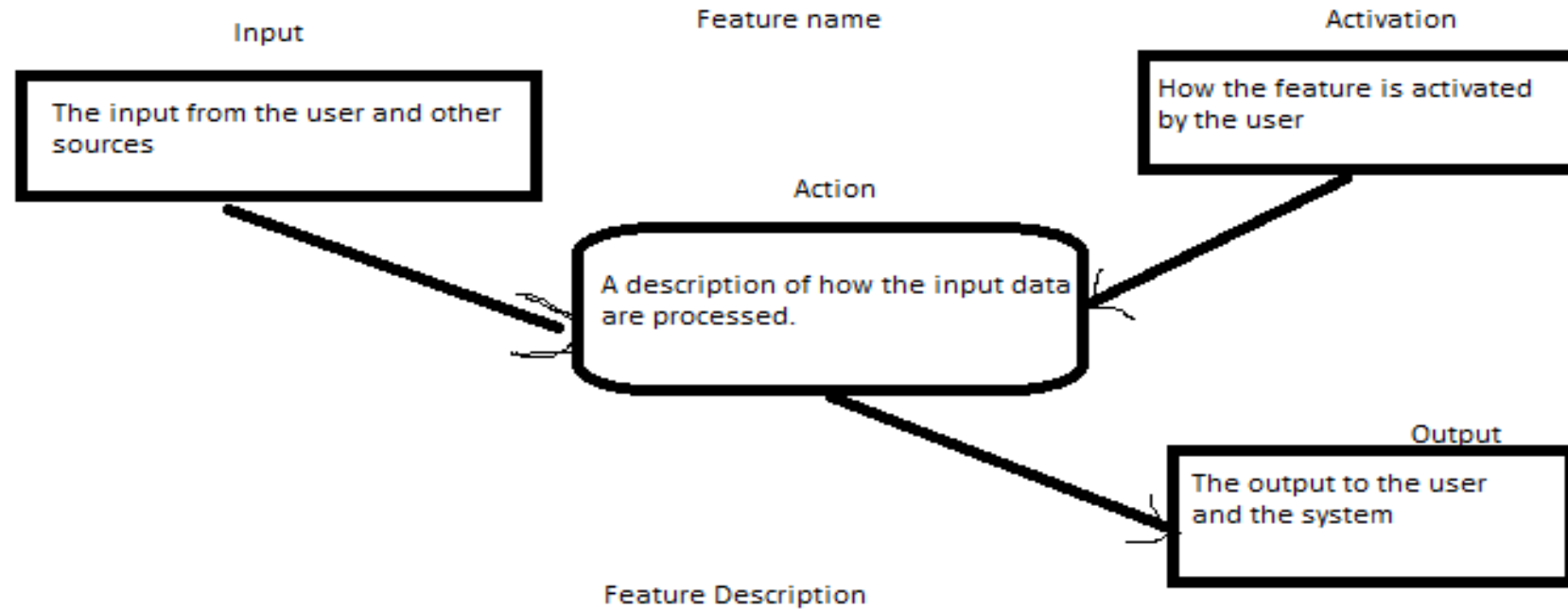
- ▶ It is a crucial step in software development and product management. **It involves identifying, defining, and documenting the specific functionalities or capabilities that a software product should have.**
- ▶ These features are essentially the building blocks of the software and are essential for meeting the project's goals and satisfying user needs.
- ▶ Here's how feature identification typically works:

- ▶ **Stakeholder Input:** Gather input from various stakeholders, including clients, end-users, marketing teams.
- ▶ **Market Research:** If applicable, conduct market research to identify what features are in **demand or are competitive advantages in the industry**. Analyze the competition to understand what features they offer.
- ▶ **User Personas:** Create **user personas or profiles to represent the different types of users who will interact with the software**. This helps in **tailoring features to specific user needs and preferences**.

- 
- The background of the slide features abstract, overlapping geometric shapes in various shades of blue, ranging from light sky blue to deep navy blue. These shapes create a dynamic, modern look on the right side of the slide.
- ▶ **Prototyping and Mockups:** Create prototypes or mockups to visualize **how the features will look and function in the final product.** This can help in further refining and validating feature ideas.
  - ▶ **Feedback and Iteration:** Share the **identified features and prototypes with stakeholders and gather feedback.** Iteratively refine and update the features based on this feedback.



# Feature Description



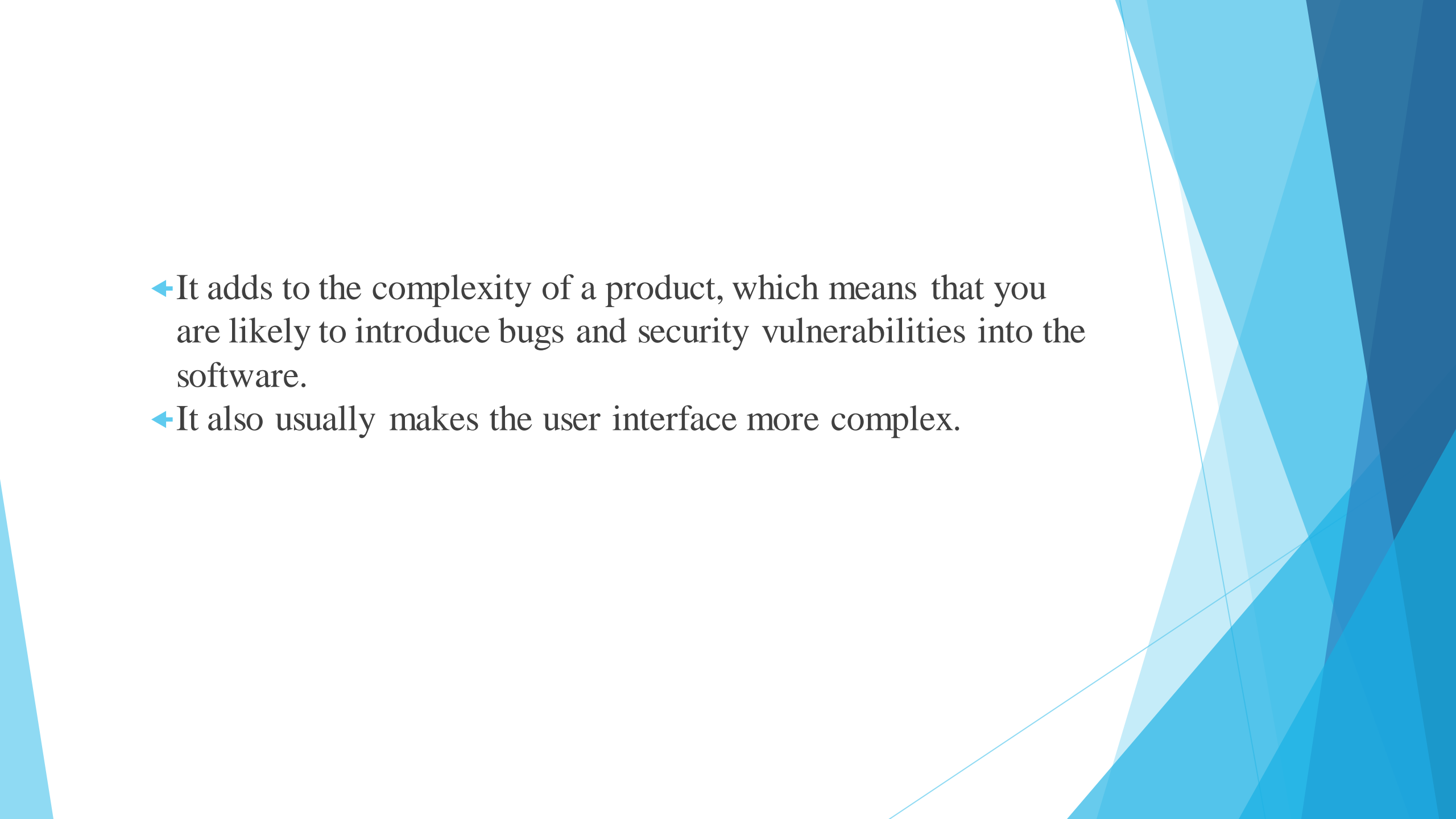
# Knowledge Require for Feature Design

**Table 3.6** Knowledge required for feature design

Knowledge	Description
User knowledge	You can use user scenarios and user stories to inform the team of what users want and how they might use the software features.
Product knowledge	You may have experience of existing products or decide to research what these products do as part of your development process. Sometimes your features have to replicate existing features in these products because they provide fundamental functionality that is always required.
Domain knowledge	This is knowledge of the domain or work area (e.g., finance, event booking) that your product aims to support. By understanding the domain, you can think of new innovative ways of helping users do what they want to do.
Technology knowledge	New products often emerge to take advantage of technological developments since their competitors were launched. If you understand the latest technology, you can design features to make use of it.

# Feature creep

- ← Feature creep, also known as scope creep, refers to the phenomenon in software engineering and project management where **additional features or requirements are added to a project without proper assessment, planning, or approval.**
- ← These additional features often go beyond the original project scope and can lead to various problems, including project delays, increased costs, and decreased overall project quality.

- 
- The background of the slide features an abstract design composed of various shades of blue and white. On the right side, there are overlapping, semi-transparent geometric shapes, primarily triangles and polygons, in different tones of blue, ranging from light sky blue to deep navy blue. These shapes create a dynamic, layered effect. The left side of the slide is mostly white, providing a clean space for the text.
- ◀ It adds to the complexity of a product, which means that you are likely to introduce bugs and security vulnerabilities into the software.
  - ◀ It also usually makes the user interface more complex.

# Avoiding Feature Creep

**Figure 3.10** Avoiding feature creep

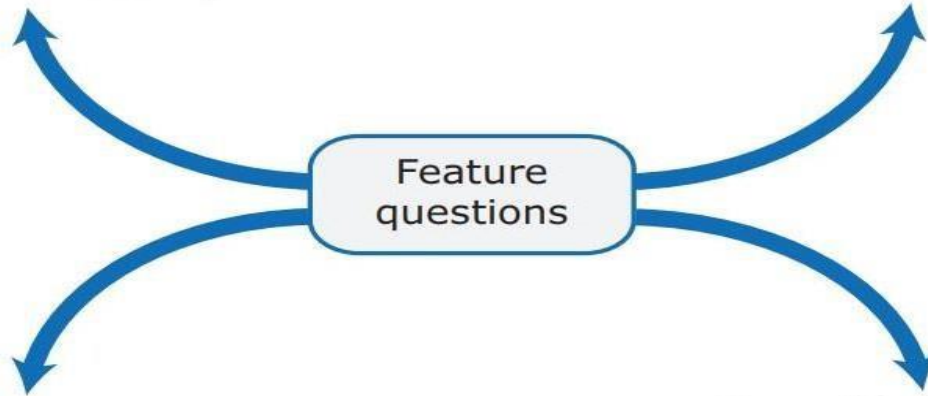
Does this feature really add anything new or is it simply an alternative way of doing something that is already supported?

Is this feature likely to be important to and used by most software users?

Feature questions


Can this feature be implemented by extending an existing feature rather than adding another feature to the system?

Does this feature provide general functionality or is it a very specific feature?



# Feature List

- ▶ The output of the feature identification process should be a list of features that you use for designing and implementing your product.
- ▶ A feature list that outlines **the functionality and capabilities** that a software product or application is intended to have.

- 
- ▶ **Feature Name:** title for each feature.
  - ▶ **Feature Description:** what the feature is and what it does.
  - ▶ **Use Cases:** Scenarios or situations where the feature will be used.

- ▶ **User Interface (UI) Requirements:** If the feature involves user interaction, specify any UI elements, such as buttons, forms, or menus, and how they should behave.
- ▶ **Data Requirements:** Specify the data that the feature will work with, including data sources, data formats, and any data storage or retrieval requirements.
- ▶ **Functional Requirements:** includes **input/output specifications, data processing, and any algorithms or logic** involved.

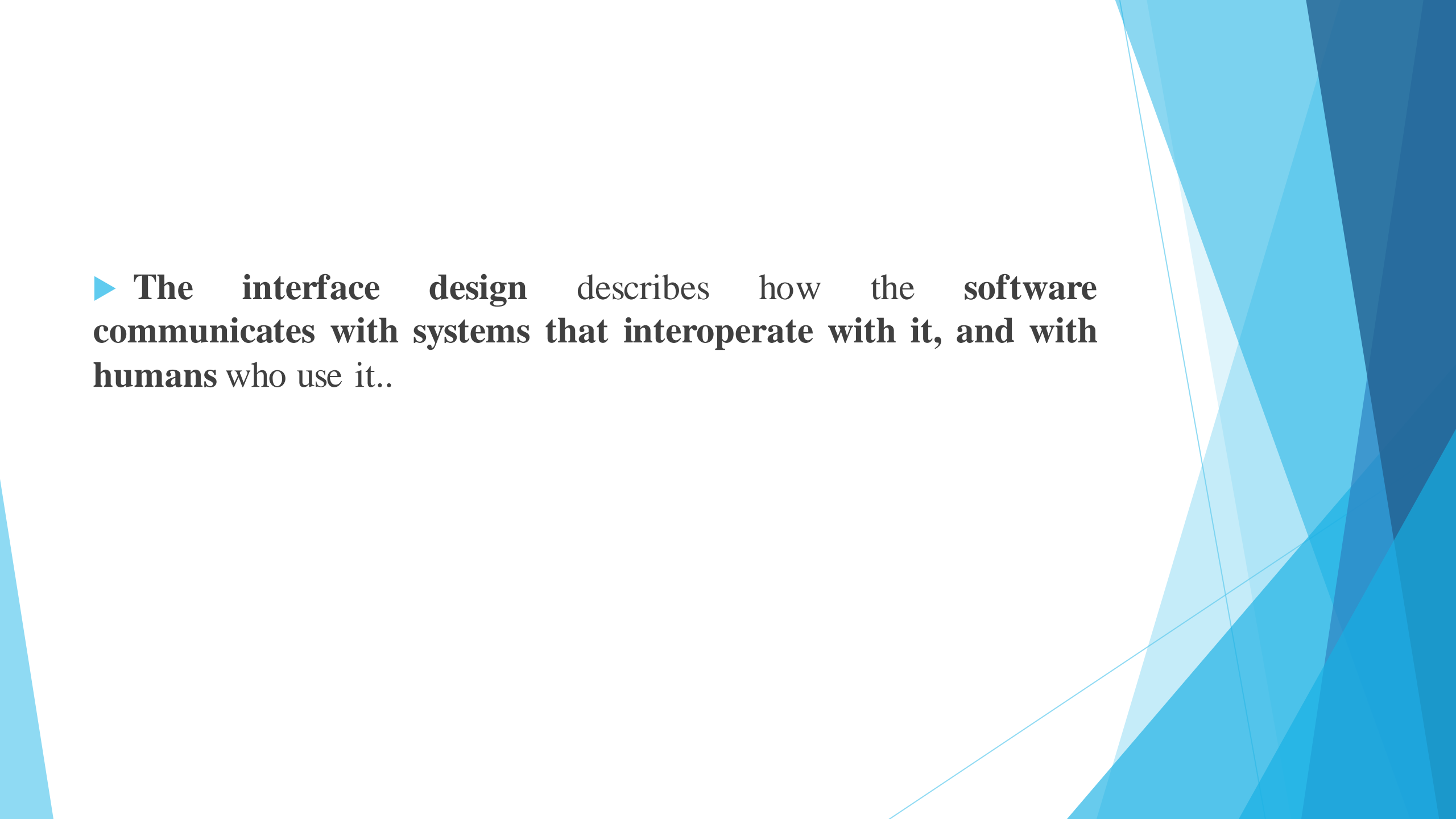


# Design concepts - Design within the context of software engineering, Design Process, Design concepts, Design Model.

- ▶ Software design encompasses the set of principles, concepts, and practices that lead to the development of a high-quality system or product.
- ▶ It is the place where creativity rules—where stakeholder requirements, business needs, and technical considerations all come together in the formulation of a product or system.
- ▶ Design creates a representation or **model of the software**, the **design model provides detail about software architecture, data structures, interfaces, and components that are necessary to implement the system.**

# DESIGN WITHIN THE CONTEXT OF SOFTWARE ENGINEERING

- ▶ **The data/class design:** Using high-level class models & refine them into concrete design class realizations ie. **determine the data structures needed to support those classes, define how objects of these classes will behave etc**
- ▶ **The architectural design** defines the relationship between major structural elements of the software. The architectural design representation—is the **overall framework** of a computer-based system.

The background of the slide features abstract, overlapping geometric shapes in various shades of blue, ranging from light sky blue to deep navy blue. These shapes are primarily located on the right side and bottom of the frame, creating a modern, dynamic feel.

▶ **The interface design** describes how the software communicates with systems that interoperate with it, and with humans who use it..

The background features abstract, overlapping geometric shapes in various shades of blue, ranging from light sky blue to deep navy blue. These shapes are primarily located on the right side of the slide, creating a modern, layered effect.

▶ The **component-level design** transforms structural elements of the software architecture into a **procedural description of software components**.

# THE DESIGN PROCESS

- ▶ Software design: requirements are translated into a “blueprint” for constructing the software. Initially, the blueprint depicts a holistic view of software
- ▶ **Quality Guidelines.**
- ▶ Consider the following guidelines:
  1. A design should exhibit an architecture that (1) has been created using **recognizable architectural styles or patterns**

- A design should be **modular**; that is, the software should be logically partitioned into elements or subsystems.
- A design should contain **distinct representations of data, interfaces, and components**.

- A design should lead to **interfaces that reduce the complexity of connections between components and with the external environment.**
- A design should be represented using a **notation** that effectively communicates its meaning.

# Assessing Design Quality—the Technical Review

- During design, quality is assessed by conducting a series of technical reviews (TRs).
- A technical review is a meeting conducted by members of the software team.
- Usually two, three, or four people participate depending on the scope of the design information to be reviewed.



# Quality Attributes

- **Functionality** is assessed by evaluating the feature set and capabilities of the program, the generality of the functions that are delivered, and the security of the overall system.
- **Reliability** is evaluated by measuring the frequency of failure, the accuracy of output results, the mean-time-to-failure (MTTF), the ability to recover from failure, and the predictability of the program

- .
- **Performance** is measured using processing speed, response time, resource consumption, throughput, and efficiency.
- **Supportability** combines extensibility, adaptability, and serviceability.
- *Extensibility is a measure of the ability to extend a system and the level of effort required to implement the extension.*
- *Adaptability is the extent to which a software system adapts to change in its environment.*
- *Serviceability the ease with which a deployed system can be maintained.*

# Common characteristics: Design Process

- (1) A mechanism for the translation of the requirements model into a design representation,
- (2) A notation for representing functional components and their interfaces,
- (3) Heuristics for refinement and partitioning
- (4) Guidelines for quality assessment.

# DESIGN CONCEPTS

## ► 1. Abstraction

- It involves simplifying and summarizing information to focus on essential aspects while ignoring unnecessary details.

A *procedural abstraction* refers to a **sequence of instructions that have a specific and limited function**. The name of a procedural abstraction implies these functions, but **specific details are suppressed**.

- ▶ *An example of a procedural abstraction would be the word open for a door. Open implies a long sequence of procedural steps (e.g., walk to the door, reach out and grasp knob, turn knob and pull door, step away from moving door, etc.).*

- A **data abstraction** is a named collection of data that describes a data object.
- *Example: data abstraction for **door** would encompass a set of attributes that describe the door (e.g., door type, swing direction, opening mechanism, weight, dimensions).*

# Design concepts (cont..)

► 2. *Software architecture* “the overall structure of the **software**”. Architecture is the structure or organization of program components (modules), the manner in which these components interact, and the structure of data that are used by the components.

▶ *Shaw and Garlan describe a set of properties that should be specified as part of an architectural design.*

➤ ***Structural properties*** define “the **components of a system** (e.g., modules, objects, filters) and the manner in which those components are **packaged and interact with one another.**”

➤ ***Extra-functional properties*** address “how the design architecture **achieves** requirements **for performance, capacity, reliability, security, adaptability,** and other system characteristics.

➤ ***Families of related systems*** “draw upon repeatable patterns that are commonly encountered in the design of families of similar systems.”



▶ *Given the specification of these properties, the architectural design can be represented using one or more of a number of different models.*

➤ ***Structural models*** represent architecture as an organized collection of program components.

➤ ***Framework models*** developed by identify repeatable architectural design frameworks (patterns) that are encountered in similar types of applications.

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

## ► Patterns

- A pattern is a proven solution to a recurring problem .
- A design pattern describes a design structure that solves a particular design problem within a specific context and how the pattern is applied and used.

- The intent of each design pattern is to provide a description that enables a designer to determine
  - (1) Whether the pattern is applicable to the current work,
  - (2) Whether the pattern can be reused (hence, saving design time), and
  - (3) Whether the pattern can serve as a guide for developing a similar, but functionally or structurally different pattern.

## ► Separation of Concerns

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

► A ***concern*** is a feature or behavior that is specified as part of the requirements model for the software. By separating concerns into smaller, and therefore more manageable pieces, a problem takes less effort and time to solve.

## ► Modularity

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

## ► Information Hiding

- The principle of *information hiding* suggests that modules be “characterized by design decisions that (each) hides from all others.”
- Hiding implies that effective modularity can be achieved by defining a set of independent modules that communicate with one another only that information necessary to achieve software function.

## ► Functional Independence

Functional independence is attained by designing modules that focus on **a single task and limit interactions with other modules**

Independence is assessed using two qualitative criteria: **cohesion and coupling**. *Cohesion* is an indication of the relative functional strength of a module. *Coupling* is an indication of the relative interdependence among modules.



- A **cohesive** module performs a single task, requiring little interaction with other components in other parts of a program
- **Coupling** is an indication of interconnection among modules in a software structure.
- Coupling depends on the interface complexity between modules, the point at which entry or reference is made to a module, and what data pass across the interface.
- High cohesion and low coupling make the module to be effectively design.

## ► Aspects

An aspect is implemented as a **separate module** (component) rather than as software fragments that are “scattered” or “tangled” throughout many components.

*Eg: Logging that involves recording events, errors, or information during the execution of a software application. It's often managed separately from the main program logic and can affect multiple parts of the codebase.*

## ► Refactoring

- *Refactoring* is a reorganization technique that simplifies the design (or code) of a component without changing its function or behavior.

▶ **Design Classes:** The model defines a set of classes used in the design.

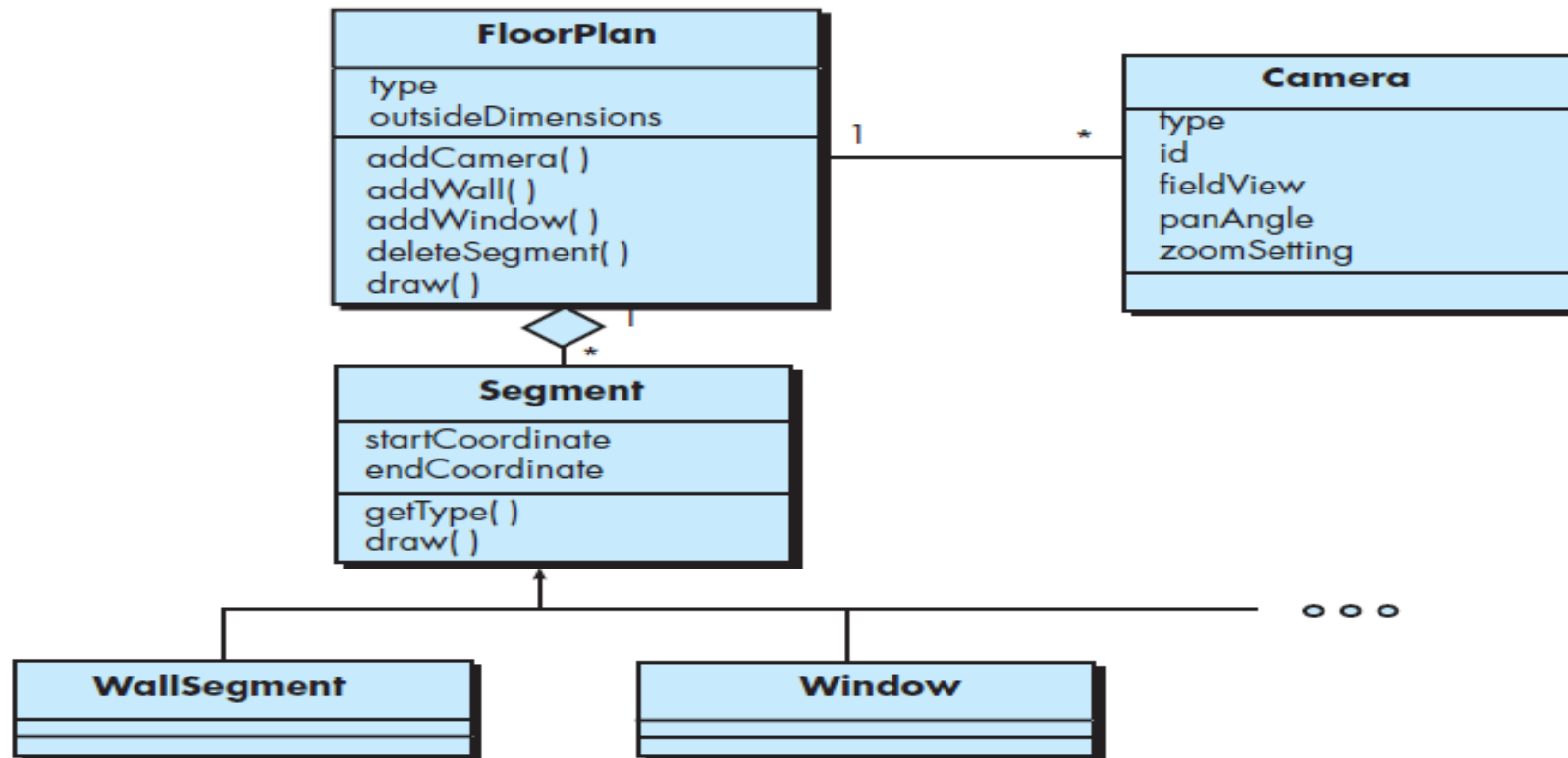
▶ *User interface classes* define all abstractions that are necessary for human-computer interaction (HCI)

▶ *Business domain classes* to implement some element of the business domain with its attribute and services

*Eg: Product: Represents items for sale, including attributes like name, price, and description.*

- *Process classes* behavior and properties of multiple instances of a related set of objects
- *Persistent classes* represent data stores (e.g., a database)
- *System classes* implement software management and control functions that enable the system to **operate and communicate within its environment and with the outside world.**

# Design class for Floor Plan



# Design models -Architectural design models

► Architectural design represents the **structure of data and program components** that are required to build a computer-based system.


## ► **SOFTWARE ARCHITECTURE**

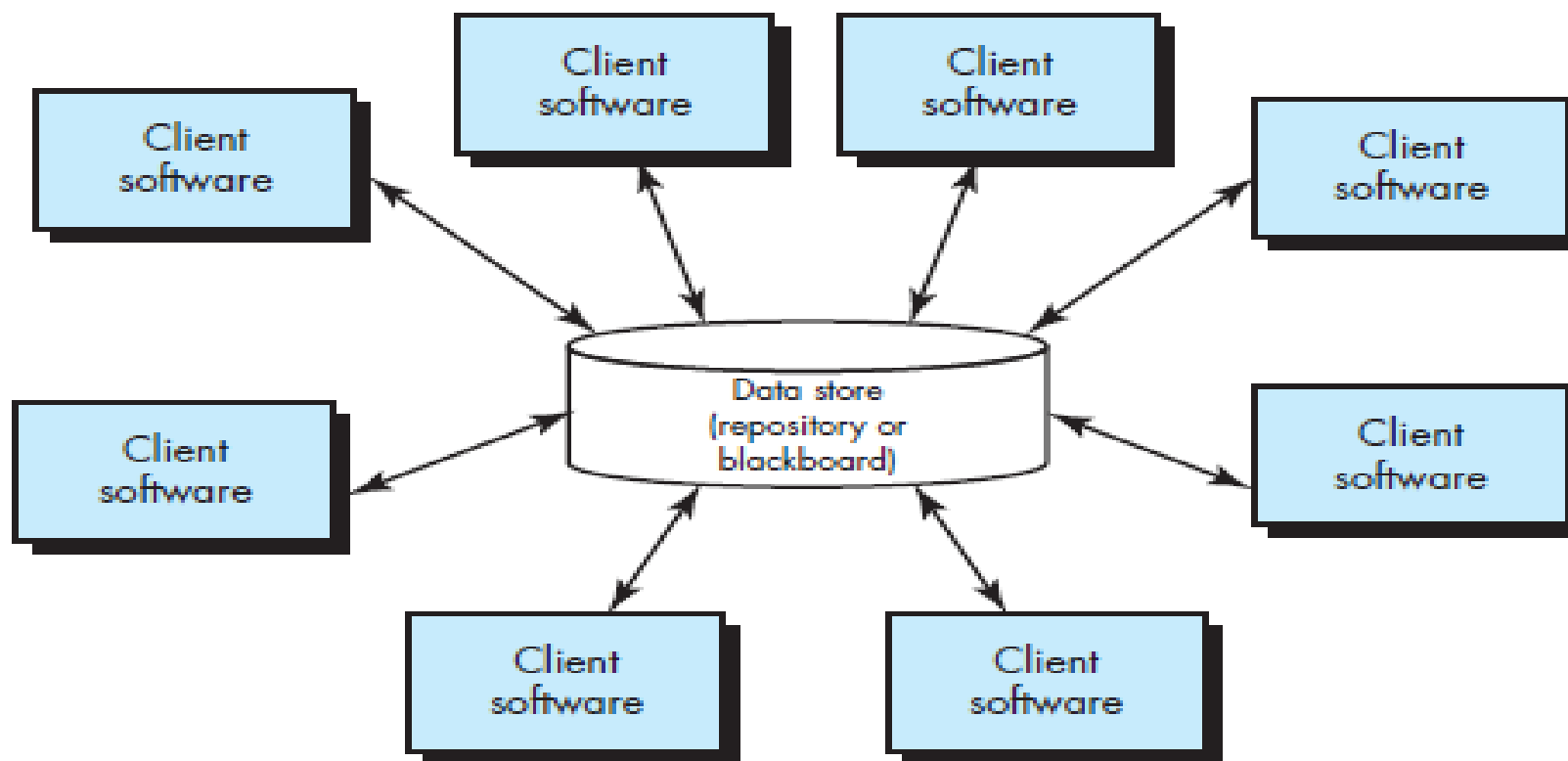
The software architecture which comprise **software components, the externally visible properties of those components, and the relationships among them.**

# Different Architectural Styles

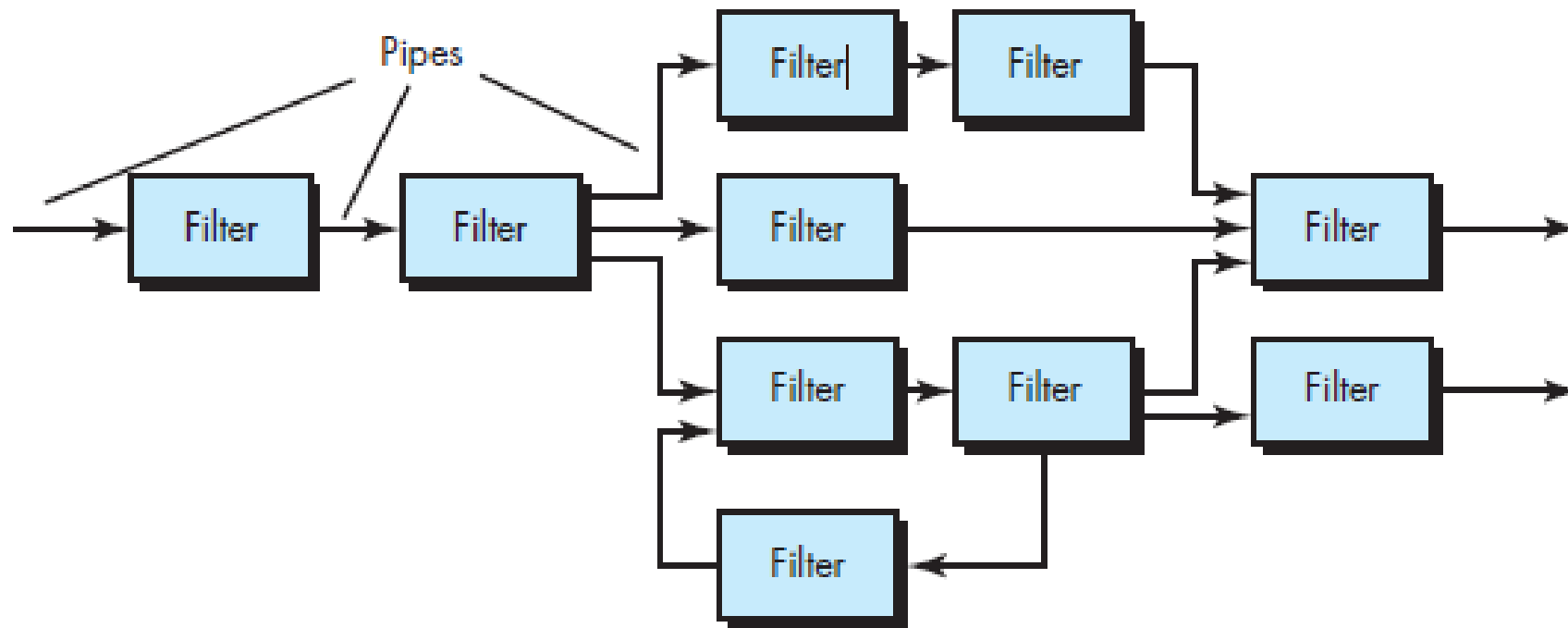
- **Data-Centered Architecture:** A data store (e.g., a file or database) resides at the center of this architecture and is accessed frequently by other components that **update, add, delete, or otherwise modify data** within the store.
- Figure illustrates a typical data-centered style. Client software accesses a central repository. In some cases the data repository is passive.



- 
- That is, client software accesses the data independent of any changes to the data or the actions of other client software.
  - A variation on this approach transforms the repository into a “blackboard” that sends notifications to client software when data of interest to the client changes.



- **Data-Flow Architectures:** This architecture is applied when input data are to be transformed through a series of computational or manipulative components into output data.
- A **pipe-and-filter** pattern has a set of components, called *filters*, connected by *pipes* that transmit data from one component to the next.



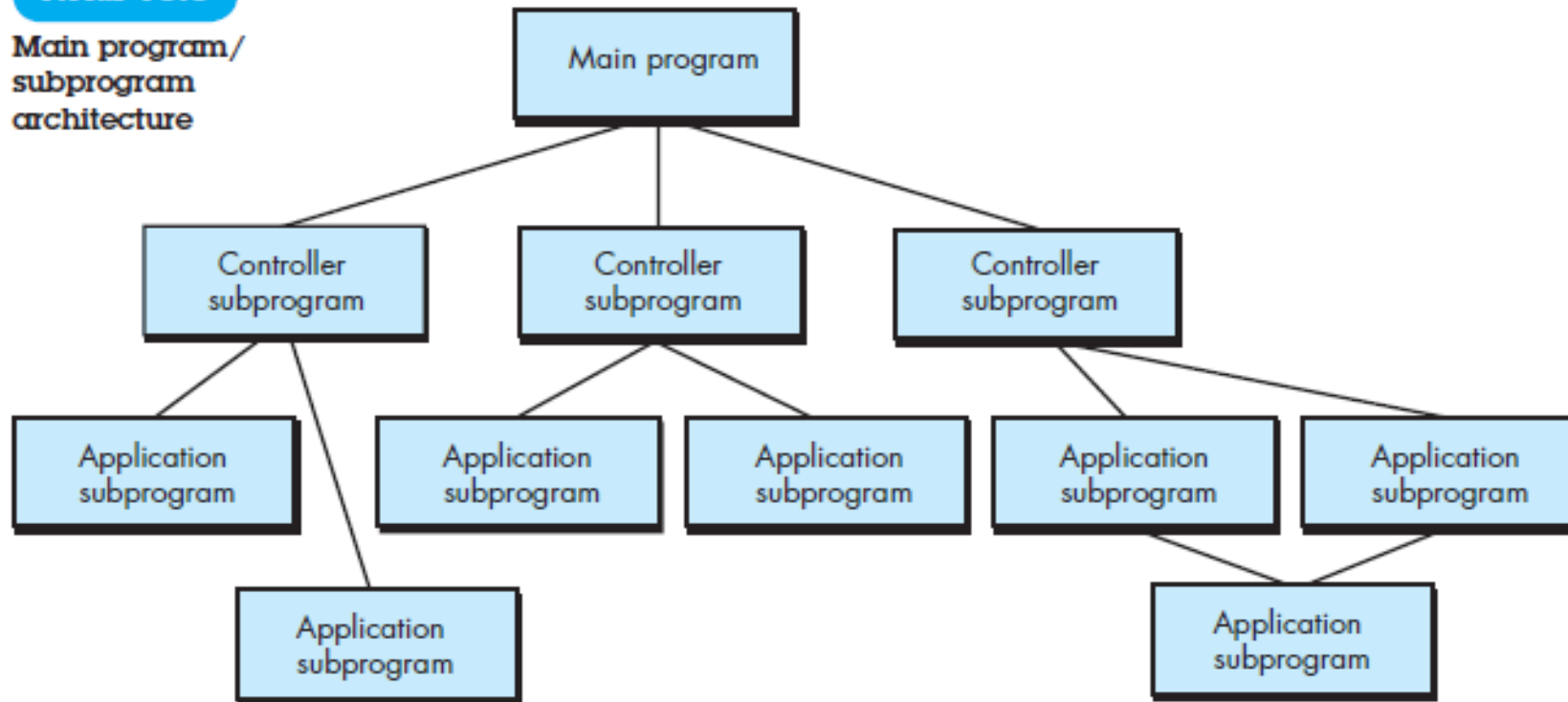
Pipes and filters

➤ **Call and Return Architectures:** This architectural style enables 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.* A “main” program invokes a number of program components, which 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.

**FIGURE 13.3**

Main program/  
subprogram  
architecture

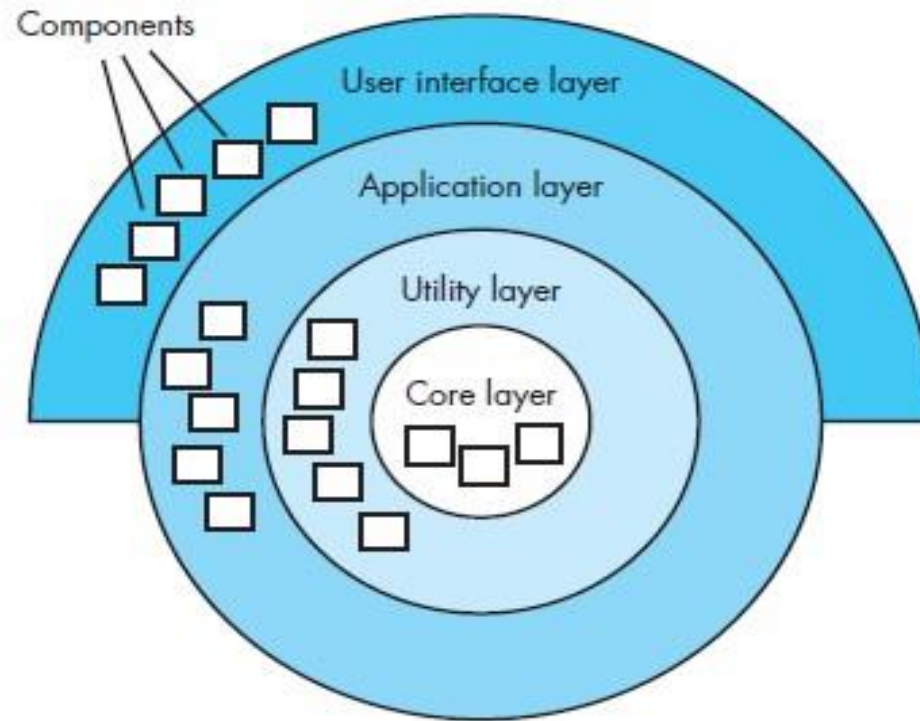


## ➤ Layered Architectures:

- The basic structure of a layered architecture is illustrated. A number of different layers are defined.
- 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**

**FIGURE 13.4**

Layered  
architecture





# ARCHITECTURAL CONSIDERATIONS

- **Economy** The best software is uncluttered and relies on abstraction to reduce **unnecessary detail**.
- **Visibility** Poor visibility arises when important design and domain concepts are poorly communicated.
- **Symmetry** —Architectural symmetry implies that a system is consistent and balanced in its attributes.
  - *As an example of architectural symmetry, consider a customer account object whose life cycle is modeled directly by a software architecture that requires both open () and close() methods.*

- **Emergence** —Emergent, self-organized behavior and control are often the key to creating scalable, efficient, and economic software architectures. For example, many real-time software applications are event driven. **The sequence and duration of the events that define the system's behavior is an emergent quality.**

# Architectural Context Diagram

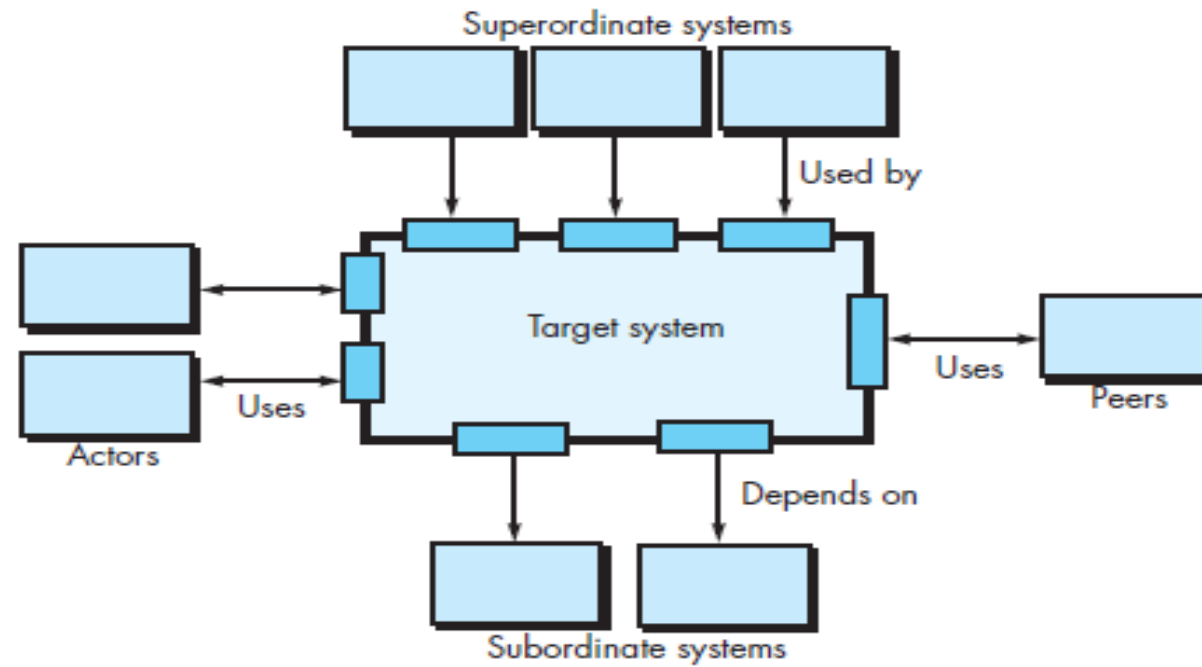
## ► Representing the System in Context

At the architectural design level, a software architect uses an *architectural context diagram (ACD)* to model the manner in which software interacts with entities external to its boundaries

**FIGURE 13.5**

**Architectural  
context  
diagram**

Source: Adapted from  
[Bos00].



# Elements of ACD

- *Superordinate systems* —those systems that **use the target system** as part of some higher-level processing scheme.
- *Actors* —entities (people, devices) that **interact with the target system** by producing or consuming information that is necessary for requisite processing.

- *Subordinate systems* —those systems that are **used by the target system and provide data or processing** that are necessary to complete target system functionality.
- *Peer-level systems* —**those systems that interact on a peer-to-peer basis** (i.e., information is either produced or consumed by the peers and the target system)

# Defining Archetypes

- ▶ An archetype, is a well-established pattern, template, or recognized solution that represents a commonly recurring design structure or concept.
- ▶ Archetypes are not specific implementations but rather high-level, abstract representations of design elements.

# Example of Archetypes:

► *Safe Home* security function, you might define the following archetypes:

- **Indicator.** An abstraction that represents all mechanisms (e.g., alarm siren, flashing lights, bell) for indicating that an alarm condition is occurring.



# Architectural Design for Web Apps

- WebApps are client-server applications typically structured using multilayered architectures, including a user interface or view layer, a controller layer which directs the flow of information to and from the client browser based on a set of business rules, and a content or model layer that may also contain the business rules for the WebApp.
- The user interface for a WebApp is designed around the characteristics of the web browser running on the client machine (usually a personal computer or mobile device). Data layers reside on a server. Business rules can be implemented using a server-based scripting language such as PHP or a client-based scripting language such as JavaScript. An architect will examine requirements for security and usability to determine which features should be allocated to the client or server.

- The architectural design of a WebApp is also influenced by the structure (linear or nonlinear) of the content that needs to be accessed by the client. The architectural components (Web pages) of a WebApp are designed to allow control to be passed to other system components, allowing very flexible navigation structures. The physical location of media and other content resources also influences the architectural choices made by software engineers.

# Architectural Design for Mobile Apps

- Mobile apps are typically structured using multilayered architectures, including a user interface layer, a business layer, and a data layer. With mobile apps you have the choice of building a thin Web-based client or a rich client. With a thin client, only the user interface resides on the mobile device, whereas the business and data layers reside on a server. With a rich client all three layers may reside on the mobile device itself.
- Mobile devices differ from one another in terms of their physical characteristics (e.g., screen sizes, input devices), software (e.g., operating systems, language support), and hardware (e.g., memory, network connections). Each of these attributes shapes the direction of the architectural alternatives that can be selected.

- A number of considerations that can influence the architectural design of a mobile app:  
(1) the type of web client (thin or rich) to be built, (2) the categories of devices (e.g., smart phones, tablets) that are supported, (3) the degree of connectivity (occasional or persistent) required, (4) the bandwidth required, (5) the constraints imposed by the mobile platform,
- ▶ (6) the degree to which reuse and maintainability are important, and (7) device resource constraints (e.g., battery life, memory size, processor speed).

# COMPONENT LEVEL DESIGN

- ▶ A component is a modular **building block** for computer software. The Unified Modeling Language Specification defines a component as “**a modular, deployable, and replaceable part of a system that encapsulates implementation**”
- ▶ Because components reside within the software architecture, they must **communicate and collaborate with other components and with entities** (e.g., other systems, devices, and people) that exist outside the boundaries of the software.

- ▶ Three important views of what a component is and how it is used as design modeling proceeds

## **An Object-Oriented View**

- ▶ In the context of object-oriented software engineering, **a component contains a set of collaborating classes**. Each class within a component include **all attributes and operations** that are relevant to its implementation.

## The Traditional View

► A traditional component called a *module*

(1) *Control component* that coordinates all other problem domain components

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

(3) *Infrastructure component* that is responsible for functions that support the processing required in the problem.

## ► A Process-Related View

**Created with reusability:** We use components or design , also provide the **details to use** them to populate the architecture , the **function(s) they perform**, and the **communication and collaboration they require**.



# DESIGN CLASS BASED COMPONENTS-

## Basic design principles

### ► *The Open-Closed Principle (OCP):*

*“A module (component) should be open for extension but closed for modification”*

Should specify the component in a way that allows it to be extended (within the functional domain that it addresses) without the need to make internal (code or logic-level) modifications to the component itself.

► *The Liskov Substitution Principle (LSP).*

- “*Subclasses should be substitutable for their base classes*”.

# Liskov Substitution Principle

*“Derived classes should extend without replacing the functionality of old classes”*



**Backward Compatibility!!**

- ▶ *Dependency Inversion Principle (DIP).*
- ▶ *“High-level modules should not depend on low-level modules, Both should depend on abstractions. Abstractions should not depend on details. Details should depend on abstraction”*

```
# High-level module
class PaymentProcessor:
    def __init__(self, payment_gateway):
        self.payment_gateway = payment_gateway

    def process_payment(self, amount):
        self.payment_gateway.pay(amount)

# Abstraction
class PaymentGateway:
    def pay(self, amount):
        pass
```

```
# Low-level modules implementing PaymentGateway
class CreditCardPaymentGateway(PaymentGateway):
    def pay(self, amount):
        print(f"Paid ${amount} via Credit Card")

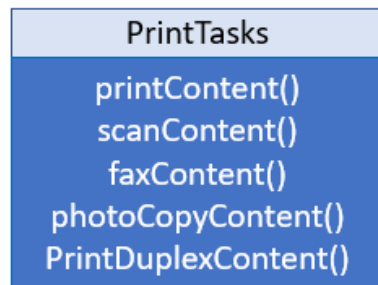
class PayPalPaymentGateway(PaymentGateway):
    def pay(self, amount):
        print(f"Paid ${amount} via PayPal")
```

- ▶ In this example, `PaymentProcessor` depends on the abstraction `PaymentGateway`, and the payment gateway implementations (`CreditCardPaymentGateway` and `PayPalPaymentGateway`) depend on the same abstraction.
- ▶ This adheres to the Dependency Inversion Principle, making it easier to switch or extend payment gateway implementations without affecting the `PaymentProcessor` class.

► *The Interface Segregation Principle (ISP).*

- *“Many client-specific interfaces are better than one general purpose interface”.*
- Only those operations that are relevant to a particular category of clients should be specified in the interface for that client.
- If multiple clients require the same operations, it should be specified in each of specialized interfaces.





Fat Interface, doing too many things



► *The Release Reuse Equivalency Principle (REP).*

- *“The granule of reuse is the granule of release”*
- When classes or components are designed for reuse, an implicit contract is established between the developer and the people who will use it.
- While the developer is committed to supporting older versions, they also encourage users to gradually transition to the latest version of the software entity.

► *The Common Closure Principle (CCP).*

- *“Classes that change together belong together.”*
- Classes should be packaged cohesively.
- That is, when classes are packaged as part of a design, they should address the **same functional or behavioural area.**

## ► *The Common Reuse Principle (CRP).*

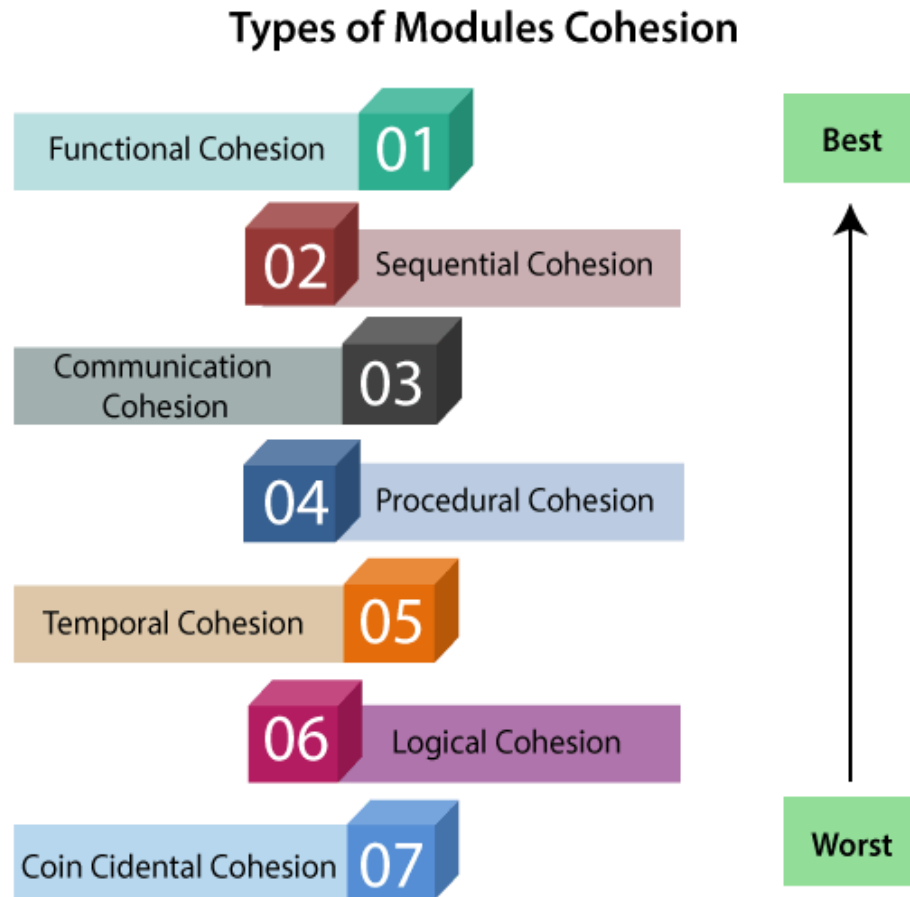
- *“Classes that aren’t reused together should not be grouped together”*
- When one or more classes with a package changes, the all other classes of packages that rely on the package that has been changed.
- For this reason, only classes that are reused together should be included within a package.

# Component-Level Design Guidelines

- ▶ These guidelines apply to components, their interfaces, dependencies and inheritance
- ▶ **Components:** Naming conventions should be established for components that are specified as part of the architectural model.
- ▶ For example, `<<infrastructure>>` might be used to identify an infrastructure component, `<<database>>` could be used to identify a database that services one or more design classes or the entire system; `<<table>>` can be used to identify a table within a database.

- ▶ **Interfaces:** Interfaces provide important information about communication and collaboration.
- ▶ **Dependencies and Inheritance:** For improved readability, it is a good idea to model dependencies from left to right and inheritance from bottom (derived classes) to top (base classes).
- ▶ **Cohesion:** Encapsulates only attributes and operations that are closely related to one another.

# Types of cohesion



- ▶ **Co-incidental cohesion** : Here the elements will be unrelated as the elements **do not have any conceptual relationship**
- ▶ **Logical cohesion** - When logically categorized elements but not **functionally**, are put together into a module, it is called logical cohesion. The operations are related but the functions are significantly different



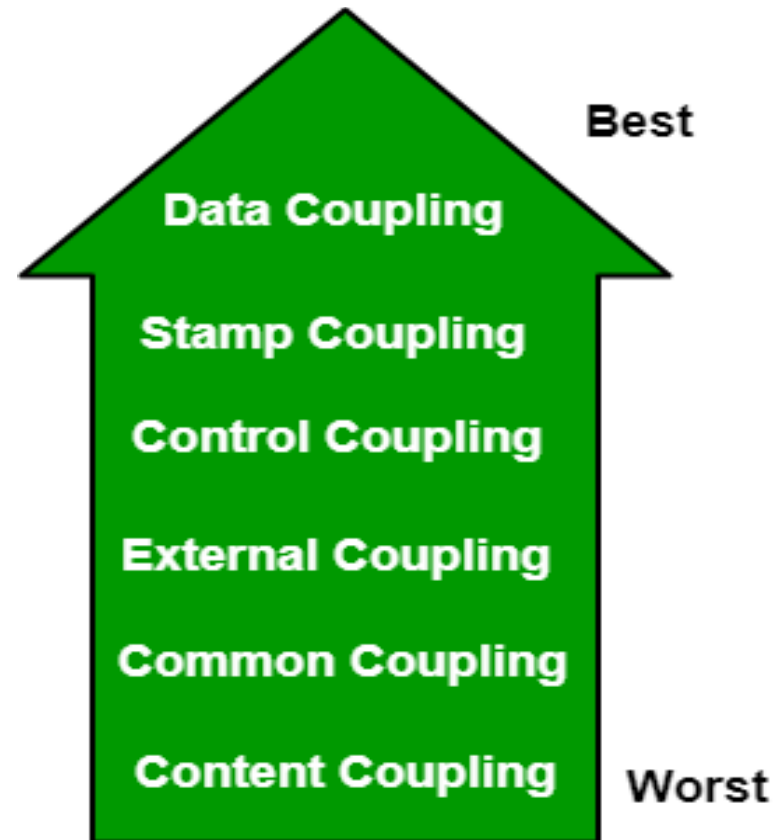
- ▶ **Temporal Cohesion** -When elements of module are organized such that they are **processed at a similar point in time**, it is called temporal cohesion.
- ▶ **Procedural cohesion** -When elements of module are grouped together, which are **executed sequentially in order to perform a task**, it is called procedural cohesion. **Order of execution** .


- ▶ **Communicational cohesion** : Operate on **same input data** or **contribute towards the same output data**
- ▶ **Sequential cohesion** -When elements of module are grouped because **the output of one element serves as input to another** and so on
- ▶ **Functional cohesion** -Elements of module in functional cohesion are grouped because they all contribute to a **single well-defined function**

# Coupling

- ▶ Coupling is a measure that defines the level of **inter-dependability among modules of a program**. It tells at what level the modules interfere and interact with each other.
- ▶ The lower the coupling, the better the program.

# Types of Coupling



- 
- ▶ **Content coupling** -When a module can directly access or modify or refer to the content of another module, it is called content level coupling.
  - ▶ **Common coupling**-When multiple modules have read and write access to some global data, it is called common or global coupling.

- ▶ **External coupling** : Modules depends on other modules external to the software.
- ▶ **Control coupling**-Two modules are called control-coupled if one of them decides the function of the other module or changes its flow of execution.

- ▶ **Stamp coupling**-When multiple modules share common data structure and work on different part of it, it is called stamp coupling.
- ▶ **Data coupling**-Data coupling is when two modules interact with each other by means of passing data (parameters)

# CONDUCTING COMPONENT LEVEL DESIGN

- ▶ The following steps represent a typical task set for component-level design, when it is applied for an object-oriented system
- ▶ *Step 1. Identify all design classes that correspond to the problem domain.* Using the requirements and architectural model, each analysis class and architectural component is elaborated
- ▶ *Step 2. Identify all design classes that correspond to the infrastructure domain.* Classes and components in this category include GUI components (often available as reusable components), operating system components, and object and data management components.



▶ *Step 3. Elaborate all design classes that are not acquired as reusable components.* all interfaces, attributes, and operations necessary to implement the class be described in detail.

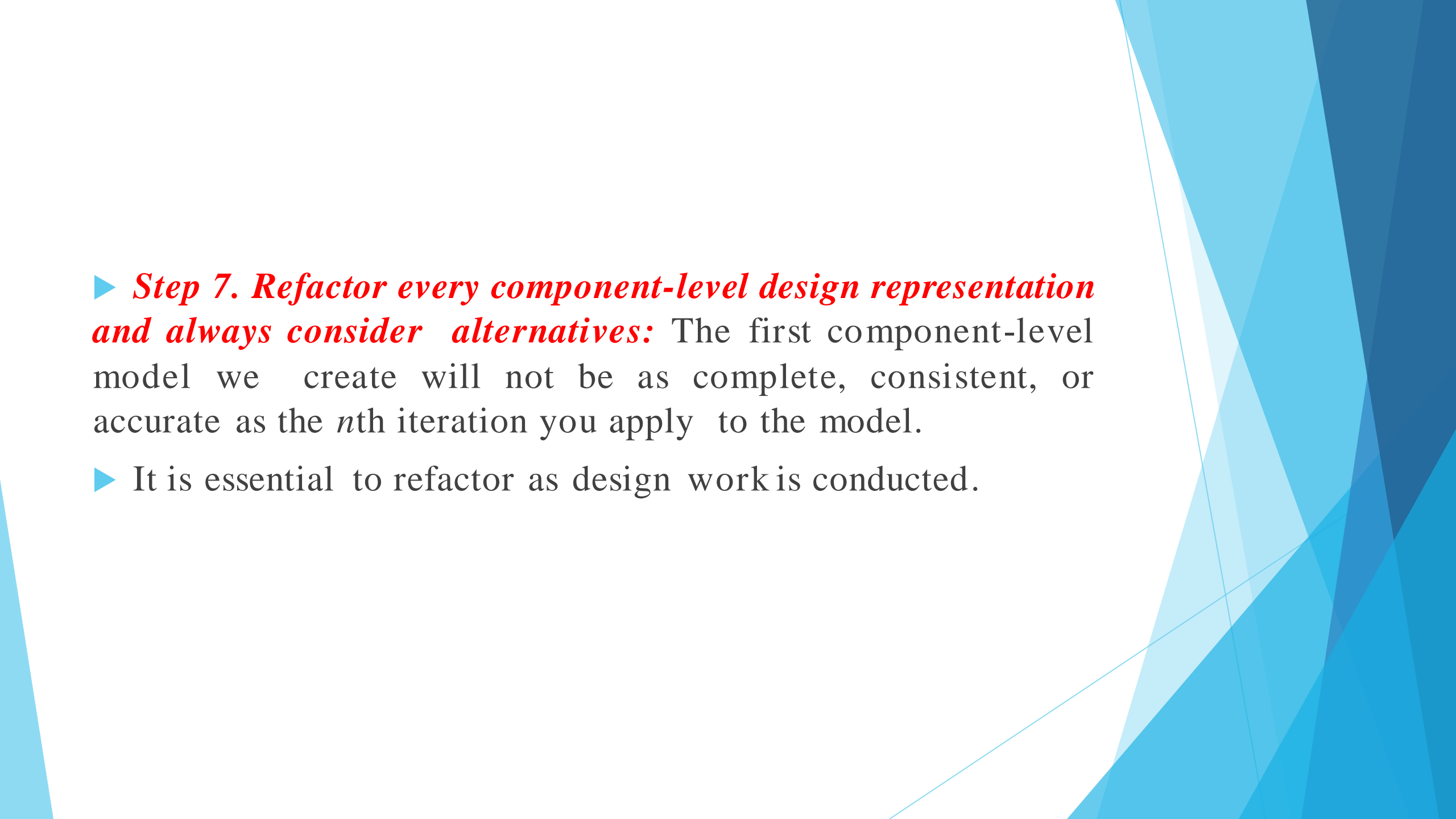
▶ *Step 3a. Specify message details when classes or components collaborate.* show how classes collaborate with one another ie, messages that are passed between objects within a system.

- ▶ *Step 3b. Identify appropriate interfaces for each*
- ▶ *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.* :using a programming language-based pseudo code or with a UML activity diagram.

▶ *Step 4. Describe persistent data sources (databases and files) and identify the classes required to manage them.*

▶ *Step5. Develop and elaborate behavioural 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.

- 
- The background of the slide features abstract, overlapping geometric shapes in various shades of blue, ranging from light sky blue to deep navy blue, creating a modern, layered effect on the right side of the frame.
- ▶ ***Step 7. Refactor every component-level design representation and always consider alternatives:*** The first component-level model we create will not be as complete, consistent, or accurate as the  $n$ th iteration you apply to the model.
  - ▶ It is essential to refactor as design work is conducted.

# COMPONENT LEVEL DESIGN FOR WEB APPLICATIONS

► 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 the end user with some required capability. Therefore, component-level design for WebApps often incorporates elements of content design and functional design.

## Content Design at the Component Level

► Content design at the component level focuses on content objects and the manner in which they may be packaged for presentation to a WebApp end user. The formality of content design at the component level should be tuned to the characteristics of the WebApp to be built. In many cases, content objects need not be organized as components and can be manipulated individually. However, as the size and complexity (of the WebApp, content objects, and their interrelation-ships) grows, it may be necessary to organize content in a way that allows easier reference and design manipulation. In addition, if content is highly dynamic (e.g., the content for an online auction site), it becomes important to establish a clear structural model that incorporates content components.

# Design Document Template

## Contents

1. Overview .....	
1.1 Scope .....	
1.2 Purpose .....	
1.3 Intended audience .....	
1.4 Conformance .....	
2. Definitions .....	
3. Conceptual model for software design descriptions .....	
3.1 Software design in context .....	
3.2 Software design descriptions within the life cycle .....	
4. Design description information content .....	
4.1 Introduction .....	
4.2 SDD identification .....	
4.3 Design stakeholders and their concerns .....	
4.4 Design views .....	
4.5 Design viewpoints .....	
4.6 Design elements .....	
4.7 Design overlays .....	
4.8 Design rationale .....	
4.9 Design languages .....	
5. Design viewpoints .....	
5.1 Introduction .....	
5.2 Context viewpoint .....	
5.3 Composition viewpoint .....	
5.4 Logical viewpoint .....	
5.5 Dependency viewpoint .....	
5.6 Information viewpoint .....	
5.7 Patterns use viewpoint .....	
5.8 Interface viewpoint .....	
5.9 Structure viewpoint .....	
5.10 Interaction viewpoint .....	
5.11 State dynamics viewpoint .....	
5.12 Algorithm viewpoint .....	
5.13 Resource viewpoint .....	
Annex A (informative) Bibliography .....	
Annex B (informative) Conforming design language description .....	

Case study:

