Requirement Engineering

Definition

- It is the process that defines, identifies, manages, and develops requirements in a software engineering design process.

- The process of defining, documentation, and maintenance of requirements in the design process of engineering is called requirements engineering.

Why is it needed ?

- It provides an apt mechanism to understand the customer’s desires, analysis of needs of the customer, feasibility assessment, negotiations for a reasonable solution, clarity in the specification of the solution, specifications validation, and requirements management.

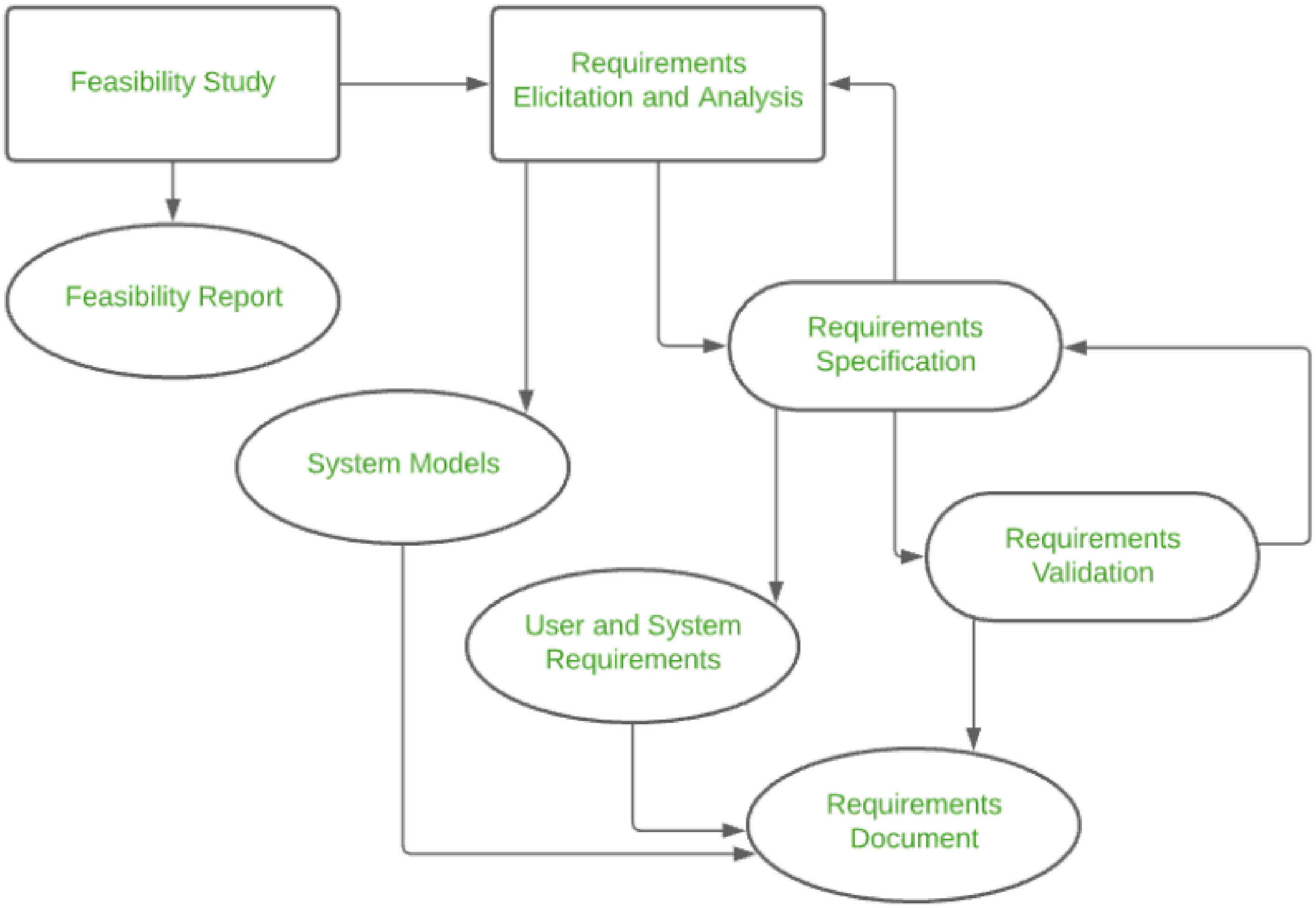
Therefore…

- Requirements Engineering is an application that is disciplined, having the principles, methods, tools, and notations that are proved, which can describe the intended behavior of the proposed system and the constraints associated with it.

The Steps

- Inception / Feasibility Study.
- Elicitation.
- Elaboration.
- Negotiation.
- Specification.
- Validation.
- Requirements Management.

Diagramatic Representation



Inception / Feasibility Study

- The main aim of a feasibility study is to create reasons for the development of the software that the users accept, that is flexible enough and open to changes, and abide by the standards chosen for software development and maintenance.

Inception / Feasibility Study

- **Technical Feasibility:** The current technologies are evaluated using technical feasibility, and they are necessary to achieve the requirements of the customer within the given time and budget.
- **Operational Feasibility:** The assessment of the range for software in which the required software performs a series of levels to solve the problems in business and the customer's requirements.
- **Economic Feasibility:** If the required software can generate profits in the area of finance is decided by economic feasibility.

Elicitation of Requirements and Analysis

- This is the second phase of the requirements analysis process.
- This phase focuses on gathering the requirements from the stakeholders.
- One should be careful in this phase, as the requirements are what establishes the key purpose of a project.
- Understanding the kind of requirements needed from the customer is very crucial for a developer. In this process, mistakes can happen in regard to, not implementing the right requirements or forgetting a part.

Challenges…

- **Problem of Scope:** The requirements given are of unnecessary detail, ill-defined, or not possible to implement.
- **Problem of Understanding:** Not having a clear-cut understanding between the developer and customer when putting out the requirements needed. Sometimes the customer might not know what they want or the developer might misunderstand one requirement for another.
- **Problem of Volatility:** Requirements changing over time can cause difficulty in leading a project. It can lead to loss and wastage of resources and time.

Elaboration

- This is the third phase of the requirements analysis process.
- This phase is the result of the inception and elicitation phase.
- In the elaboration process, it takes the requirements that have been stated and gathered in the first two phases and refines them.
- The main task in this phase is to indulge in modeling activities and develop a prototype that elaborates on the features and constraints using the necessary tools and functions.

Negotiation

- This is the fourth phase of the requirements analysis process. This phase emphasizes discussion and exchanging conversation on what is needed and what is to be eliminated.
- In the negotiation phase, negotiation is between the developer and the customer and they dwell on how to go about the project with limited business resources. Customers are asked to prioritize the requirements and make guesstimates on the conflicts that may arise along with it.
- Risks of all the requirements are taken into consideration and negotiated in a way where the customer and developer are both satisfied with reference to the further implementation.

Points for discussion

- Availability of Resources.
- Delivery Time.
- Scope of requirements.
- Project Cost.
- Estimations on development.

Specification of Software Requirements

- A document consisting of requirements that are collected from various sources like the requirements from customers expressed in an ordinary language and created by the software analyst is called a **specification document** for software requirements.
- The analyst understands the customers’ requirements in ordinary language and converts them into a technical language that the development team can easily understand.
- Several models are used during the process of specification of software requirements like Entity-Relationship diagrams (ER diagrams), data flow diagrams (DFD), data dictionaries, function decomposition diagrams (FDD), etc.

- **Data Flow Diagrams (DFD):** The modeling of requirements can be done using Data Flow Diagrams (DFD). The flow of data within the system can be seen by using data flow diagrams (DFD). The system here can be a company, an organization, a hardware system in a computer, a software system in a computer, a set of procedures or a combination of everything. The data flow diagrams (DFD) are also called bubble charts or data flow graph.
- **Data Dictionaries:** The data defined using a data flow diagram (DFD) is stored as information in the form of repositories called data dictionaries. The customer's data items must be defined by the data dictionaries at the stage of requirements gathering to make sure that the customers and developers use the same methodologies and definitions.
- **Entity Relationship Diagrams:** One of the other tools used for the specification of requirements is entity-relationship diagrams. It is also called as ER diagrams. The detailed representation of logic for the organization is done using entity relationship diagrams (ER diagrams). They make use of three types of constructs: relationships, entities of data, and the attributes associated with them.

Validation of Software Requirements

- After the development of the specification of requirements, the requirements laid down in the document are validated. There may be requirements from the users that are illegal or which cannot be accomplished, or the experts can misunderstand the needs. The requirements must satisfy the following conditions:
  - If the requirements can be implemented practically.
  - If the requirements are correct and they are according to the software functionality.
  - If there are any confusions.
  - If the requirements are full.
  - If the requirements can be described.

Techniques…

- There are several techniques for the validation of requirements. They are:
  - **Inspection of requirements or reviews of requirements**.
- This includes an analysis of the requirements systematically and manually.
  - **Prototyping**
- The requirements of the model are checked by using a model that is executable.
  - **Generation of test case**
- The testability is checked for the requirements by the development of tests.
  - **Automated consistency analysis**
- The consistency of the descriptions of requirements is checked.

Management of Software Requirements

- The process of managing the requirements that keep changing during the process of requirements engineering and development of the system is called management of software requirement.
- During the process of software management, there are new requirements with the change in the needs of business, and there is the development of a better understanding of the system.
- During the process of development, the requirements priority changes from different views.
- During the process of development, the technical and business environment of the system changes.

Advantages of Requirement Engineering

- **Lesser chances of process overhead**. The required for the analysis and documentation of the requirements will be comparatively less because requirements capture using requirements engineering are quicker and more precise.

- The **requirements that are critical can be identified and implemented in the initial stage using requirements engineering**. As there is concurrency between requirements engineering activities, communication is more efficient between the stakeholders and the software engineers.

- There is a **quick response to the changes in requirements**. The identification of requirements and their documentation undergo a series of iteration before they are finalized, so the response to the changes in the requirements can be quicker and can be done at a relatively lower cost.

Thank you

Software Requirements – User Perspective

- **Functional Requirements**. Requirements, which are related to functional aspect of software fall into this category.
  - Search option given to user to search from various invoices.
  - User should be able to mail any report to management.
  - Users can be divided into groups and groups can be given separate rights.
  - Should comply business rules and administrative functions.
  - Software is developed keeping downward compatibility intact.

- **Non-Functional Requirements.** Requirements, which are not related to functional aspect of software, fall into this category. They are implicit or expected characteristics of software, which users make assumption of.
  - Security
  - Logging
  - Storage
  - Configuration
  - Performance
  - Cost
  - Interoperability
  - Flexibility
  - Disaster recovery
  - Accessibility

Software Requirements – User Perspective

- Requirements are categorized logically as…
  - **Must Have** : Software cannot be said operational without them.
  - **Should have** : Enhancing the functionality of software.
  - **Could have** : Software can still properly function with these requirements.
  - **Wish list** : These requirements do not map to any objectives of software.

Software Requirements – User Perspective

- Requirements are categorized logically as…
  - **Must Have** : Software cannot be said operational without them.
  - **Should have** : Enhancing the functionality of software.
  - **Could have** : Software can still properly function with these requirements.
  - **Wish list** : These requirements do not map to any objectives of software.

While developing software, 'Must have' must be implemented, 'Should have' is a matter of debate with stakeholders and negation, whereas 'could have' and 'wish list' can be kept for software updates.

Software Requirements – User Perspective

- **User Interface requirements**. UI is an important part of any software or hardware or hybrid system. A software is widely accepted if it is -
  - easy to operate
  - quick in response
  - effectively handling operational errors
  - providing simple yet consistent user interface

  UI is the only way for users to perceive the system.
  - Content presentation
  - Easy Navigation
  - Simple interface
  - Responsive
  - Consistent UI elements
  - Feedback mechanism
  - Default settings
  - Purposeful layout
  - Strategical use of color and texture.
  - Provide help information
  - User centric approach
  - Group based view settings.

Software System Analyst

- System analyst in an IT organization is a person, who analyzes the requirement of proposed system and ensures that requirements are conceived and documented properly & correctly. Role of an analyst starts during Software Analysis Phase of SDLC. It is the responsibility of analyst to make sure that the developed software meets the requirements of the client.
- System Analysts have the following responsibilities:
  - Analyzing and understanding requirements of intended software
  - Understanding how the project will contribute in the organization objectives
  - Identify sources of requirement
  - Validation of requirement
  - Develop and implement requirement management plan
  - Documentation of business, technical, process and product requirements
  - Coordination with clients to prioritize requirements and remove and ambiguity
  - Finalizing acceptance criteria with client and other stakeholders

Software Measurement and Metrics

- Software measurement is a titrate impute of a characteristic of a software product or the software process. It is an authority within software engineering. The software measurement process is defined and governed by ISO Standard.

- **Software Measurement Principles:**
  - **Formulation:** The derivation of software measures and metrics appropriate for the representation of the software that is being considered.
  - **Collection:** The mechanism used to accumulate data required to derive the formulated metrics.
  - **Analysis:** The computation of metrics and the application of mathematical tools.
  - **Interpretation:** The evaluation of metrics resulting in insight into the quality of the representation.
  - **Feedback:** Recommendation derived from the interpretation of product metrics transmitted to the software team.

Software Measurement and Metrics

- **Need for Software Measurement:**
  - Create the quality of the current product or process.
  - Anticipate future qualities of the product or process.
  - Enhance the quality of a product or process.
  - Regulate the state of the project in relation to budget and schedule.
  - Enable data-driven decision-making in project planning and control.
  - Identify bottlenecks and areas for improvement to drive process improvement activities.
  - Ensure that industry standards and regulations are followed.
  - Give software products and processes a quantitative basis for evaluation.
  - Enable the ongoing improvement of software development practices.

Software Measurement and Metrics

- **Advantages of Software Metrics :**
  - Reduction in cost or budget.
  - It helps to identify the particular area for improvising.
  - It helps to increase the product quality.
  - Managing the workloads and teams.
  - Reduction in overall time to produce the product,.
  - It helps to determine the complexity of the code and to test the code with resources.
  - It helps in providing effective planning, controlling and managing of the entire product.

Software Measurement and Metrics

- **Advantages of Software Metrics :**
  - Reduction in cost or budget.
  - It helps to identify the particular area for improvising.
  - It helps to increase the product quality.
  - Managing the workloads and teams.
  - Reduction in overall time to produce the product,.
  - It helps to determine the complexity of the code and to test the code with resources.
  - It helps in providing effective planning, controlling and managing of the entire product.

Software Metrics

Size Metrics - LOC (Lines of Code), mostly calculated in thousands of delivered source code lines, denoted as KLOC. Function Point Count is measure of the functionality provided by the software. Function Point count defines the size of functional aspect of software.

Complexity Metrics - McCabe's Cyclomatic complexity quantifies the upper bound of the number of independent paths in a program, which is perceived as complexity of the program or its modules. It is represented in terms of graph theory concepts by using control flow graph.

Quality Metrics - Defects, their types and causes, consequence, intensity of severity and their implications define the quality of product. The number of defects found in development process and number of defects reported by the client after the product is installed or delivered at client-end, define quality of product.

Process Metrics - In various phases of SDLC, the methods and tools used, the company standards and the performance of development are software process metrics.

Resource Metrics - Effort, time and various resources used, represents metrics for resource measurement.

Cyclomatic Complexity – An example

- The Cyclomatic complexity of a code section is the quantitative measure of the number of linearly independent paths in it. It is a software metric used to indicate the complexity of a program. It is computed using the Control Flow Graph of the program. The nodes in the graph indicate the smallest group of commands of a program, and a directed edge in it connects the two nodes i.e. if the second command might immediately follow the first command.
- For example, if the source code contains no control flow statement then its cyclomatic complexity will be 1, and the source code contains a single path in it. Similarly, if the source code contains one **if condition** then cyclomatic complexity will be 2 because there will be two paths one for true and the other for false.
- Mathematically, for a structured program, the directed graph inside the control flow is the edge joining two basic blocks of the program as control may pass from first to second. So, **Cyclomatic complexity M** would be defined as,

$$M = E - N + 2P$$

where

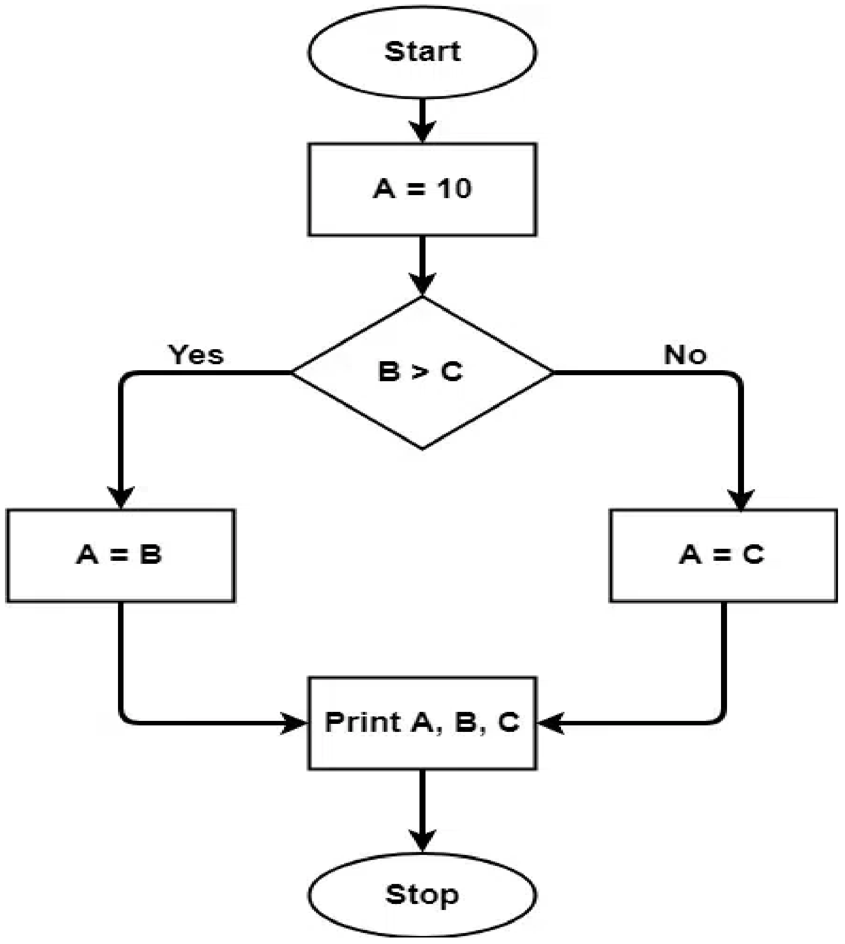
E = the number of edges in the control flow graph
N = the number of nodes in the control flow graph
P = the number of connected components

Cyclomatic Complexity – An example

Let a section of code as such:

```
A = 10
IF B > C THEN
A = B
ELSE
A = C
ENDIF
Print A
Print B
Print C
```

The cyclomatic complexity calculated for the above code will be from the control flow graph. The graph shows seven shapes(nodes), and seven lines(edges), hence cyclomatic complexity is
7-7+2 = 2.



Control Flow Graph of the code

Note - *Since the code snippet depicts only 1 component P = 1*

Conclusion

A complete Software Requirement Specifications must be:

- Clear
- Correct
- Consistent
- Coherent
- Comprehensible
- Modifiable
- Verifiable
- Prioritized
- Unambiguous
- Traceable
- Credible source

Thank you

System Modeling

Topics covered

- Context models
- Interaction models
- Structural models
- Behavioral models
- Model-driven engineering

System modeling

- System modeling is the process of developing abstract models of a system, with each model presenting a different view or perspective of that system.
- System modeling has now come to mean representing a system using some kind of graphical notation, which is now almost always based on notations in the Unified Modeling Language (UML).
- System modeling helps the analyst to understand the functionality of the system and models are used to communicate with customers.

Existing and planned system models

- Models of the existing system are used during requirements engineering. They help clarify what the existing system does and can be used as a basis for discussing its strengths and weaknesses. These then lead to requirements for the new system.
- Models of the new system are used during requirements engineering to help explain the proposed requirements to other system stakeholders. Engineers use these models to discuss design proposals and to document the system for implementation.
- In a model-driven engineering process, it is possible to generate a complete or partial system implementation from the system model.

System perspectives

- An external perspective, where you model the context or environment of the system.
- An interaction perspective, where you model the interactions between a system and its environment, or between the components of a system.
- A structural perspective, where you model the organization of a system or the structure of the data that is processed by the system.
- A behavioral perspective, where you model the dynamic behavior of the system and how it responds to events.

UML diagram types

- Activity diagrams, which show the activities involved in a process or in data processing .
- Use case diagrams, which show the interactions between a system and its environment.
- Sequence diagrams, which show interactions between actors and the system and between system components.
- Class diagrams, which show the object classes in the system and the associations between these classes.
- State diagrams, which show how the system reacts to internal and external events.

Use of graphical models

- As a means of facilitating discussion about an existing or proposed system
  - Incomplete and incorrect models are OK as their role is to support discussion.
- As a way of documenting an existing system
  - Models should be an accurate representation of the system but need not be complete.
- As a detailed system description that can be used to generate a system implementation
  - Models have to be both correct and complete.

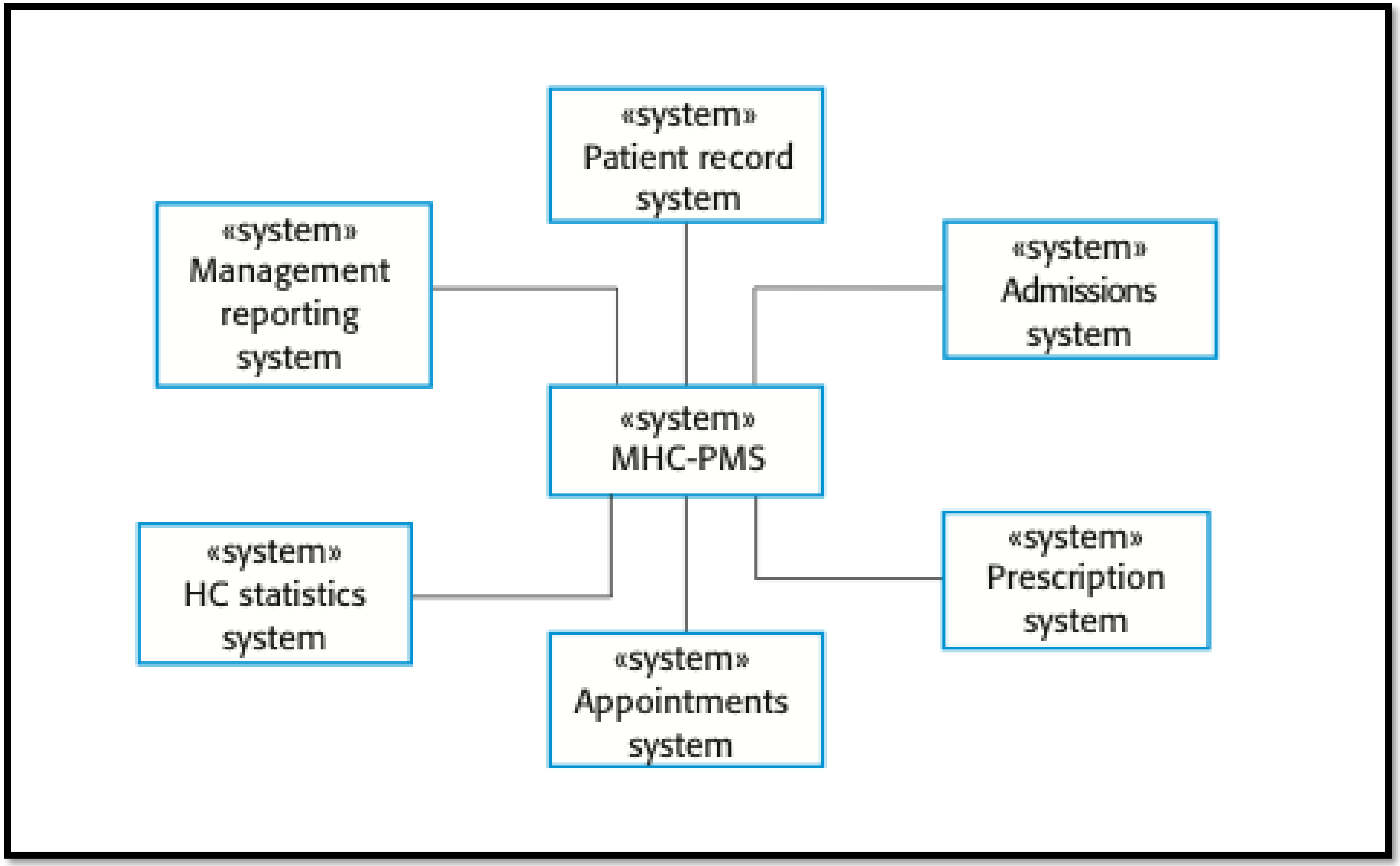
Context models

- Context models are used to illustrate the operational context of a system - they show what lies outside the system boundaries.
- Social and organizational concerns may affect the decision on where to position system boundaries.
- Architectural models show the system and its relationship with other systems.

System boundaries

- System boundaries are established to define what is inside and what is outside the system.
  - They show other systems that are used or depend on the system being developed.
- The position of the system boundary has a profound effect on the system requirements.
- Defining a system boundary is a political judgment
  - There may be pressures to develop system boundaries that increase / decrease the influence or workload of different parts of an organization.

The context of the MHC-PMS

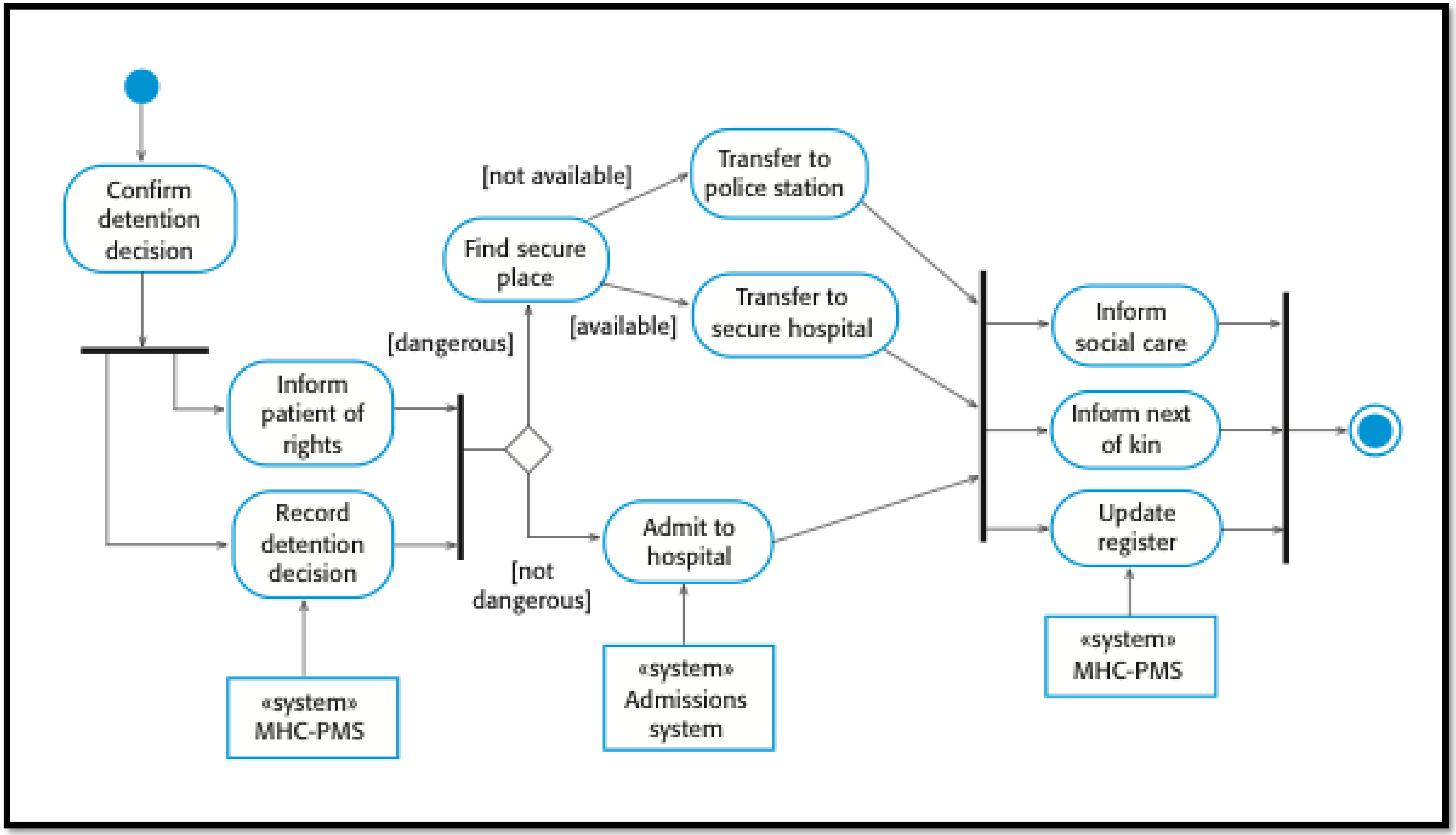

«system» Patient record system
«system» Management reporting system
«system» Admissions system
«system» MHC-PMS
«system» HC statistics system
«system» Prescription system
«system» Appointments system

Patient Management System for Mental Health Care

Process perspective

- Context models simply show the other systems in the environment, not how the system being developed is used in that environment.
- Process models reveal how the system being developed is used in broader business processes.
- UML activity diagrams may be used to define business process models.

Process model of involuntary detention

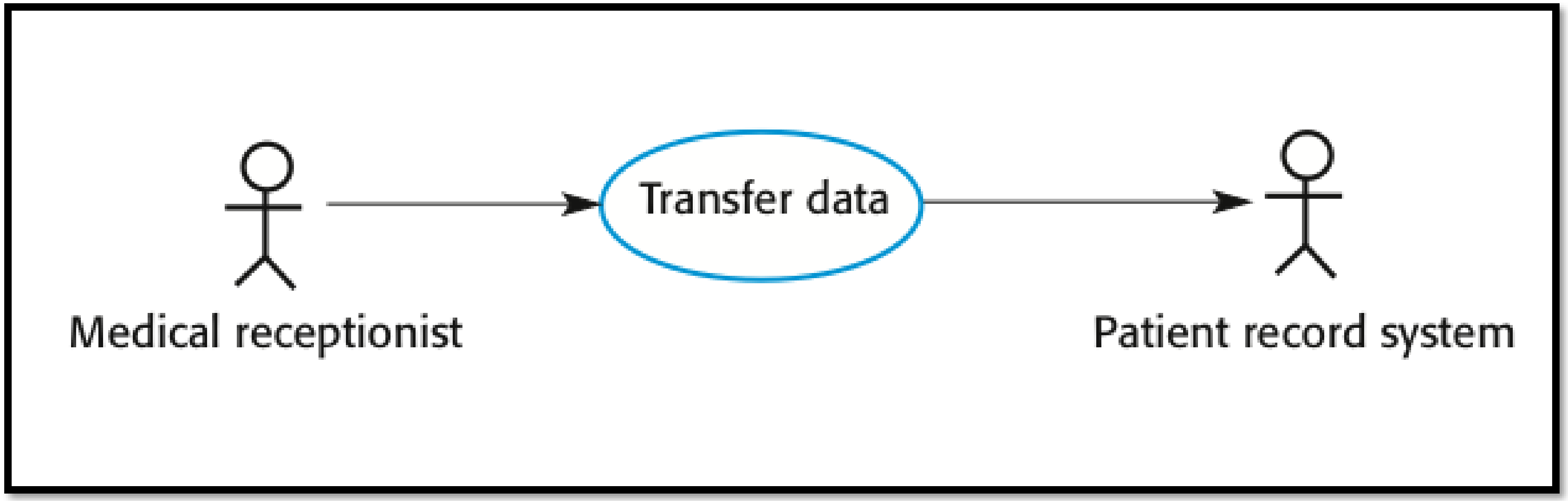


Interaction models

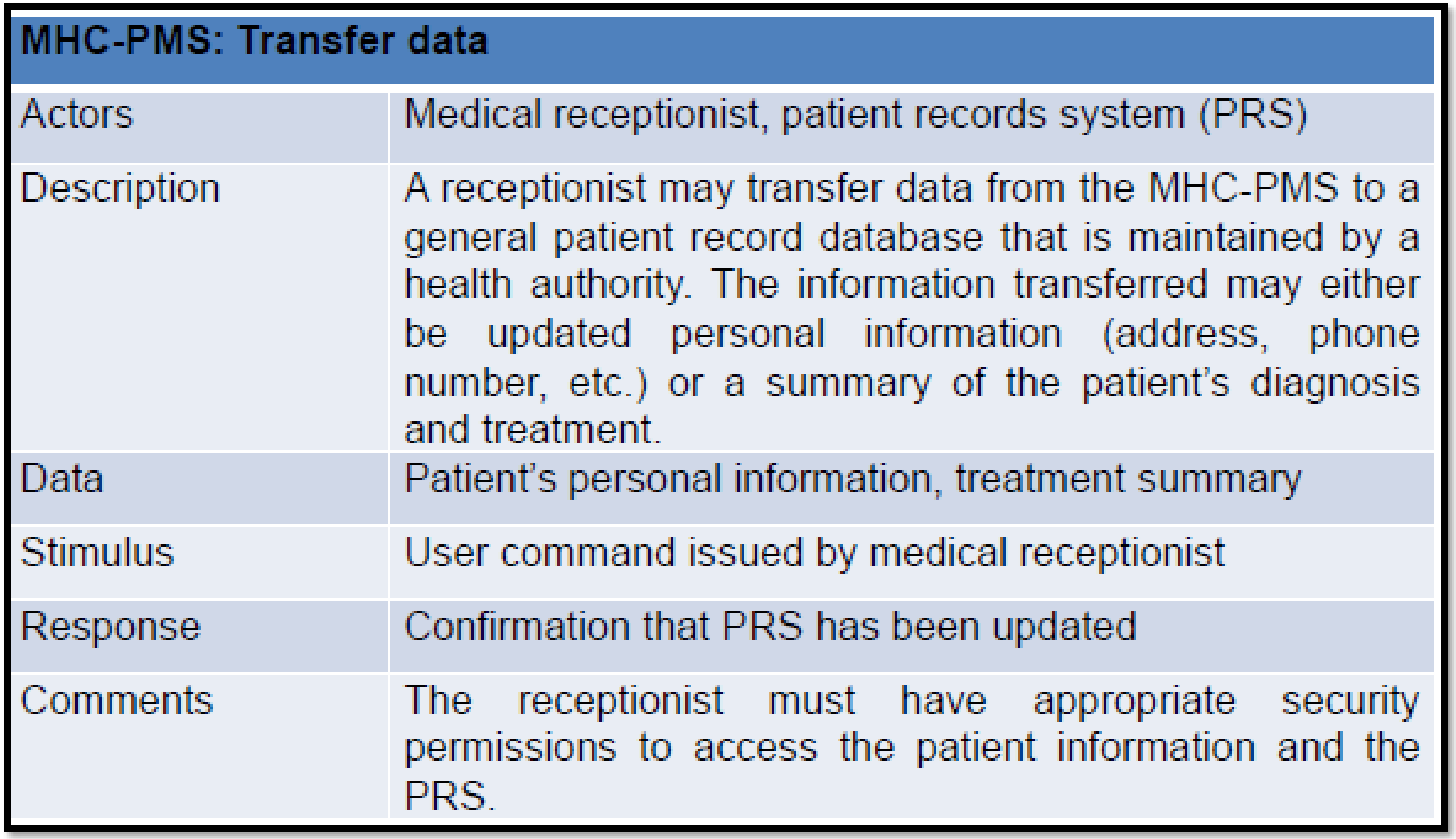
- Modeling user interaction is important as it helps to identify user requirements.
- Modeling system-to-system interaction highlights the communication problems that may arise.
- Modeling component interaction helps us understand if a proposed system structure is likely to deliver the required system performance and dependability.
- Use case diagrams and sequence diagrams may be used for interaction modeling.

Transfer-data use case

- A use case in the MHC-PMS

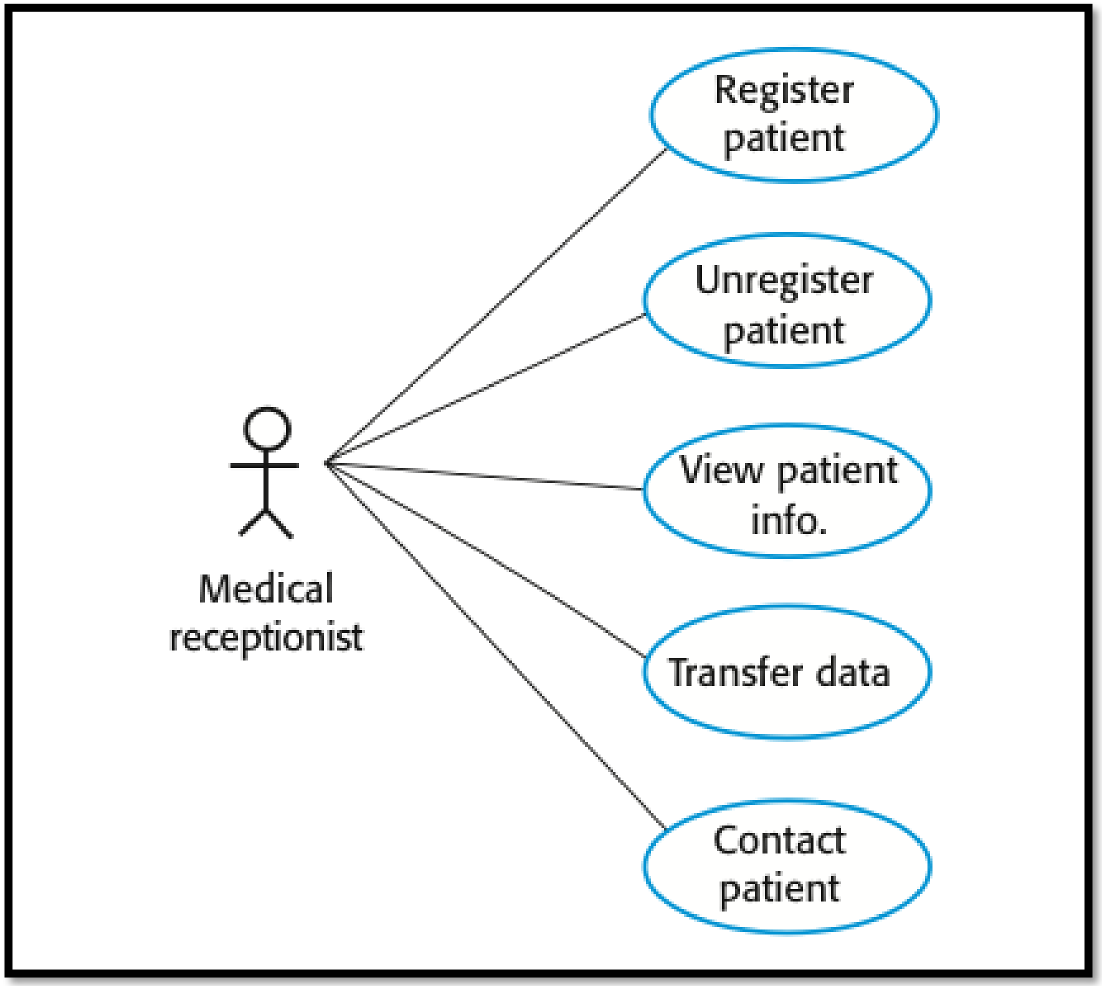


Tabular description of the ‘Transfer data’ use-case



MHC-PMS: Transfer data	
Actors	Medical receptionist, patient records system (PRS)
Description	A receptionist may transfer data from the MHC-PMS to a general patient record database that is maintained by a health authority. The information transferred may either be updated personal information (address, phone number, etc.) or a summary of the patient’s diagnosis and treatment.
Data	Patient’s personal information, treatment summary
Stimulus	User command issued by medical receptionist
Response	Confirmation that PRS has been updated
Comments	The receptionist must have appropriate security permissions to access the patient information and the PRS.

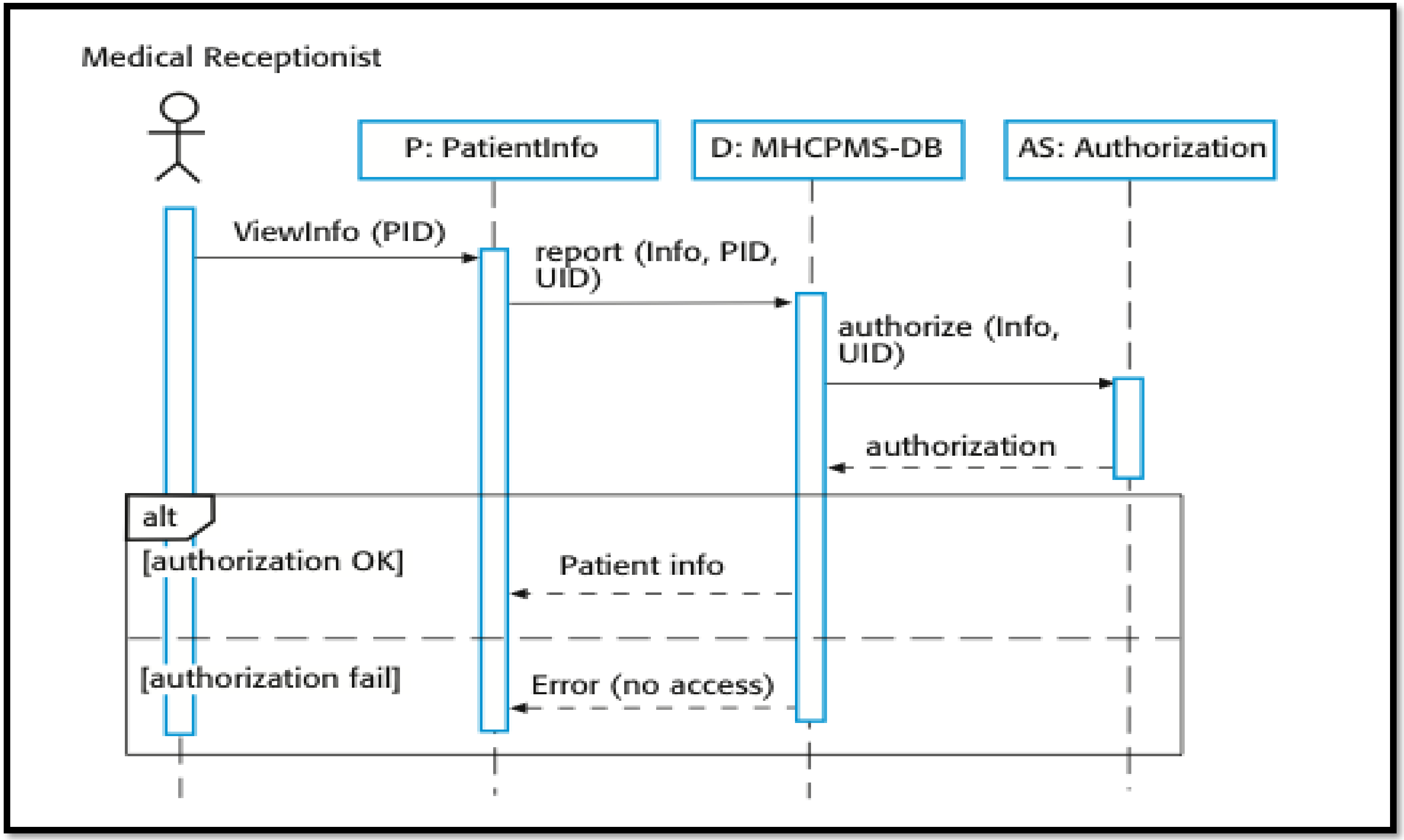
Use cases in the MHC-PMS involving the role 'Medical Receptionist'



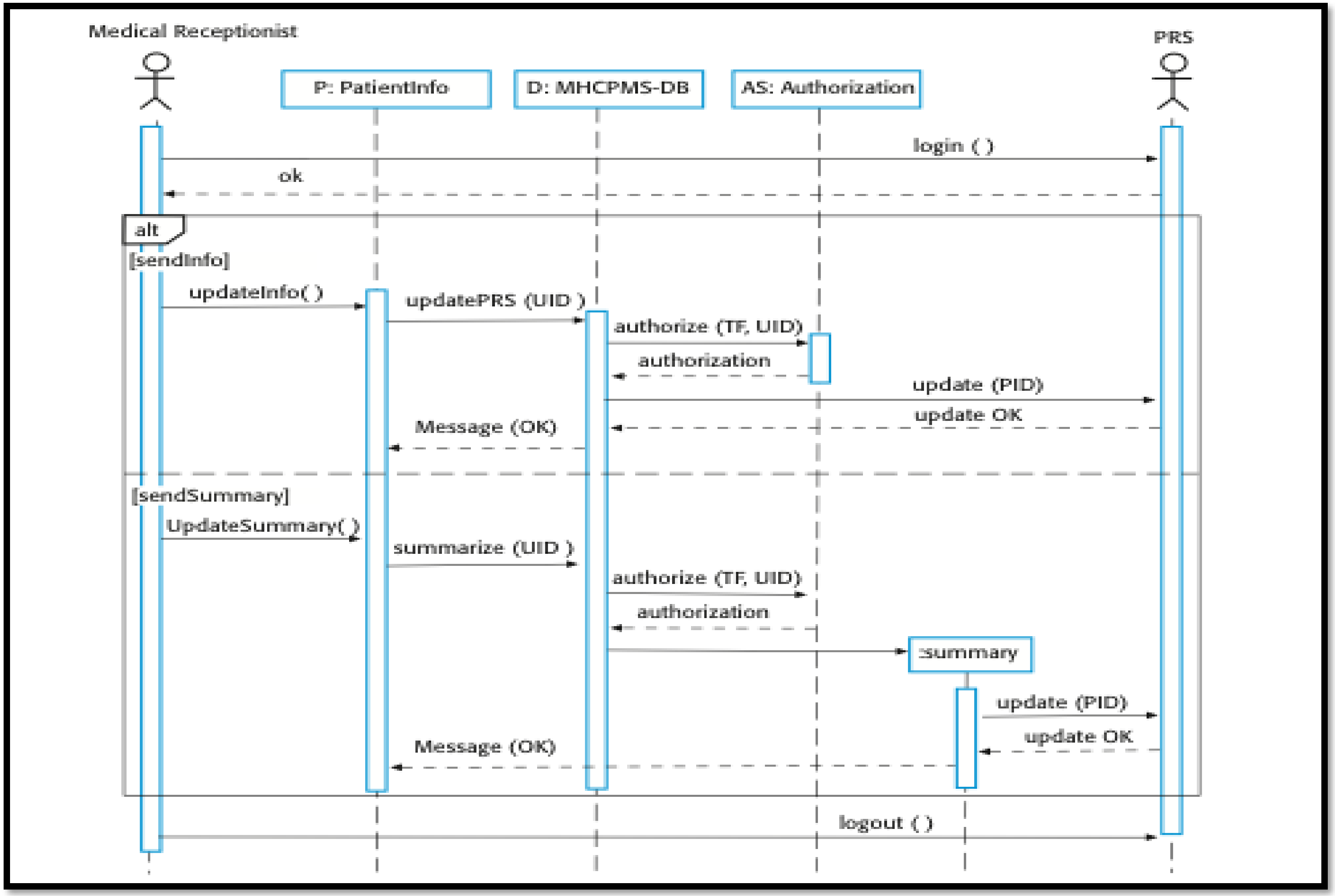
Sequence diagrams

- Sequence diagrams are part of the UML and are used to model the interactions between the actors and the objects within a system.
- A sequence diagram shows the sequence of interactions that take place during a particular use case or use case instance.
- The objects and actors involved are listed along the top of the diagram, with a dotted line drawn vertically from these.
- Interactions between objects are indicated by annotated arrows.

Sequence diagram for View patient information



Sequence diagram for Transfer Data



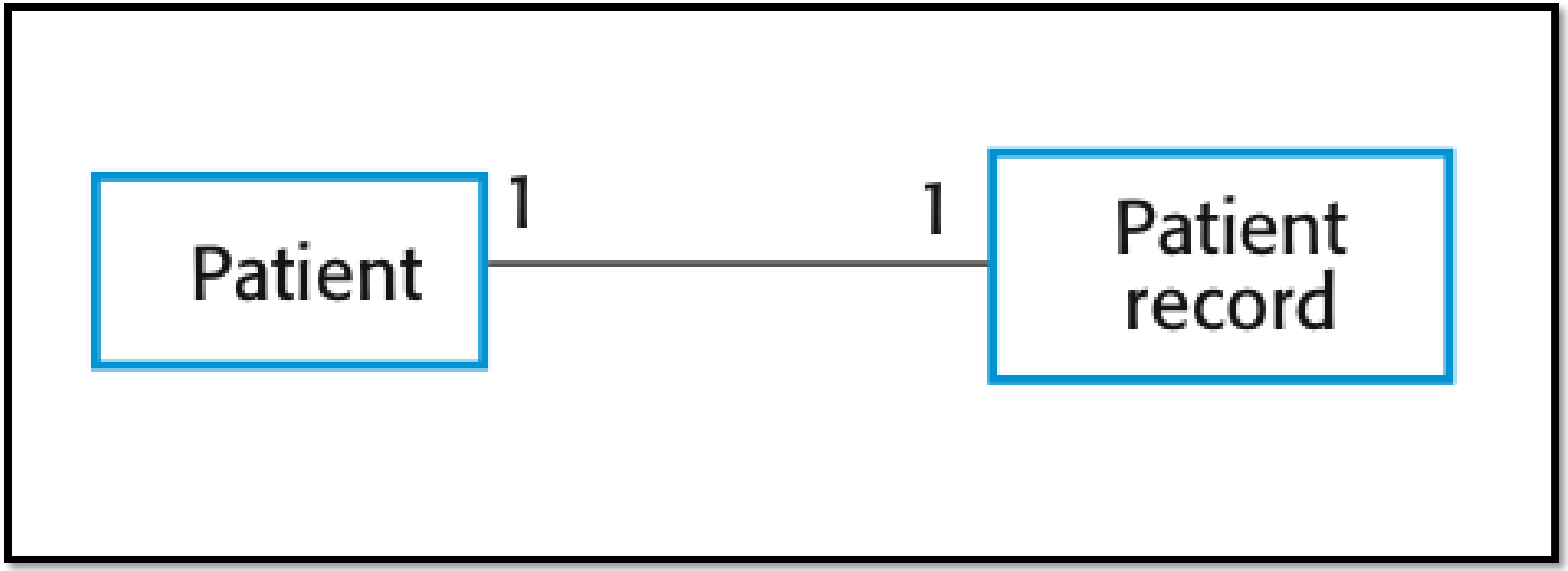
Structural models

- Structural models of software display the organization of a system in terms of the components that make up that system and their relationships.
- Structural models may be static models, which show the structure of the system design, or dynamic models, which show the organization of the system when it is executing.
- You create structural models of a system when you are discussing and designing the system architecture.

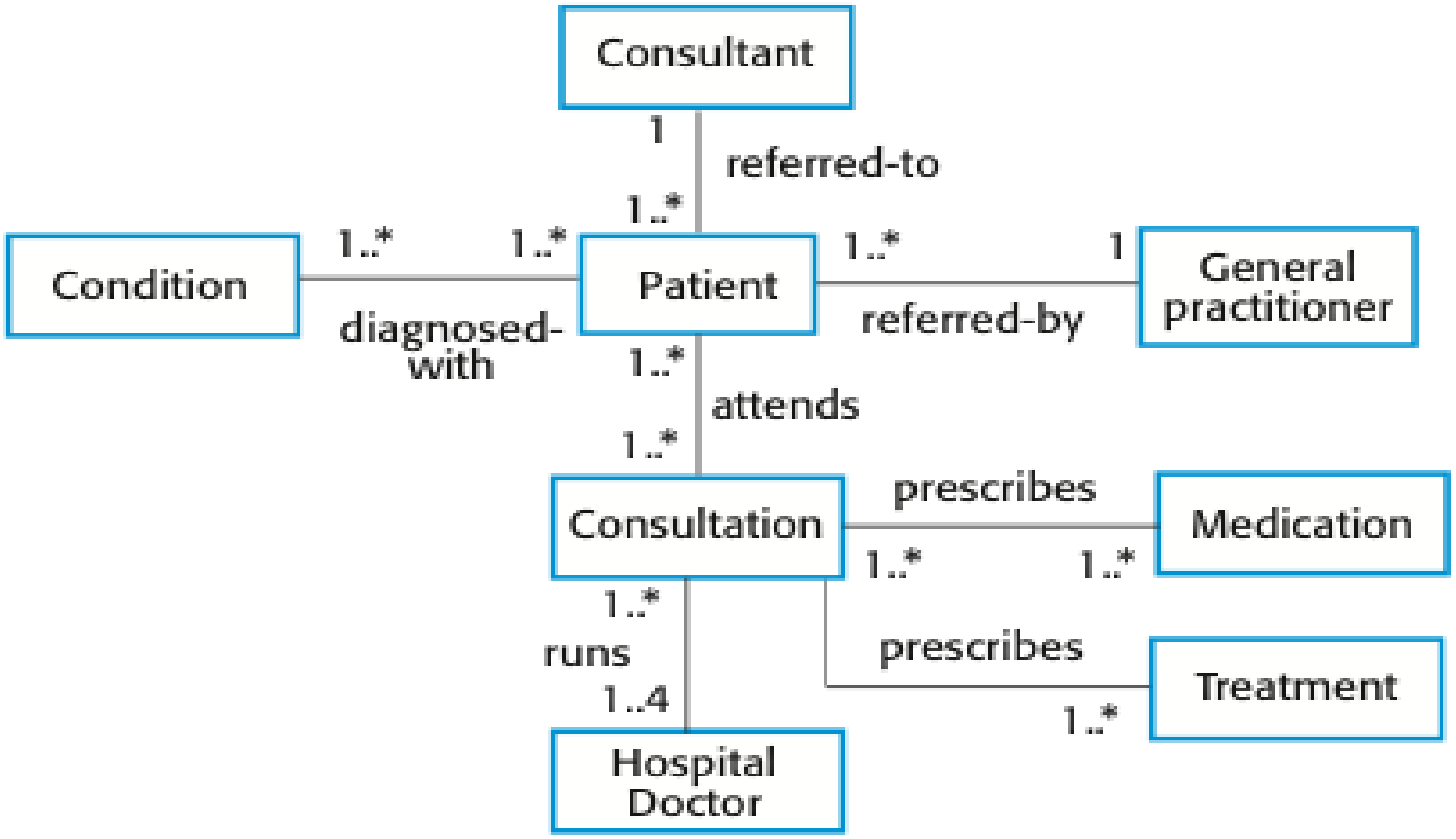
Class diagrams

- Class diagrams are used when developing an object-oriented system model to show the classes in a system and the associations between these classes.
- An object class can be thought of as a general definition of one kind of system object.
- An association is a link between classes that indicates that there is some relationship between these classes.
- When you are developing models during the early stages of the software engineering process, objects represent something in the real world, such as a patient, a prescription, doctor, etc.

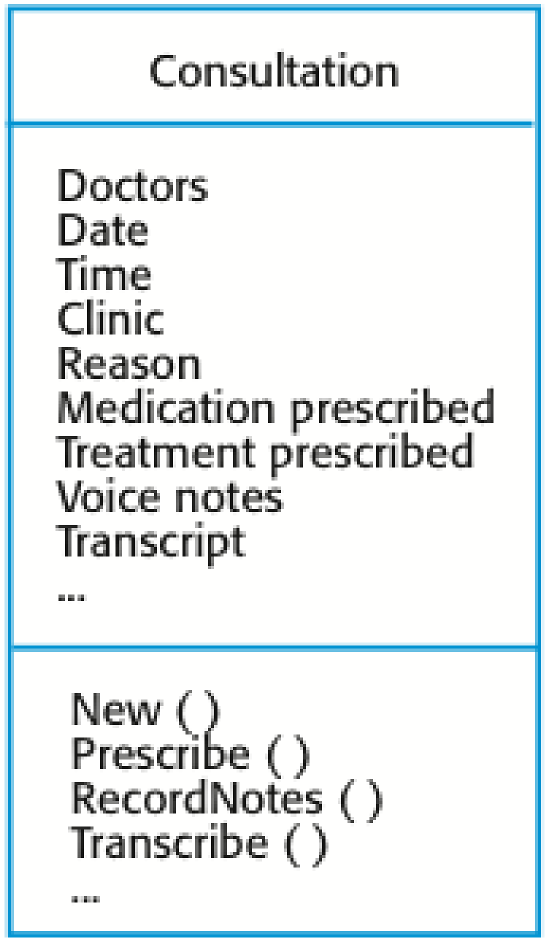
UML classes and association



Classes and associations in the MHC-PMS



The Consultation class



Key points

- A model is an abstract view of a system that ignores system details. Complementary system models can be developed to show the system's context, interactions, structure and behavior.
- Context models show how a system that is being modeled is positioned in an environment with other systems and processes.
- Use case diagrams and sequence diagrams are used to describe the interactions between users and systems in the system being designed. Use cases describe interactions between a system and external actors; sequence diagrams add more information to these by showing interactions between system objects.
- Structural models show the organization and architecture of a system. Class diagrams are used to define the static structure of classes in a system and their associations.

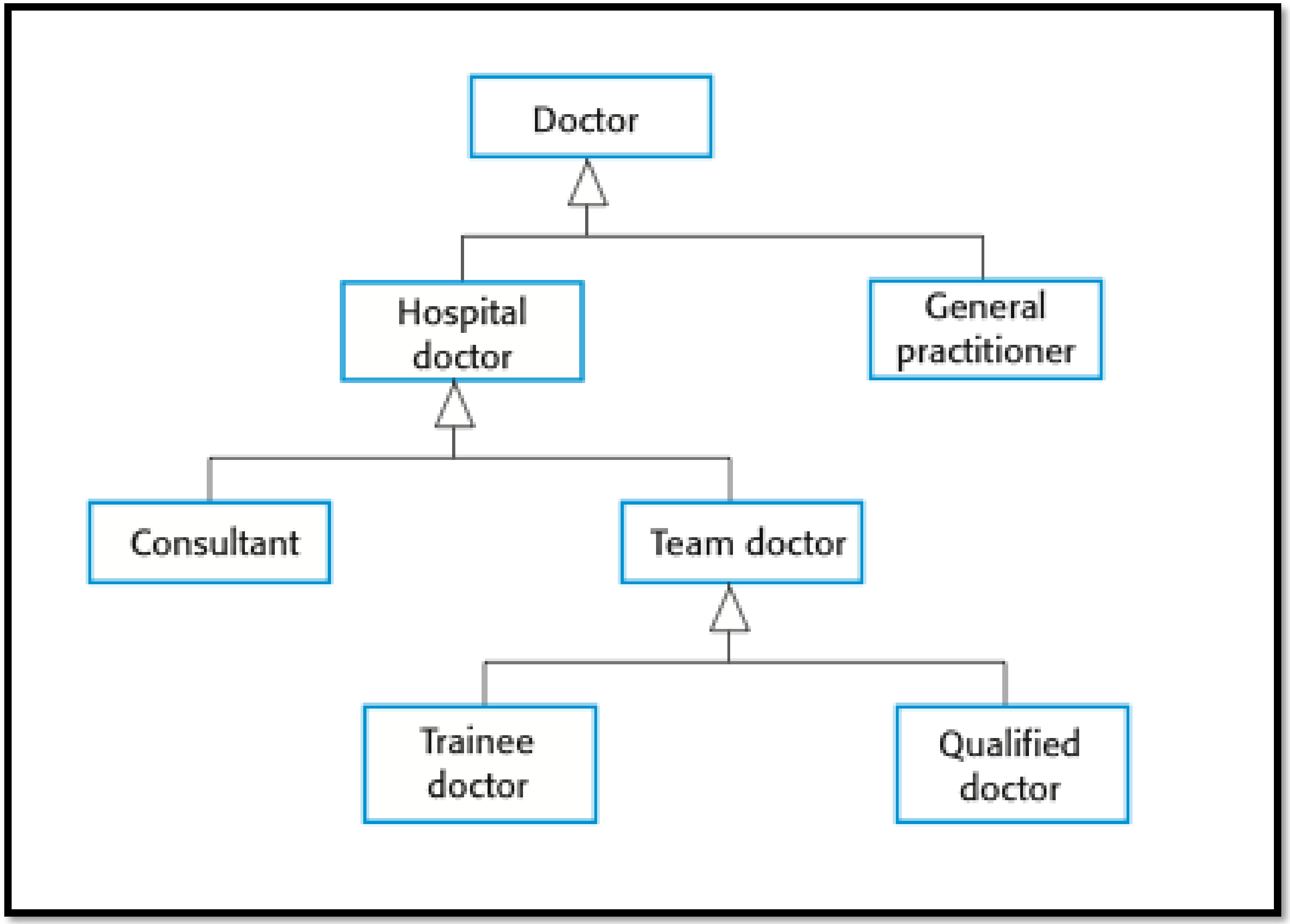
Generalization

- Generalization is an everyday technique that we use to manage complexity.
- Rather than learn the detailed characteristics of every entity that we experience, we place these entities in more general classes (animals, cars, houses, etc.) and learn the characteristics of these classes.
- This allows us to infer that different members of these classes have some common characteristics e.g. squirrels and rats are rodents.

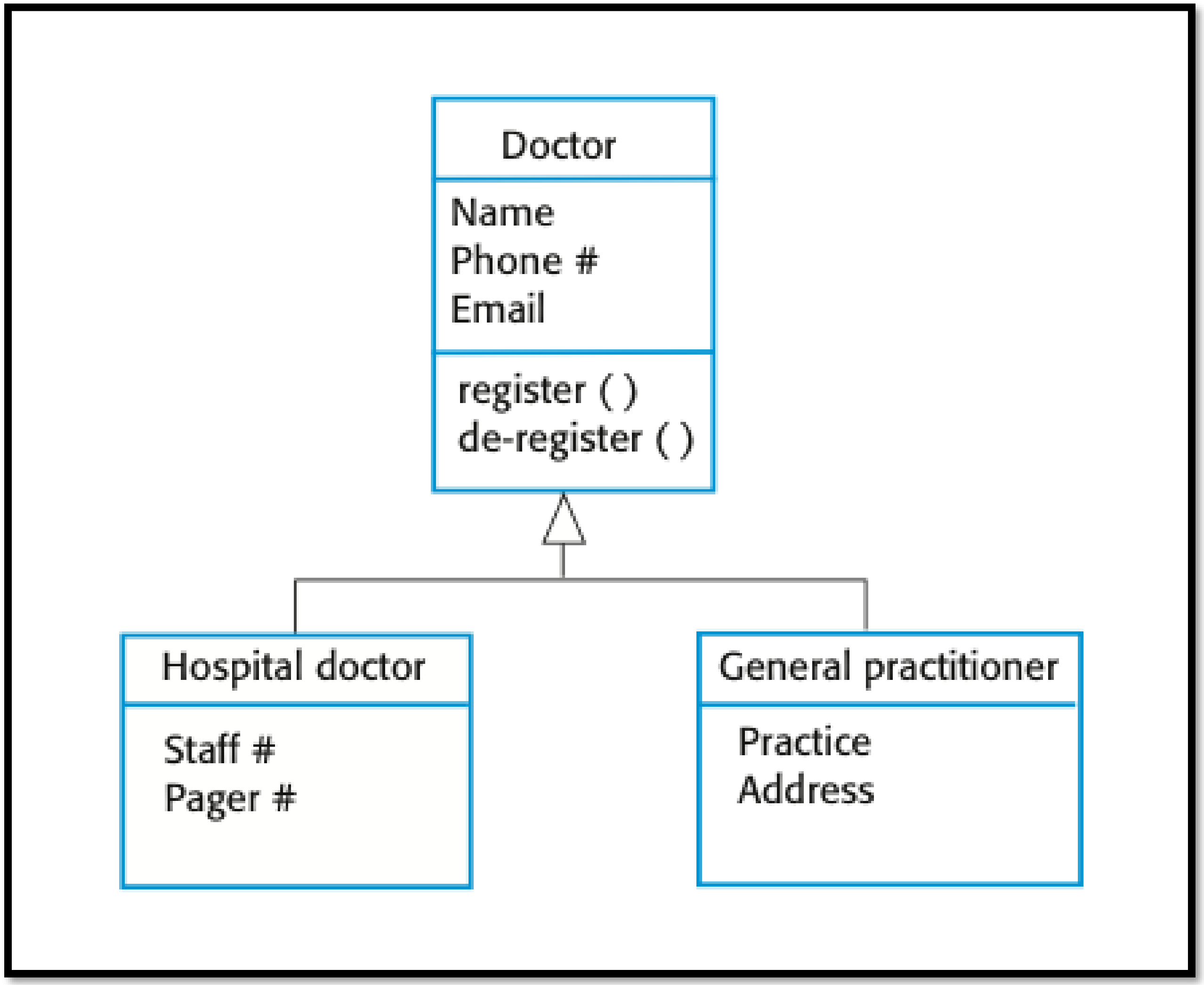
Generalization

- In modeling systems, it is often useful to examine the classes in a system to see if there is scope for generalization. If changes are proposed, then you do not have to look at all classes in the system to see if they are affected by the change.
- In object-oriented languages, such as Java, generalization is implemented using the class inheritance mechanisms built into the language.
- In a generalization, the attributes and operations associated with higher-level classes are also associated with the lower-level classes.
- The lower-level classes are subclasses inherit the attributes and operations from their superclasses. These lower-level classes then add more specific attributes and operations.

A generalization hierarchy



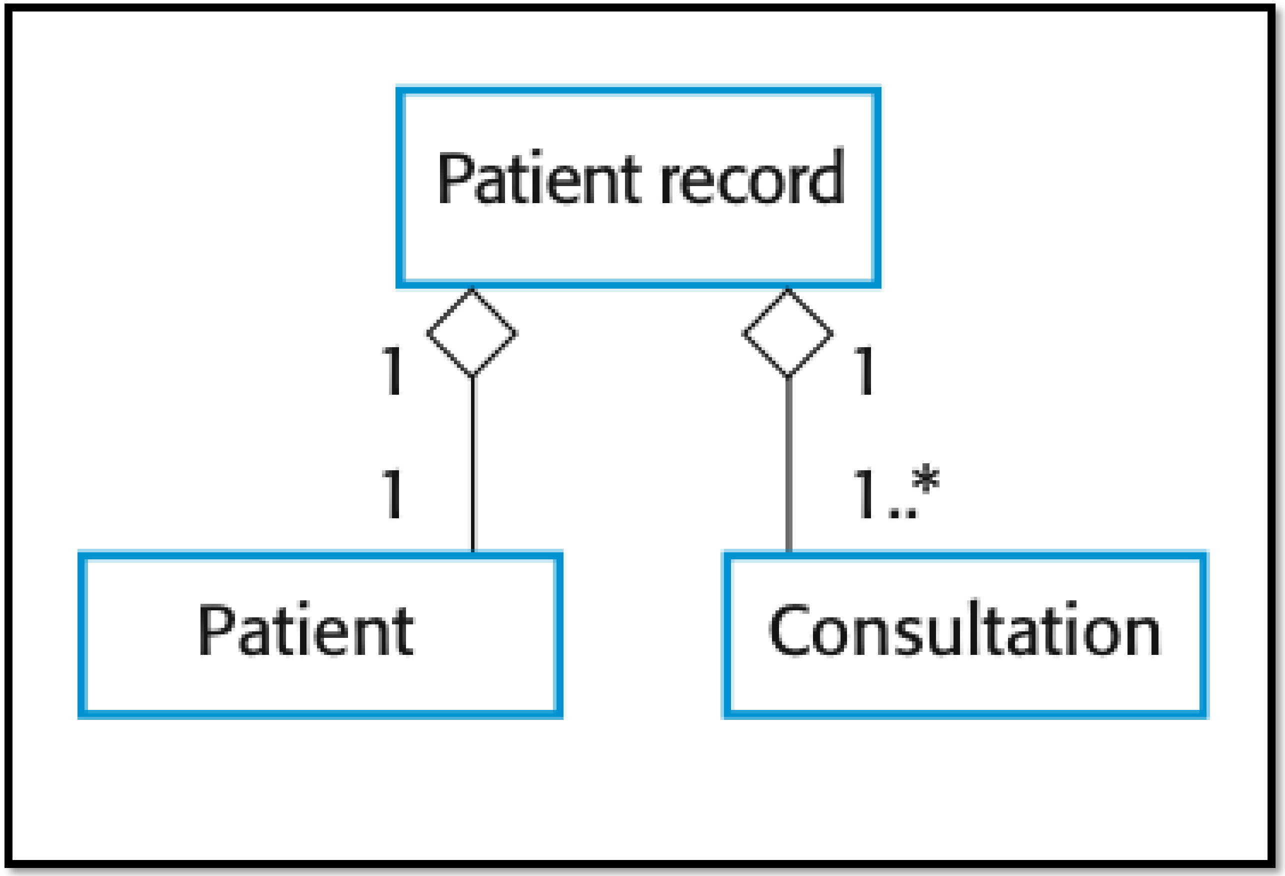
A generalization hierarchy with added detail



Object class aggregation models

- An aggregation model shows how classes that are collections are composed of other classes.
- Aggregation models are similar to the part-of relationship in semantic data models.

The aggregation association



Behavioral models

- Behavioral models are models of the dynamic behavior of a system as it is executing. They show what happens or what is supposed to happen when a system responds to a stimulus from its environment.
- You can think of these stimuli as being of two types:

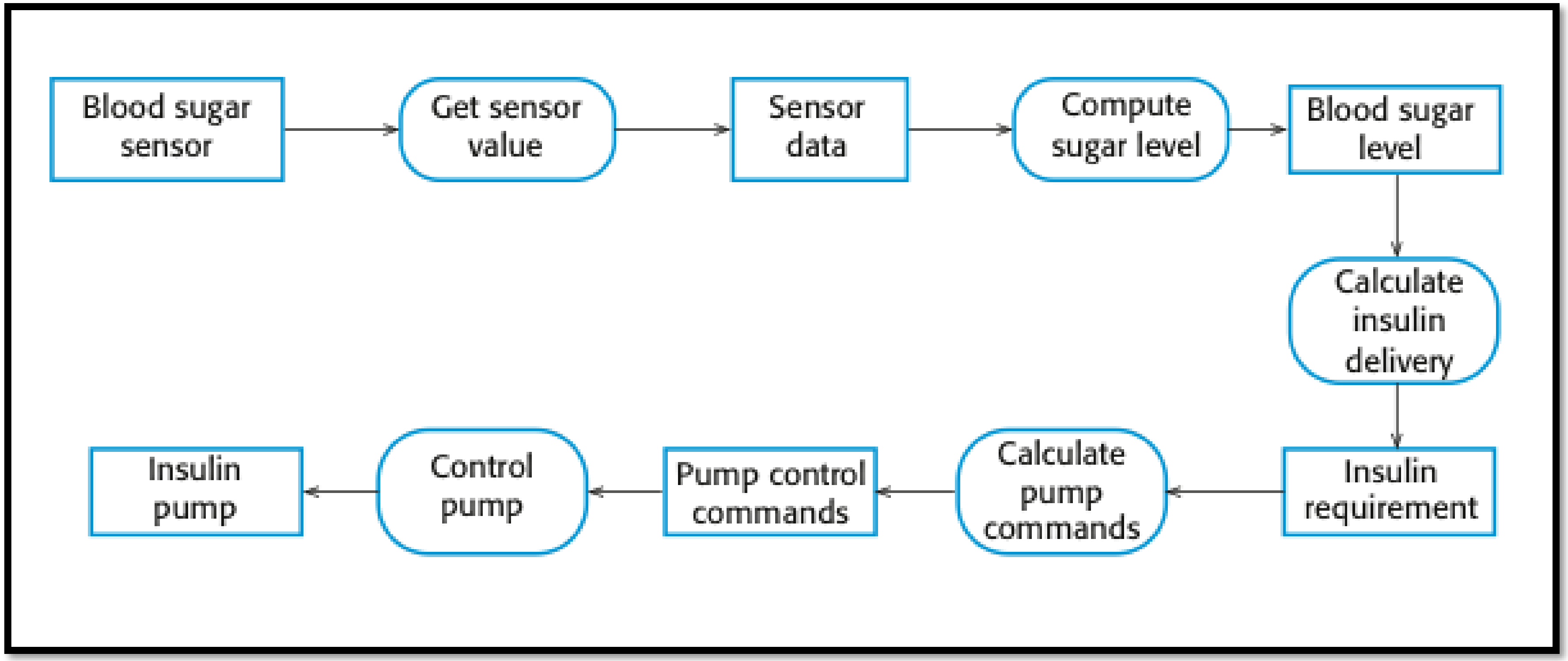
–Data Some data arrives that has to be processed by the system.

–Events Some event happens that triggers system processing. Events may have associated data, although this is not always the case.

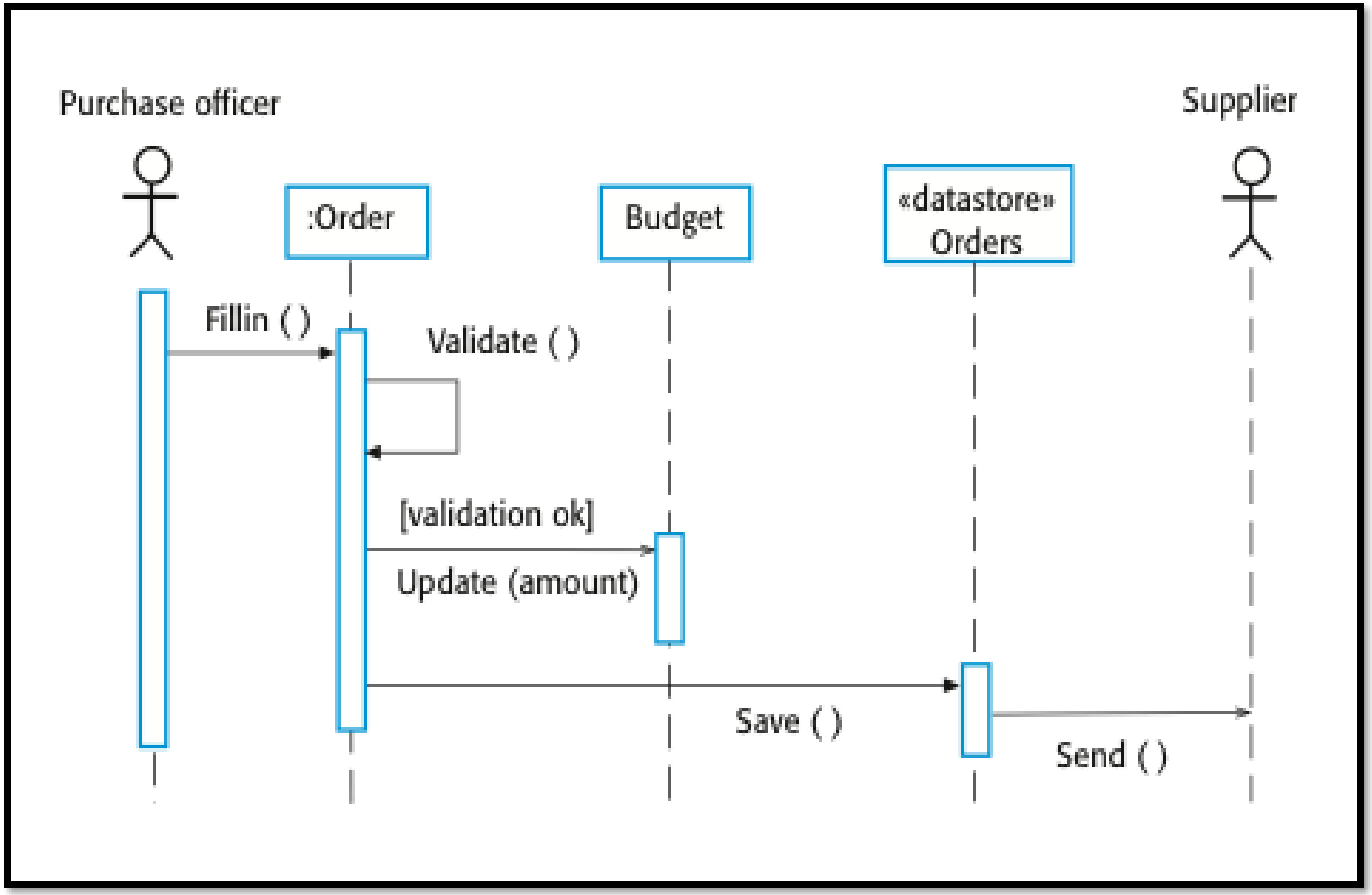
Data-driven modeling

- Many business systems are data-processing systems that are primarily driven by data. They are controlled by the data input to the system, with relatively little external event processing.
- Data-driven models show the sequence of actions involved in processing input data and generating an associated output.
- They are particularly useful during the analysis of requirements as they can be used to show end-to-end processing in a system.

An activity model of the insulin pump's operation



Order processing



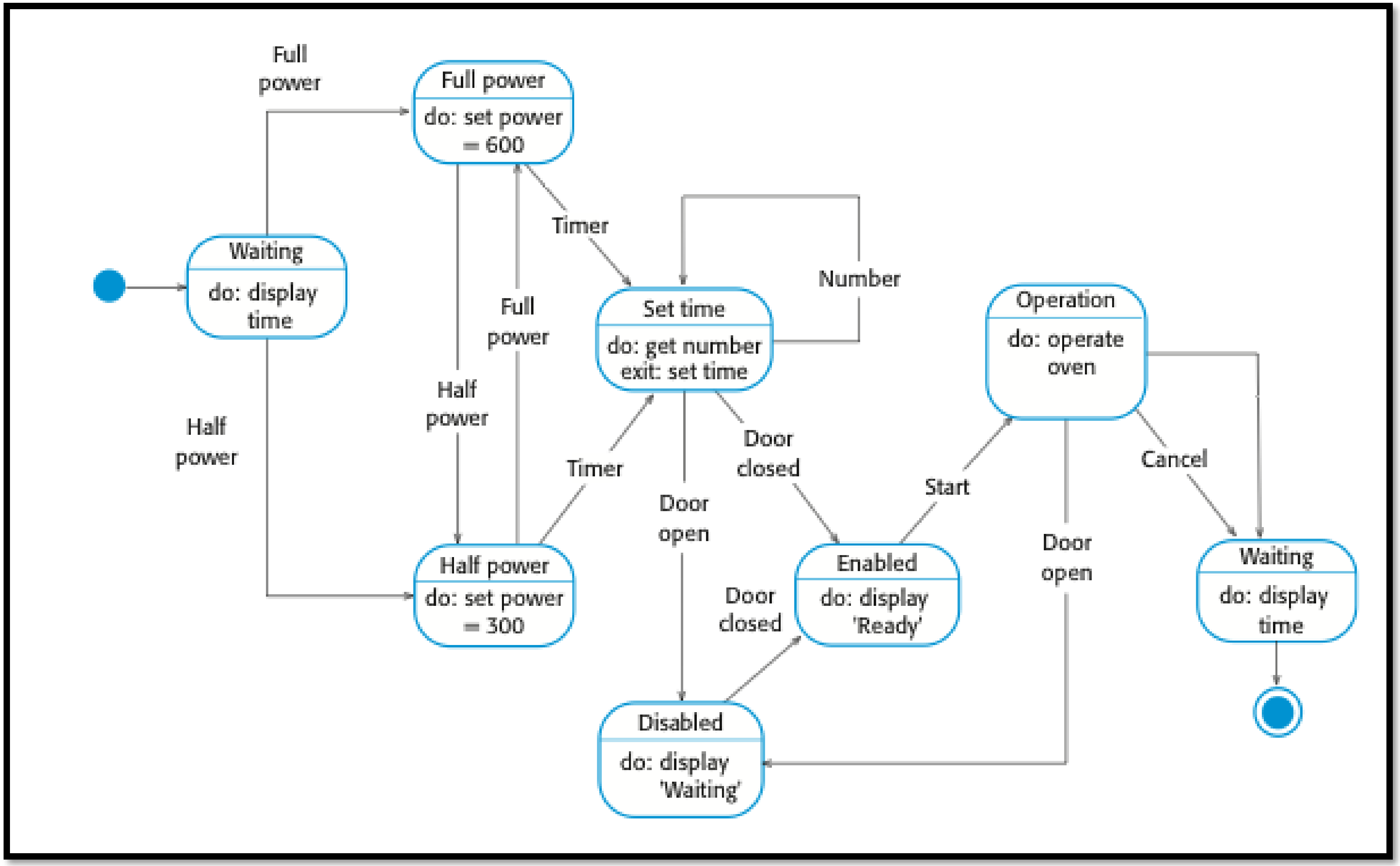
Event-driven modeling

- Real-time systems are often event-driven, with minimal data processing. For example, a landline phone switching system responds to events such as ‘receiver off hook’ by generating a dial tone.
- Event-driven modeling shows how a system responds to external and internal events.
- It is based on the assumption that a system has a finite number of states and that events (stimuli) may cause a transition from one state to another.

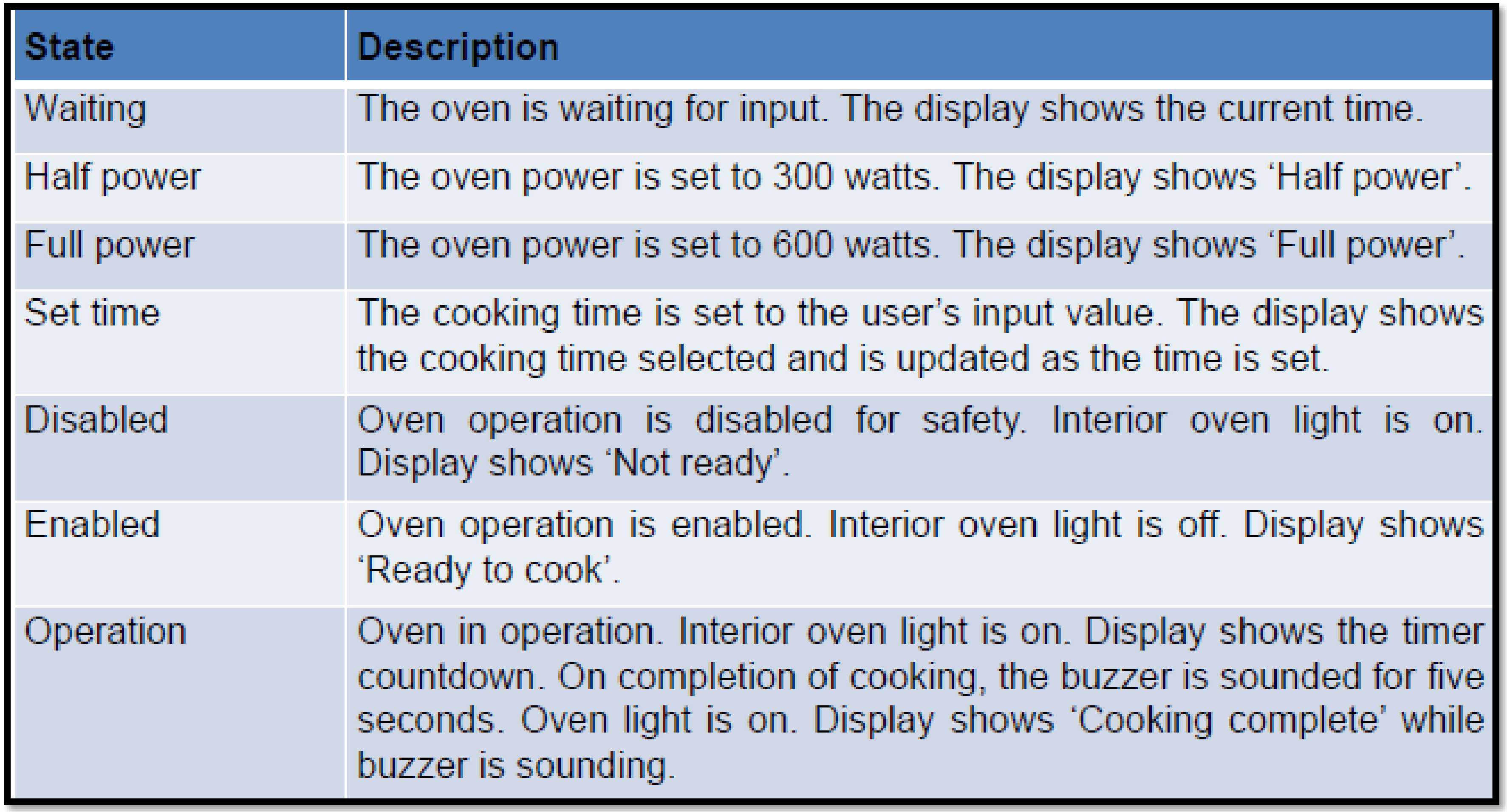
State machine models

- These model the behavior of the system in response to external and internal events.
- They show the system's responses to stimuli so are often used for modeling real-time systems.
- State machine models show system states as nodes and events as arcs between these nodes. When an event occurs, the system moves from one state to another.
- State charts are an integral part of the UML and are used to represent state machine models.

State diagram of a microwave oven

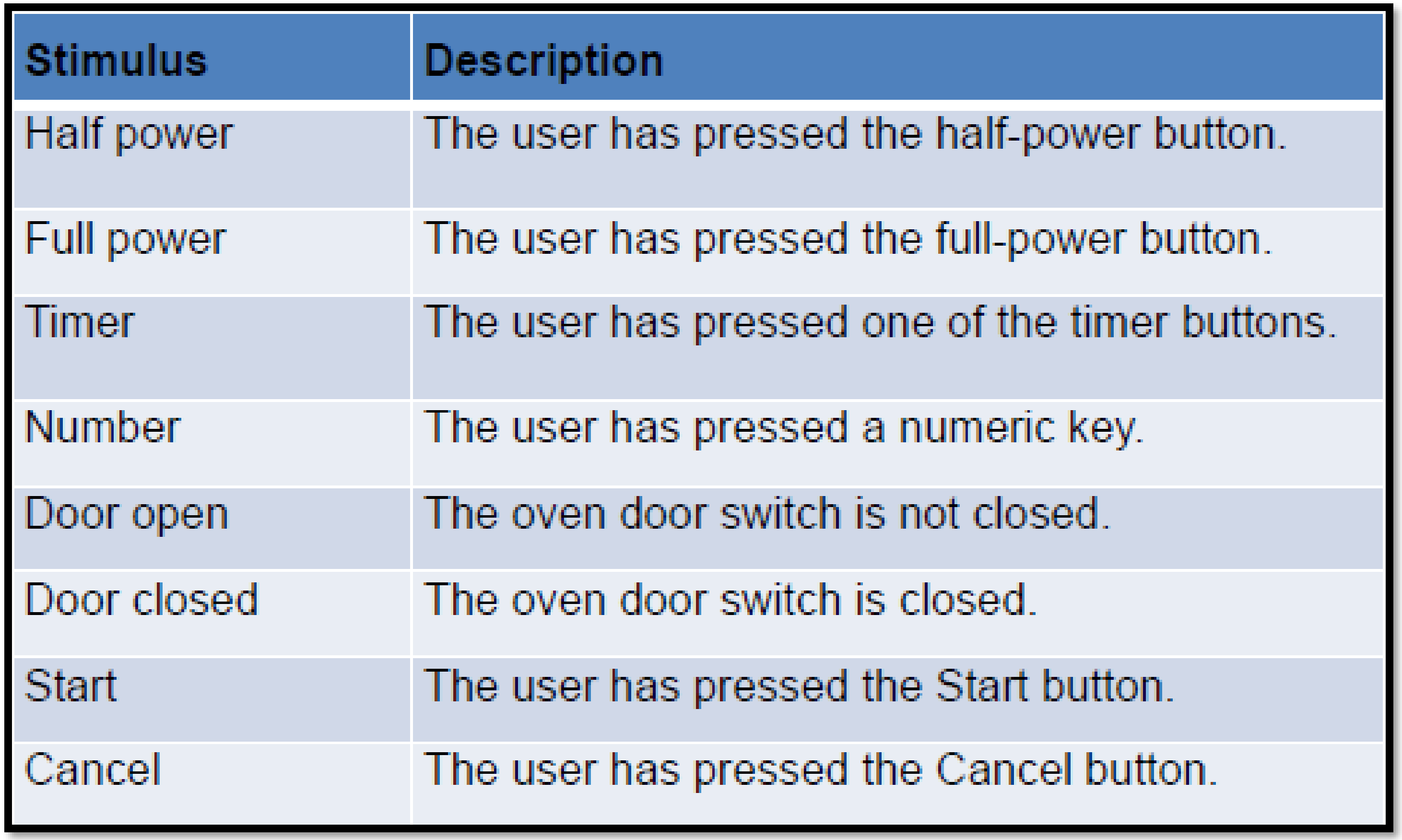


States and stimuli for the microwave oven - A



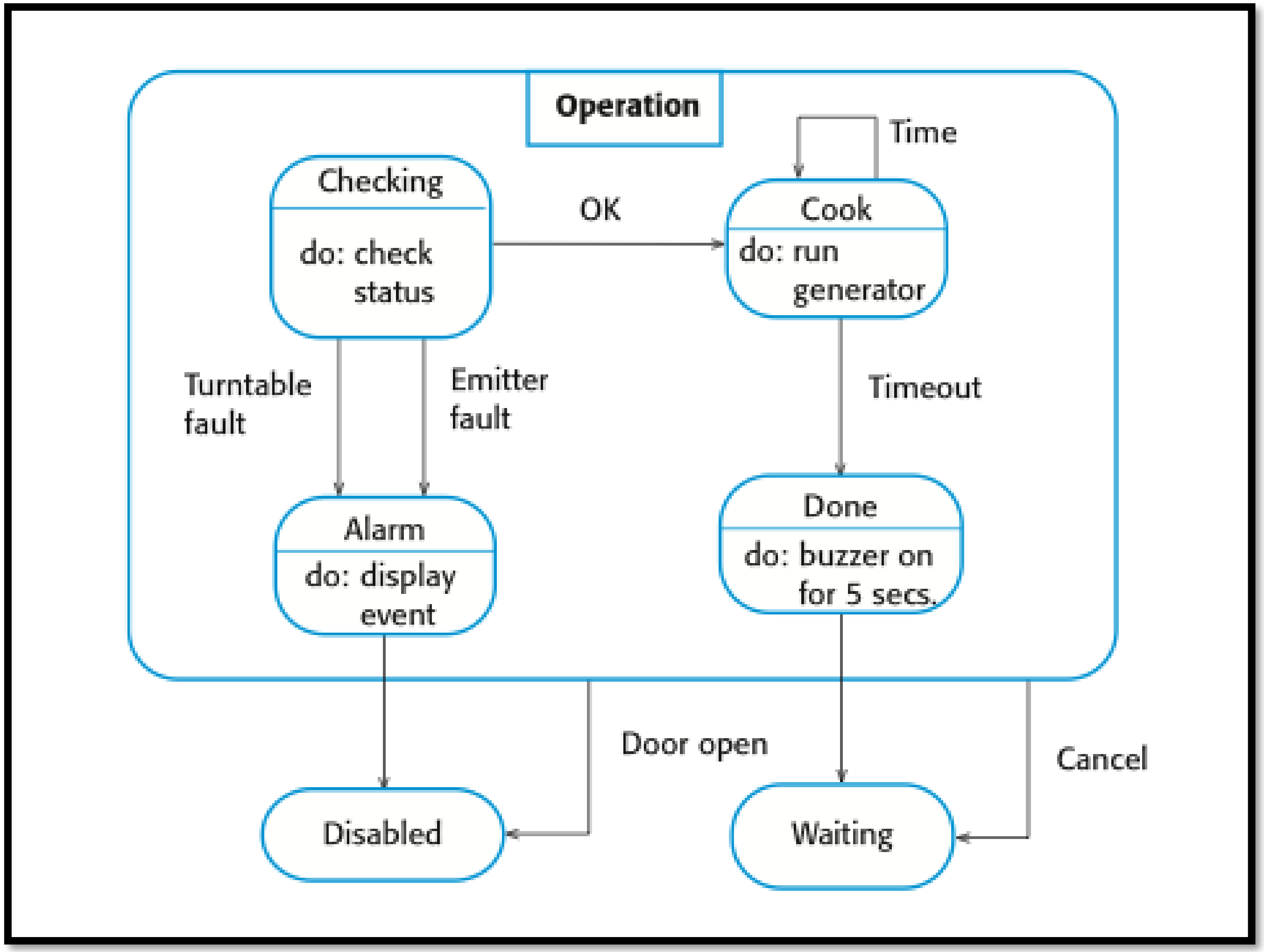
State	Description
Waiting	The oven is waiting for input. The display shows the current time.
Half power	The oven power is set to 300 watts. The display shows 'Half power'.
Full power	The oven power is set to 600 watts. The display shows 'Full power'.
Set time	The cooking time is set to the user's input value. The display shows the cooking time selected and is updated as the time is set.
Disabled	Oven operation is disabled for safety. Interior oven light is on. Display shows 'Not ready'.
Enabled	Oven operation is enabled. Interior oven light is off. Display shows 'Ready to cook'.
Operation	Oven in operation. Interior oven light is on. Display shows the timer countdown. On completion of cooking, the buzzer is sounded for five seconds. Oven light is on. Display shows 'Cooking complete' while buzzer is sounding.

States and stimuli for the microwave oven-B



Stimulus	Description
Half power	The user has pressed the half-power button.
Full power	The user has pressed the full-power button.
Timer	The user has pressed one of the timer buttons.
Number	The user has pressed a numeric key.
Door open	The oven door switch is not closed.
Door closed	The oven door switch is closed.
Start	The user has pressed the Start button.
Cancel	The user has pressed the Cancel button.

Microwave oven operation



Model-driven engineering

- Model-driven engineering (MDE) is an approach to software development where models rather than programs are the principal outputs of the development process.
- The programs that execute on a hardware/software platform are then generated automatically from the models.
- Proponents of MDE argue that this raises the level of abstraction in software engineering so that engineers no longer have to be concerned with programming language details or the specifics of execution platforms.

Usage of model-driven engineering

- Model-driven engineering is still at an early stage of development, and it is unclear whether or not it will have a significant effect on software engineering practice.
- Pros

–Allows systems to be considered at higher levels of abstraction

–Generating code automatically means that it is cheaper to adapt systems to new platforms.

- Cons

–Models for abstraction and not necessarily right for implementation.

–Savings from generating code may be outweighed by the costs of developing translators for new platforms.

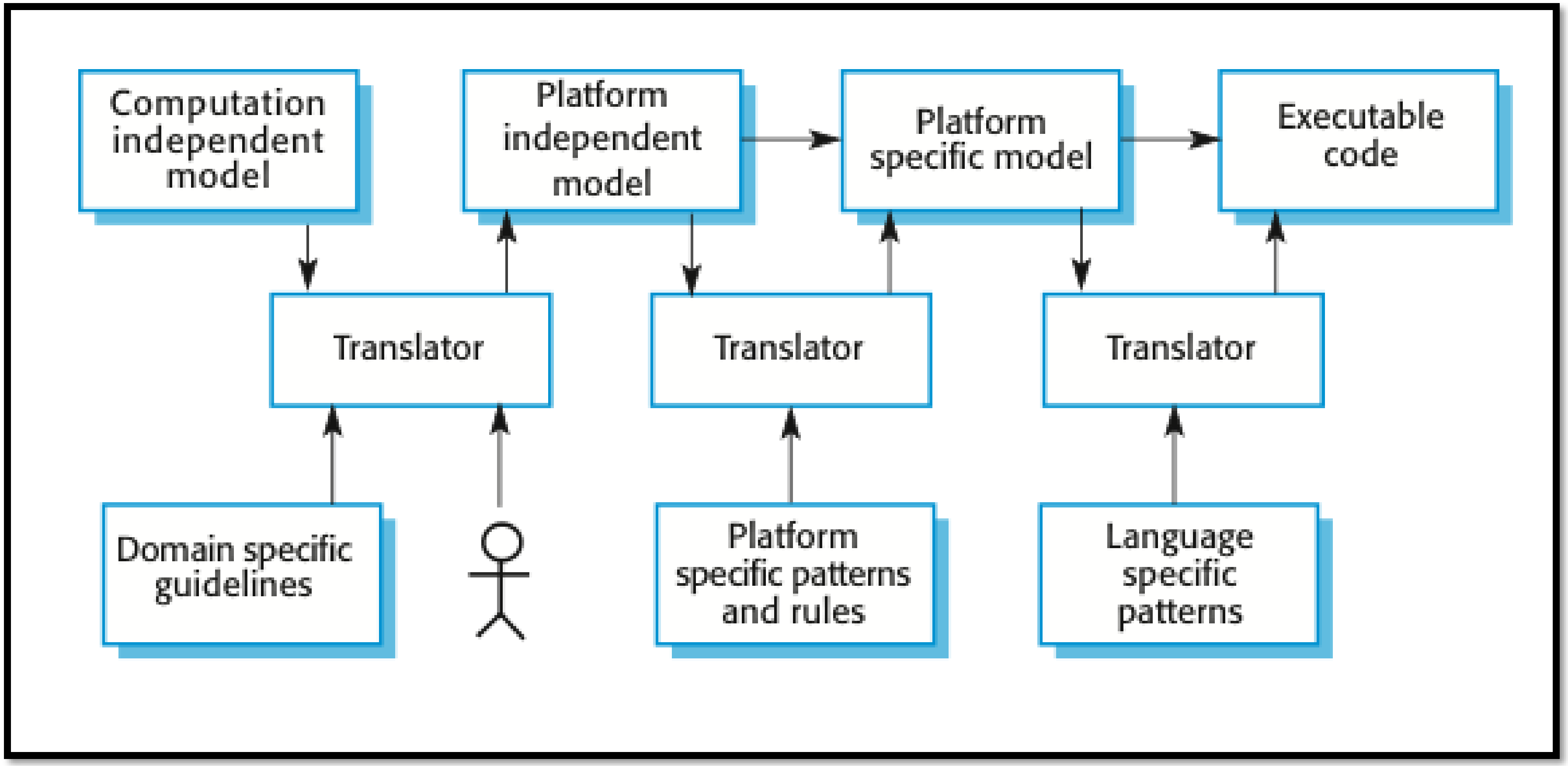
Model driven architecture

- Model-driven architecture (MDA) was the precursor of more general model-driven engineering
- MDA is a model-focused approach to software design and implementation that uses a subset of UML models to describe a system.
- Models at different levels of abstraction are created. From a high-level, platform independent model, it is possible, in principle, to generate a working program without manual intervention.

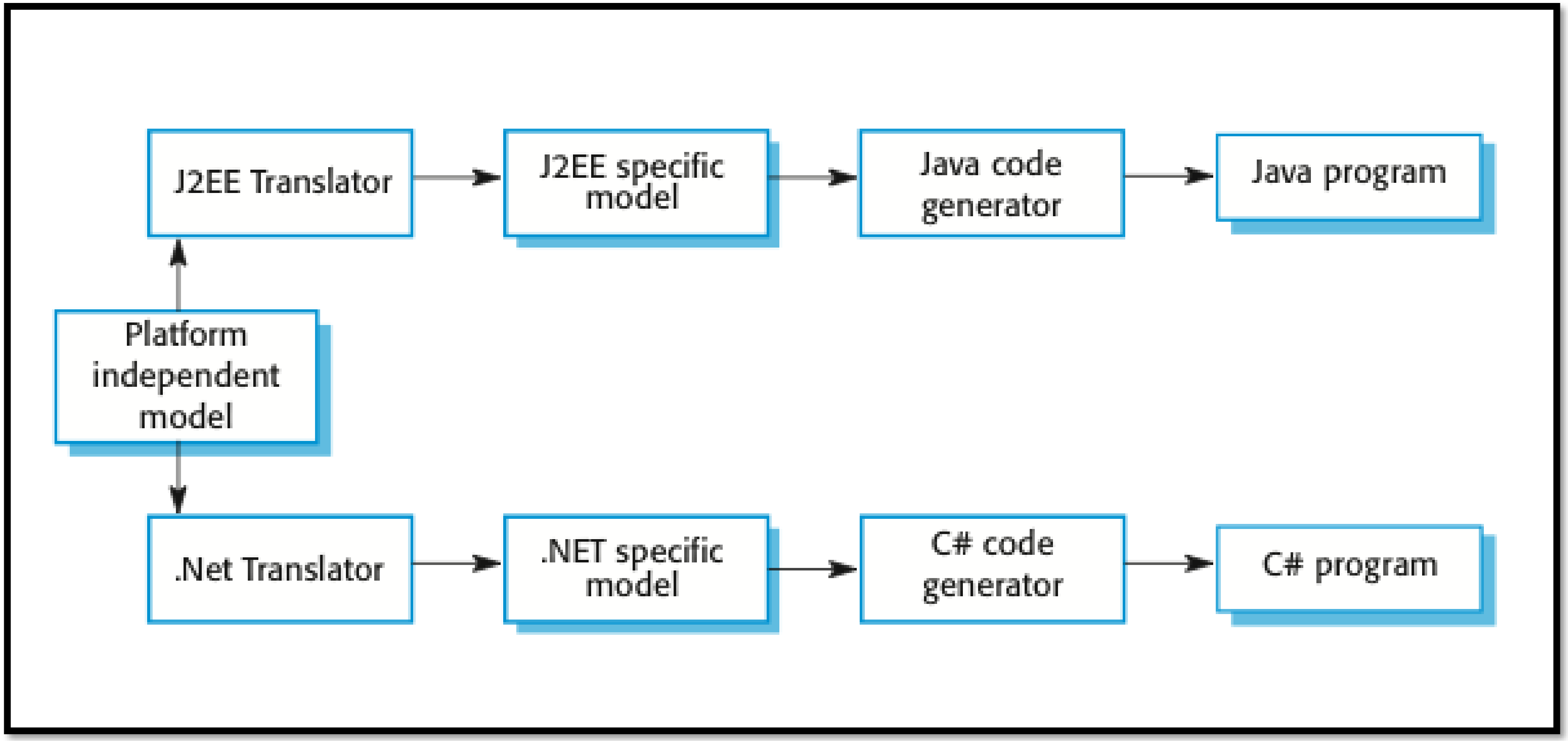
Types of model

- **A computation independent model (CIM)**
  - These model the important domain abstractions used in a system. CIMs are sometimes called domain models.
- **A platform independent model (PIM)**
  - These model the operation of the system without reference to its implementation. The PIM is usually described using UML models that show the static system structure and how it responds to external and internal events.
- ***Platform specific models (PSM)***
  - These are transformations of the platform- independent model with a separate PSM for each application platform. In principle, there may be layers of PSM, with each layer adding some platform-specific detail.

MDA transformations


Computation independent model
Platform independent model
Platform specific model
Executable code
Translator
Translator
Translator
Domain specific guidelines
Platform specific patterns and rules
Language specific patterns

Multiple platform-specific models



Agile methods and MDA

- The developers of MDA claim that it is intended to support an iterative approach to development and so can be used within agile methods.
- The notion of extensive up-front modeling contradicts the fundamental ideas in the agile manifesto and I suspect that few agile developers feel comfortable with model-driven engineering.
- If transformations can be completely automated and a complete program generated from a PIM, then, in principle, MDA could be used in an agile development process as no separate coding would be required.

Executable UML

- The fundamental notion behind model-driven engineering is that completely automated transformation of models to code should be possible.
- This is possible using a subset of UML 2, called Executable UML or xUML.

Features of executable UML

- To create an executable subset of UML, the number of model types has therefore been dramatically reduced to these 3 key types:

–Domain models that identify the principal concerns in a system. They are defined using UML class diagrams and include objects, attributes and associations.

–Class models in which classes are defined, along with their attributes and operations.

–State models in which a state diagram is associated with each class and is used to describe the life cycle of the class.

- The dynamic behavior of the system may be specified declaratively using the object constraint language (OCL), or may be expressed using UML's action language.

Key points

- Behavioral models are used to describe the dynamic behavior of an executing system. This behavior can be modeled from the perspective of the data processed by the system, or by the events that stimulate responses from a system.
- Activity diagrams may be used to model the processing of data, where each activity represents one process step.
- State diagrams are used to model a system’s behavior in response to internal or external events.
- Model-driven engineering is an approach to software development in which a system is represented as a set of models that can be automatically transformed to executable code.

Thank You

SOFTWARE DESIGN AND IMPLEMENTATION

OBJECT ORIENTED DESIGN

DESIGN PATTERNS

SOFTWARE IMPLEMENTATION

Object Oriented Design

- **Preview** - Object oriented design works around the entities and their characteristics instead of functions involved in the software system. This design strategies focuses on entities and its characteristics. The whole concept of software solution revolves around the engaged entities.

Important Concepts

- **Objects** - All entities involved in the solution design are known as objects. For example, person, banks, company and customers are treated as objects. Every entity has some attributes associated to it and has some methods to perform on the attributes.

- **Classes** - A class is a generalized description of an object. An object is an instance of a class. Class defines all the attributes, which an object can have and methods, which defines the functionality of the object. In the solution design, attributes are stored as variables and functionalities are defined by means of methods or procedures.

Important Concepts

- **Encapsulation** - In OOD, the attributes (data variables) and methods (operation on the data) are bundled together is called encapsulation. Encapsulation not only bundles important information of an object together, but also restricts access of the data and methods from the outside world. This is called information hiding.

Important Concepts

- **Inheritance** **-** OOD allows similar classes to stack up in hierarchical manner where the lower or sub-classes can import, implement and re-use allowed variables and methods from their immediate super classes. This property of OOD is known as inheritance. This makes it easier to define specific class and to create generalized classes from specific ones.

Important Concepts

- **Polymorphism** - OOD languages provide a mechanism where methods performing similar tasks but vary in arguments, can be assigned same name. This is called polymorphism, which allows a single interface performing tasks for different types. Depending upon how the function is invoked, respective portion of the code gets executed.

Design Process

- Software design process can be perceived as series of well-defined steps.
  - A solution design is created from requirement or previous used system and/or system sequence diagram.
  - Objects are identified and grouped into classes on behalf of similarity in attribute characteristics.
  - Class hierarchy and relation among them is defined.
  - Application framework is defined.

Software Design Approaches

Top Down Design

- We know that a system is composed of more than one sub-systems and it contains a number of components. Further, these sub-systems and components may have their on set of sub-system and components and creates hierarchical structure in the system.
- Top-down design takes the whole software system as one entity and then decomposes it to achieve more than one sub-system or component based on some characteristics. Each sub-system or component is then treated as a system and decomposed further. This process keeps on running until the lowest level of system in the top-down hierarchy is achieved.

Software Design Approaches

Top Down Design

- Top-down design starts with a generalized model of system and keeps on defining the more specific part of it. When all components are composed the whole system comes into existence.
- Top-down design is more suitable when the software solution needs to be designed from scratch and specific details are unknown.

Software Design Approaches

Bottom-up Design

- The bottom up design model starts with most specific and basic components. It proceeds with composing higher level of components by using basic or lower level components. It keeps creating higher level components until the desired system is not evolved as one single component. With each higher level, the amount of abstraction is increased.

Software Design Approaches

Bottom-up Design

- Bottom-up strategy is more suitable when a system needs to be created from some existing system, where the basic primitives can be used in the newer system.
- Both, top-down and bottom-up approaches are not practical individually. Instead, a good combination of both is used.

Design Patterns

Design Patterns

- By definition, Design Patterns are reusable solutions to commonly occurring problems(in the context of software design). Design patterns were started as best practices that were applied again and again to similar problems encountered in different contexts.

Design Patterns

Key Characteristics of Design Patterns

- **Reusability**: Patterns can be applied to different projects and problems, saving time and effort in solving similar issues.
- **Standardization**: They provide a shared language and understanding among developers, helping in communication and collaboration.
- **Efficiency**: By using these popular patterns, developers can avoid finding the solution to same recurring problems, which leads to faster development.
- **Flexibility**: Patterns are abstract solutions/templates that can be adapted to fit various scenarios and requirements.

Design Patterns

Why Learn Design Patterns?

- There are multiple reasons to learn design patterns:
- Code that is simpler to comprehend, update, and expand is produced with the help of design patterns.
- They offer solutions that have been tried and tested as well as best practices.
- Learning this enables them to quickly and effectively address similar challenges in various projects.
- Developers can produce reusable components that can be utilized in a variety of applications by implementing design patterns.
- This reduces redundancy and saves development time.

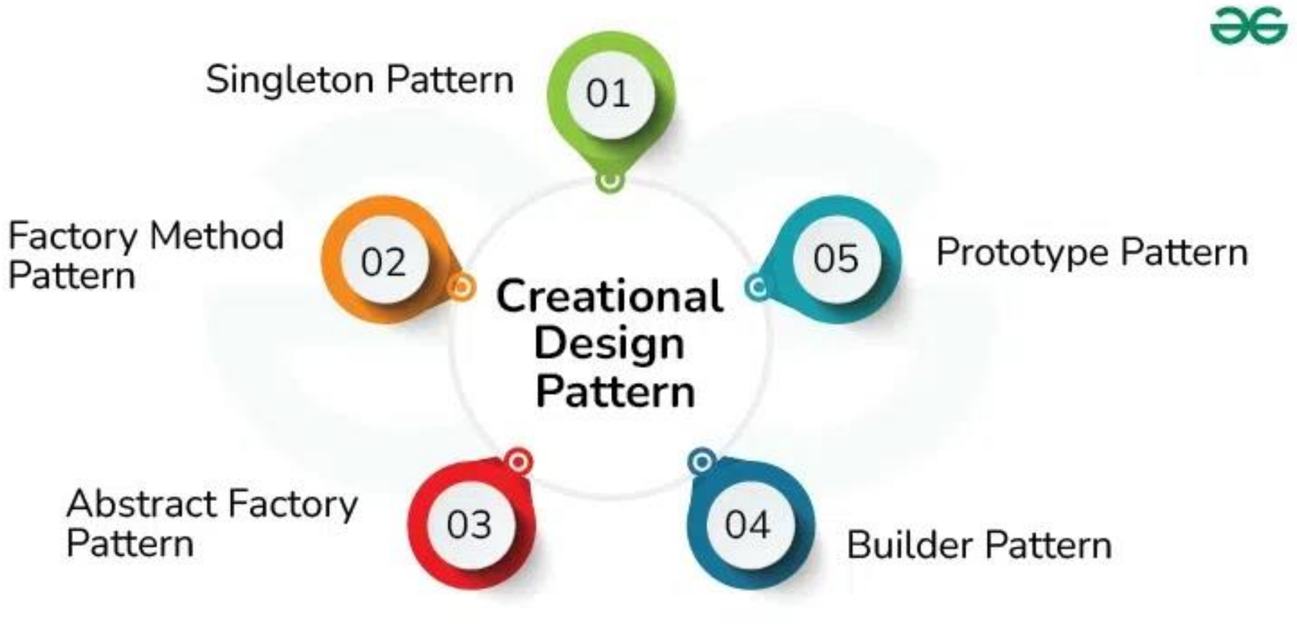
Design Patterns

Types of Software Design Patterns - There are three types of Design Patterns:

- Creational Design Pattern
- Structural Design Pattern
- Behavioral Design Pattern

Design Patterns

- 1. Creational Design Patterns - Creational Design Patterns focus on the process of object creation or problems related to object creation. They help in making a system independent of how its objects are created, composed and represented.

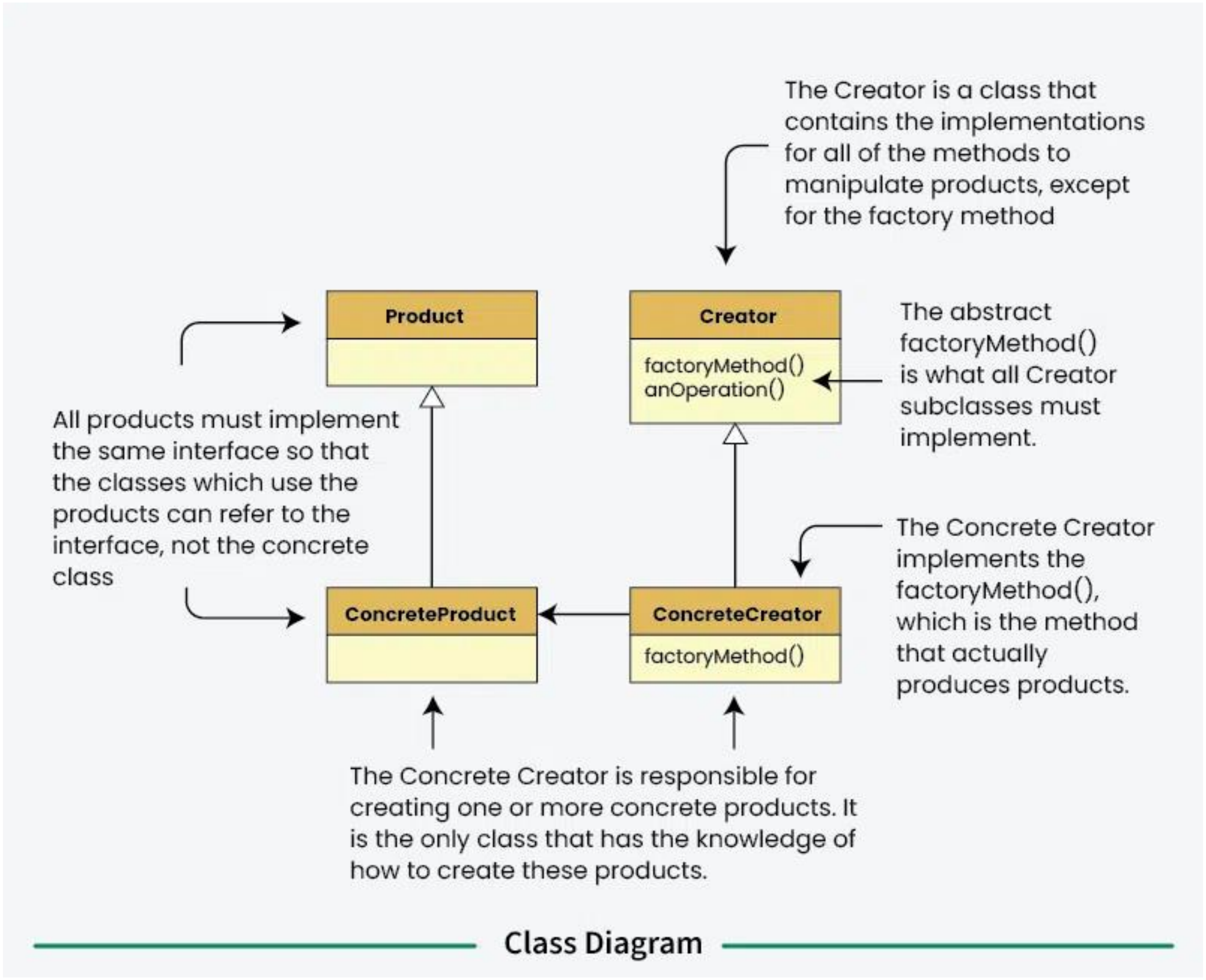


Design Patterns

Types of Creational Design Patterns:

- Factory Method Design Pattern
  - This pattern is typically helpful when it's necessary to separate the construction of an object from its implementation.
  - With the use of this design pattern, objects can be produced without having to define the exact class of object to be created.
- Abstract Factory Method Design Pattern
  - Abstract Factory pattern is almost similar to Factory Pattern and is considered as another layer of abstraction over factory pattern.
  - Abstract Factory patterns work around a super-factory which creates other factories.

Core Components of the Factory Method


The Creator is a class that contains the implementations for all of the methods to manipulate products, except for the factory method
Product
Creator
factoryMethod()
anOperation()
The abstract factoryMethod() is what all Creator subclasses must implement.
All products must implement the same interface so that the classes which use the products can refer to the interface, not the concrete class
The Concrete Creator implements the factoryMethod(), which is the method that actually produces products.
ConcreteProduct
ConcreteCreator
factoryMethod()
The Concrete Creator is responsible for creating one or more concrete products. It is the only class that has the knowledge of how to create these products.
Class Diagram

Core Components of the Factory Method

- **Creator**: This is an abstract class or an interface that declares the factory method. The creator typically contains a method that serves as a factory for creating objects. It may also contain other methods that work with the created objects.
- **Concrete Creator**: Concrete Creator classes are subclasses of the Creator that implement the factory method to create specific types of objects. Each Concrete Creator is responsible for creating a particular product.
- **Product**: This is the interface or abstract class for the objects that the factory method creates. The Product defines the common interface for all objects that the factory method can create.
- **Concrete Product**: Concrete Product classes are the actual objects that the factory method creates. Each Concrete Product class implements the Product interface or extends the Product abstract class.

Core Components of the Factory Method

- **Implementation of the Factory Method in C++**

Let's implement the Factory Method Pattern in C++ step by step, explaining each part in detail. In this example, we'll create a simple shape factory that produces different shapes (e.g., Circle and Square).

- **Step 1: Define the Abstract Product (Shape)**

```
// Abstract product class
class Shape {
public:
   virtual void draw() = 0;
   virtual ~Shape() {} // Virtual destructor for polymorphism
};
```

- *Here, we've defined an abstract class* ***Shape*** *with a pure virtual function draw(). This class represents the abstract product that all concrete products must inherit from.*

Core Components of the Factory Method

- **Step 2: Define Concrete Products (Circle and Square)**

```
// Concrete product class - Circle
class Circle : public Shape {
public:
   void draw() override {
      std::cout<<"Drawing a Circle"<< std::endl;
   }
};

// Concrete product class - Square
class Square : public Shape {
public:
   void draw() override {
      std::cout<<"Drawing a Square"<<std::endl;
   }
};
```

- *Here, we have two concrete classes, Circle and Square, that inherits from the Shape abstract class. Each concrete product (Circle and Square) provides its implementation of the draw() method.*

Core Components of the Factory Method

- **Step 3: Define the Abstract Creator**

```
// Abstract creator class
class ShapeFactory {
public:
    virtual Shape* createShape() = 0;
    virtual ~ShapeFactory() {} // Virtual destructor for polymorphism
};
```

- *The abstract creator class, **ShapeFactory,** declare a pure virtual function **createShape()**, which will be implemented by concrete creators to create specific shapes.*

Core Components of the Factory Method

- **Step 4: Define Concrete Creators (CircleFactory and Square Factory)**

```
// Concrete creator class - CircleFactory
class CircleFactory : public ShapeFactory {
public:
    Shape* createShape() override {
        return new Circle();
    }
};

// Concrete creator class - SquareFactory
class SquareFactory : public ShapeFactory {
public:
    Shape* createShape() override {
        return new Square();
    }
};
```

- *In this step, we've created two concrete creator classes,* ***CircleFactory and SquareFactory,*** *which implement the* ***createShape()*** *method to create instances of* ***Circle*** *and* ***Square,*** *respectively.*

Core Components of the Factory Method

- **Step 5: Client Code**

```
int main() {
   ShapeFactory* circleFactory = new CircleFactory();
   ShapeFactory* squareFactory = new SquareFactory();

   Shape* circle = circleFactory->createShape();
   Shape* square = squareFactory->createShape();

   circle->draw(); // Output: Drawing a Circle
   square->draw(); // Output: Drawing a Square

   delete circleFactory;
   delete squareFactory;
   delete circle;
   delete square;

   return 0;
}
```

- *In this client code, we first create instances of the concrete creators (**circleFactory** and **squareFactory)** and then use them to create instances of concrete products (**cirlce** and **square).** Finally, we call the **draw()** method on these objects, which produces the expected output.*

Design Patterns

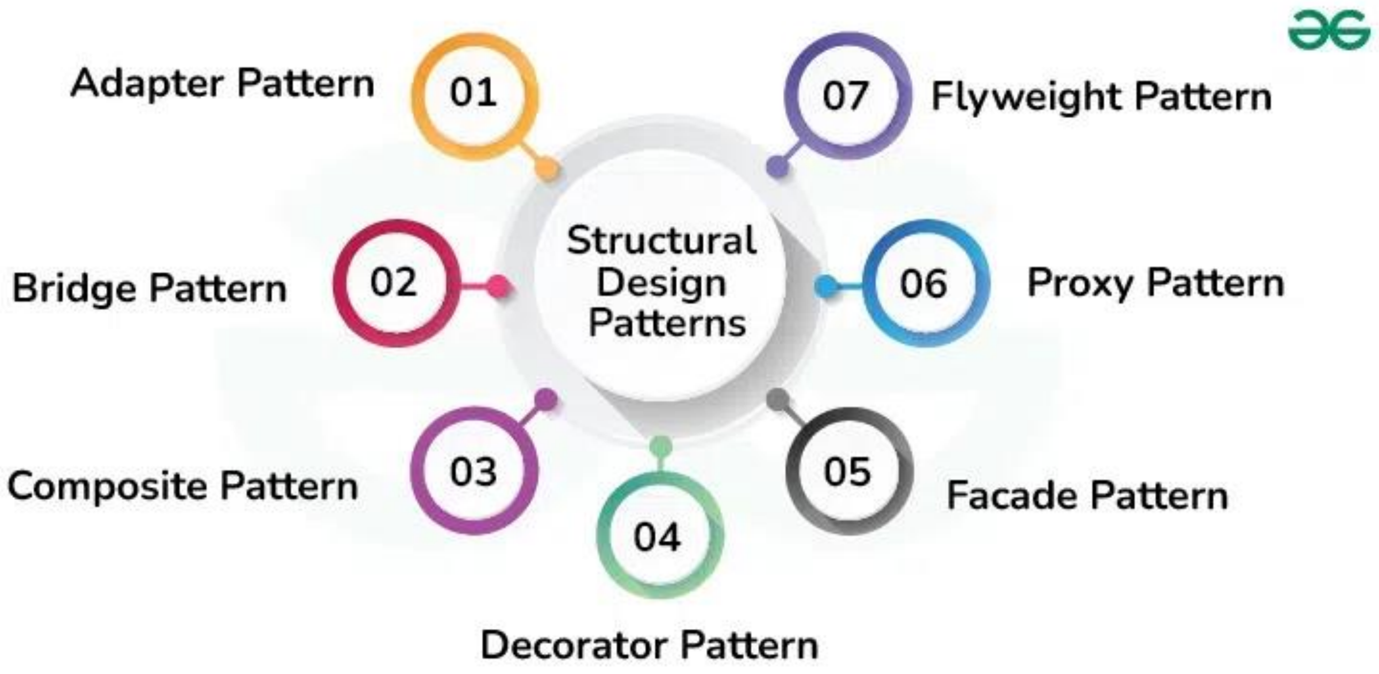
- Singleton Method Design Pattern
  - Of all, the Singleton Design pattern is the most straightforward to understand.
  - It guarantees that a class has just one instance and offers a way to access it globally.
- Prototype Method Design Pattern
  - Prototype allows us to hide the complexity of making new instances from the client.
  - The concept is to copy an existing object rather than creating a new instance from scratch, something that may include costly operations

Design Patterns

- Builder Method Design Pattern
  - To “Separate the construction of a complex object from its representation so that the same construction process can create different representations.” Builder pattern is used
  - It helps in constructing a complex object step by step and the final step will return the object.

Design Patterns

- Structural Design Patterns - Structural Design Patterns solves problems related to how classes and objects are composed/assembled to form larger structures which are efficient and flexible in nature. Structural class patterns use inheritance to compose interfaces or implementations.



Design Patterns

Types of Structural Design Patterns:

- Adapter Method Design Pattern
  - The adapter pattern convert the interface of a class into another interface clients expect.
  - Adapter lets classes work together that couldn’t otherwise because of incompatible interfaces.
- Bridge Method Design Pattern
  - The bridge pattern allows the Abstraction and the Implementation to be developed independently.
  - The client code can access only the Abstraction part without being concerned about the Implementation part.

Design Patterns

Adapter Pattern example in C++ Design Patterns

Problem Statement - Suppose you have a legacy printer class that only understands commands in uppercase, and a modern computer class that sends commands in lowercase. You need to make the modern computer work with the legacy printer without modifying the existing printer class.

Implementation of the Adapter Pattern in C++ Design Patterns:

It defines three classes:

LegacyPrinter is the legacy component (**Adaptee**). It has a method printInUppercase that can print text in uppercase.

ModernComputer is the modern client class. It has a method sendCommand to send commands, but it sends them in lowercase.

PrinterAdapter is the adapter class. It encapsulates the LegacyPrinter and adapts it to work with the ModernComputer.

Design Patterns

LegacyPrinter Class - The LegacyPrinter class has a single method, printInUppercase, which takes a string as an argument and prints it in uppercase.

```
/ Legacy Printer (Adaptee)
class LegacyPrinter {
public:
    void printInUppercase(const std::string& text) {
        std::cout << "Printing: " << text << std::endl;
    }
};
```

Design Patterns

ModernComputer Class - The ModernComputer class has a single method, sendCommand, which also takes a string as an argument but sends it as a command (in lowercase).

```
// Modern Computer (Client)
class ModernComputer {
public:
    void sendCommand(const std::string& command) {
        std::cout << "Sending command: " << command << std::endl;
    }
};
```

Design Patterns

PrinterAdapter Class

The PrinterAdapter class is the adapter. It contains an instance of LegacyPrinter. It has a method sendCommand that takes a lowercase command as an argument.

```
// Adapter class to make the LegacyPrinter compatible with ModernComputer
class PrinterAdapter {
private:
    LegacyPrinter legacyPrinter;

public:
    void sendCommand(const std::string& command) {
        // Convert the command to uppercase and pass it to the LegacyPrinter
        std::string uppercaseCommand = command;
        for (char& c : uppercaseCommand) {
            c = std::toupper(c);
        }
        legacyPrinter.printInUppercase(uppercaseCommand);
    }
};
```

Design Patterns

Explanation of the above code:

- ✓ It creates a copy of the lowercase command as uppercaseCommand.
- ✓ It iterates over each character in uppercaseCommand and converts it to uppercase using std::toupper.
- ✓ It then calls the printInUppercase method of the encapsulated LegacyPrinter with the uppercaseCommand.
- ✓ Essentially, the adapter class converts the lowercase command from the ModernComputer into uppercase and delegates it to the LegacyPrinter.

Design Patterns

Main Function: In the main function, the code demonstrates how the ModernComputer and the PrinterAdapter work together:

```
int main() {
    ModernComputer computer;
    PrinterAdapter adapter;

    computer.sendCommand("Print this in lowercase");
    adapter.sendCommand("Print this in lowercase (adapted)");

    return 0;
}
```

The explanation of the above code:

- An instance of ModernComputer named computer is created.
- An instance of PrinterAdapter named adapter is created.
- The computer sends a command in lowercase: "Print this in lowercase."
- The adapter receives the command, adapts it to uppercase, and sends it to the LegacyPrinter

Design Patterns

- Composite Method Design Pattern
  - As a partitioning design pattern, the composite pattern characterizes a collection of items that are handled the same way as a single instance of the same type of object.
  - The intent of a composite is to “compose” objects into tree structures to represent part-whole hierarchies.
- Decorator Method Design Pattern
  - It allows us to dynamically add functionality and behavior to an object without affecting the behavior of other existing objects within the same class.
  - We use inheritance to extend the behavior of the class. This takes place at compile-time, and all the instances of that class get the extended behavior.

Design Patterns

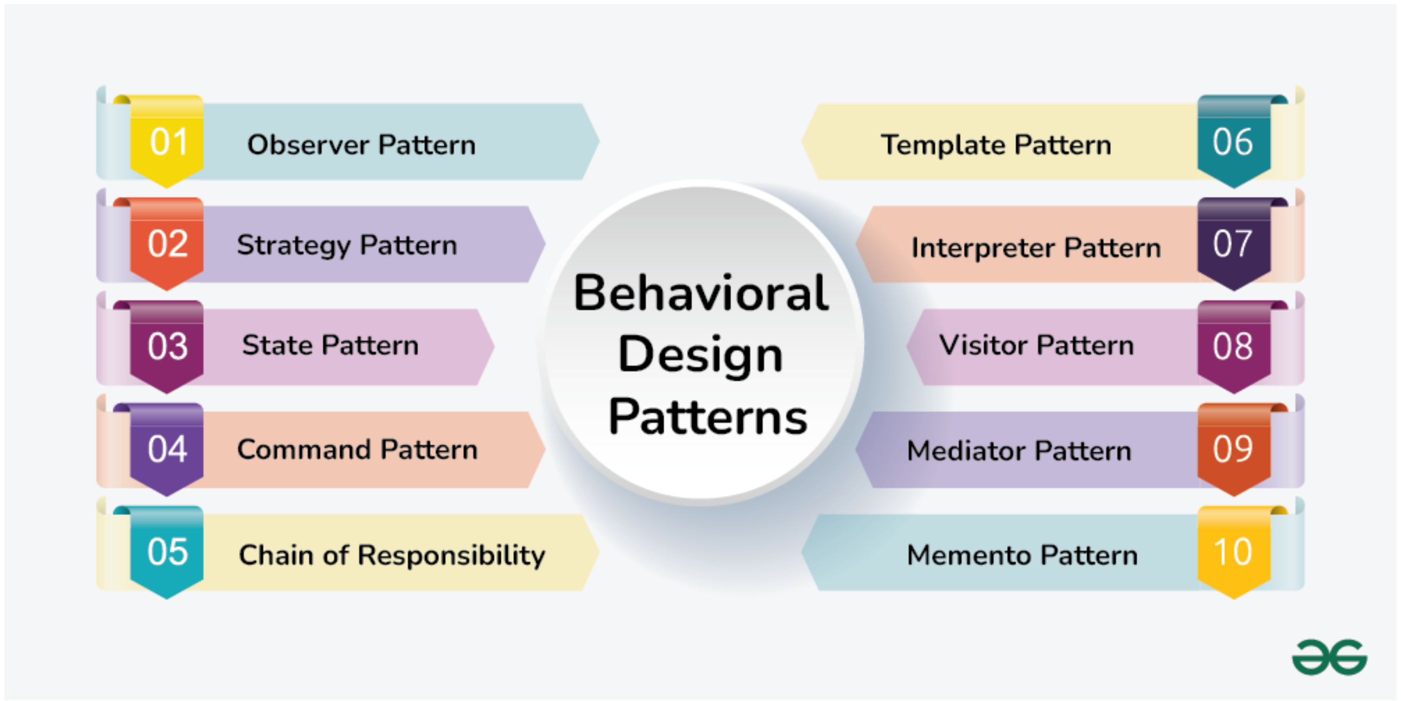
- Facade Method Design Pattern
  - Facade Method Design Pattern provides a unified interface to a set of interfaces in a subsystem.
  - Facade defines a high-level interface that makes the subsystem easier to use.
- Flyweight Method Design Pattern
  - This pattern provides ways to decrease object count thus improving application required objects structure.
  - Flyweight pattern is used when we need to create a large number of similar objects.

Design Patterns

- Proxy Method Design Pattern
  - Proxy means ‘in place of’, representing’ or ‘in place of’ or ‘on behalf of’ are literal meanings of proxy and that directly explains Proxy Design Pattern.
  - Proxies are also called surrogates, handles, and wrappers. They are closely related in structure, but not purpose, to Adapters and Decorators.

Design Patterns

- Behavioral Design Patterns - Behavioral Patterns are concerned with algorithms and the assignment of responsibilities between objects. Behavioral patterns describe not just patterns of objects or classes but also the patterns of communication between them. These patterns characterize complex control flow that’s difficult to follow at run-time.



Design Patterns

Types of Behavioral Design Patterns:

- Chain Of Responsibility Method Design Pattern
  - Chain of responsibility pattern is used to achieve loose coupling in software design where a request from the client is passed to a chain of objects to process them.
  - Later, the object in the chain will decide themselves who will be processing the request and whether the request is required to be sent to the next object in the chain or not.
- Command Method Design Pattern
  - A behavioral design pattern called the Command Pattern transforms a request into an independent object with all of the information's request
  - This object can be passed around, stored, and executed at a later time.

Design Patterns

Implementation of Chain of Responsibility Pattern in C++:

AuthenticationHandler (Handler Interface):

- This is an abstract base class that defines the interface for all authentication handlers. It declares two pure virtual functions:

- setNextHandler(AuthenticationHandler* handler): This function allows setting the next handler in the chain.
- handleRequest(const std::string& request): This function is responsible for handling authentication requests

Design Patterns

Two concrete handler classes are defined:

1. **UsernamePasswordHandler**: This handler checks if the authentication request is for "username_password." If it is, it prints a message indicating successful authentication using a username and password. If the request is not for "username_password," it forwards the request to the next handler in the chain, if available. If there is no next handler, it prints a message indicating an invalid authentication method.
2. **OAuthHandler**: This handler checks if the authentication request is for "oauth_token." If it is, it prints a message indicating successful authentication using an OAuth token. If the request is not for "oauth_token," it forwards the request to the next handler in the chain, if available. If there is no next handler, it prints a message indicating an invalid authentication method

Design Patterns

Note: *In the main function, two instances of the concrete handlers are created: usernamePasswordHandler and oauthHandler.*

Main Function:

Three authentication requests are made using handleRequest:

1. First, an "oauth_token" request is sent. The request is successfully handled by the oauthHandler.
2. Second, a "username_password" request is sent. This time, the request is successfully handled by the usernamePasswordHandler.
3. Third, an "invalid_method" request is sent. None of the handlers in the chain can handle this request, so an "Invalid authentication method" message is printed.

Design Patterns

- Interpreter Method Design Pattern
  - Interpreter pattern is used to defines a grammatical representation for a language and provides an interpreter to deal with this grammar.
- Mediator Method Design Pattern
  - It enables decoupling of objects by introducing a layer in between so that the interaction between objects happen via the layer.
- Memento Method Design Patterns
  - It is used to return an object's state to its initial state.
  - You might wish to create checkpoints in your application and return to them at a later time when it develops.

Design Patterns

- Observer Method Design Pattern
  - It establishes a one-to-many dependency between objects, meaning that all of the dependents (observers) of the subject are immediately updated and notified when the subject changes.
- State Method Design Pattern
  - When an object modifies its behavior according to its internal state, the state design pattern is applied.
  - If we have to change the behavior of an object based on its state, we can have a state variable in the Object and use the if-else condition block to perform different actions based on the state.

Design Patterns

- Strategy Method Design Pattern
  - It is possible to select an object's behavior at runtime by utilizing the Strategy Design Pattern.
  - Encapsulating a family of algorithms into distinct classes that each implement a common interface is the foundation of the Strategy pattern.
- Template Method Design Pattern
  - The template method design pattern defines an algorithm as a collection of skeleton operations, with the child classes handling the implementation of the specifics.
  - The parent class maintains the overall structure and flow of the algorithm.
- Visitor Method Design Pattern
  - It is used when we have to perform an operation on a group of similar kind of Objects. With the help of visitor pattern, we can move the operational logic from the objects to another class.

Software Implementation

Structured Programming

- In the process of coding, the lines of code keep multiplying, thus, size of the software increases. Gradually, it becomes next to impossible to remember the flow of program.
- If one forgets how software and its underlying programs, files, procedures are constructed it then becomes very difficult to share, debug and modify the program.
- The solution to this is structured programming. It encourages the developer to use subroutines and loops instead of using simple jumps in the code, thereby bringing clarity in the code and improving its efficiency Structured programming also helps programmer to reduce coding time and organize code properly.

Structured Programming

- Structured programming states how the program shall be coded. Structured programming uses three main concepts:
  - **Top-down analysis** - A software is always made to perform some rational work. This rational work is known as problem in the software parlance. Thus it is very important that we understand how to solve the problem. Under top-down analysis, the problem is broken down into small pieces where each one has some significance. Each problem is individually solved and steps are clearly stated about how to solve the problem.

Structured Programming

- **Modular Programming** - While programming, the code is broken down into smaller group of instructions. These groups are known as modules, subprograms or subroutines. Modular programming based on the understanding of top-down analysis. It discourages jumps using 'goto' statements in the program, which often makes the program flow non-traceable. Jumps are prohibited and modular format is encouraged in structured programming.
- **Structured Coding** - In reference with top-down analysis, structured coding sub-divides the modules into further smaller units of code in the order of their execution. Structured programming uses control structure, which controls the flow of the program, whereas structured coding uses control structure to organize its instructions in definable patterns.

Functional Programming

- Functional programming provides means of computation as mathematical functions, which produces results irrespective of program state. This makes it possible to predict the behavior of the program.
- Functional programming uses the following concepts:
  - **First class and High-order functions** - These functions have capability to accept another function as argument or they return other functions as results.
  - **Pure functions** - These functions do not include destructive updates, that is, they do not affect any I/O or memory and if they are not in use, they can easily be removed without hampering the rest of the program.
  - **Recursion** - Recursion is a programming technique where a function calls itself and repeats the program code in it unless some pre-defined condition matches. Recursion is the way of creating loops in functional programming.

Functional Programming

- **Strict evaluation** - It is a method of evaluating the expression passed to a function as an argument. Functional programming has two types of evaluation methods, strict (eager) or non-strict (lazy). Strict evaluation always evaluates the expression before invoking the function. Non-strict evaluation does not evaluate the expression unless it is needed.
- **λ-calculus** - Most functional programming languages use λ-calculus as their type systems. λ-expressions are executed by evaluating them as they occur.

Programming style

- Programming style is set of coding rules followed by all the programmers to write the code. When multiple programmers work on the same software project, they frequently need to work with the program code written by some other developer. This becomes tedious or at times impossible, if all developers do not follow some standard programming style to code the program.
- An appropriate programming style includes using function and variable names relevant to the intended task, using well-placed indentation, commenting code for the convenience of reader and overall presentation of code. This makes the program code readable and understandable by all, which in turn makes debugging and error solving easier. Also, proper coding style helps ease the documentation and updation.

Coding Guidelines

- **Naming conventions** - This section defines how to name functions, variables, constants and global variables.
- **Indenting** - This is the space left at the beginning of line, usually 2-8 whitespace or single tab.
- **Whitespace** - It is generally omitted at the end of line.
- **Operators** - Defines the rules of writing mathematical, assignment and logical operators. For example, assignment operator ‘=’ should have space before and after it, as in “x = 2”.
- **Control Structures** - The rules of writing if-then-else, case-switch, while-until and for control flow statements solely and in nested fashion.

Coding Guidelines

- **Line length and wrapping** - Defines how many characters should be there in one line, mostly a line is 80 characters long. Wrapping defines how a line should be wrapped, if is too long.
- **Functions** - This defines how functions should be declared and invoked, with and without parameters.
- **Variables** - This mentions how variables of different data types are declared and defined.
- **Comments** - This is one of the important coding components, as the comments included in the code describe what the code actually does and all other associated descriptions. This section also helps creating help documentations for other developers.

Software Documentation

- Software documentation is an important part of software process. A well written document provides a great tool and means of information repository necessary to know about software process. Software documentation also provides information about how to use the product.
- A well-maintained documentation should involve the following documents:
  - **Requirement documentation** - This documentation works as key tool for software designer, developer and the test team to carry out their respective tasks. This document contains all the functional, non-functional and behavioral description of the intended software.
  - Source of this document can be previously stored data about the software, already running software at the client’s end, client’s interview, questionnaires and research. Generally it is stored in the form of spreadsheet or word processing document with the high-end software management team.
  - This documentation works as foundation for the software to be developed and is majorly used in verification and validation phases. Most test-cases are built directly from requirement documentation.

Software Documentation

- **Software Design documentation** - These documentations contain all the necessary information, which are needed to build the software. It contains: **(a)** High-level software architecture, **(b)** Software design details, **(c)** Data flow diagrams, **(d)** Database design
- These documents work as repository for developers to implement the software. Though these documents do not give any details on how to code the program, they give all necessary information that is required for coding and implementation.
- **Technical documentation** - These documentations are maintained by the developers and actual coders. These documents, as a whole, represent information about the code. While writing the code, the programmers also mention objective of the code, who wrote it, where will it be required, what it does and how it does, what other resources the code uses, etc.
- The technical documentation increases the understanding between various programmers working on the same code. It enhances re-use capability of the code. It makes debugging easy and traceable.
- There are various automated tools available and some comes with the programming language itself. For example java comes JavaDoc tool to generate technical documentation of code.
- **User documentation** - This documentation is different from all the above explained. All previous documentations are maintained to provide information about the software and its development process. But user documentation explains how the software product should work and how it should be used to get the desired results.
- These documentations may include, software installation procedures, how-to guides, user-guides, uninstallation method and special references to get more information like license updation etc.

Software Implementation Challenges

- There are some challenges faced by the development team while implementing the software. Some of them are mentioned below:
  - **Code-reuse** - Programming interfaces of present-day languages are very sophisticated and are equipped huge library functions. Still, to bring the cost down of end product, the organization management prefers to re-use the code, which was created earlier for some other software. There are huge issues faced by programmers for compatibility checks and deciding how much code to re-use.
  - **Version Management** - Every time a new software is issued to the customer, developers have to maintain version and configuration related documentation. This documentation needs to be highly accurate and available on time.

Software Implementation Challenges

- There are some challenges faced by the development team while implementing the software. Some of them are mentioned below:
  - **Target-Host** - The software program, which is being developed in the organization, needs to be designed for host machines at the customers end. But at times, it is impossible to design a software that works on the target machines.
  - **Version Management** - Every time a new software is issued to the customer, developers have to maintain version and configuration related documentation. This documentation needs to be highly accurate and available on time.

<u>Thank you !!!</u>

Introduction to Software Testing for Quality, Dependability, and Availability

Development Testing, Test-Driven Development (TDD), Reliability, and Performance Testing

Introduction to Software Testing

What is Software Testing?

- Software testing ensures that the software behaves as expected and meets quality standards. It aims to detect defects early and guarantee software reliability, performance, and availability.

Why Testing is Crucial:

- **Assuring Software Quality**: Tests verify that software meets user expectations and functional requirements.
- **Dependability and Availability**: Testing ensures the software can handle real-world conditions, with minimal downtime and robust performance.

Introduction to Software Testing

Key Concepts:

- **Quality**: Functionality, usability, and performance.
- **Dependability**: Reliability, availability, and security.
- **Availability**: Ensuring the system is available for use when needed (no downtime).

Software Testing Lifecycle

Phases:

- **Requirement Analysis**: Understanding user requirements, including reliability, performance, and availability needs.
- **Test Planning**: Including strategies for reliability testing, stress testing, and availability verification.
- **Test Design**: Writing test cases that account for the software's dependability under normal and extreme conditions.
- **Test Execution**: Running functional, non-functional, and reliability tests.
- **Defect Reporting**: Logging defects with a focus on system availability and performance.
- **Test Closure**: Summarizing test results with emphasis on reliability and availability metrics.

Types of Software Testing

- **Manual Testing**: Conducting tests manually, useful for exploratory and usability testing.
- **Automated Testing**: Efficient for regression, performance, and load testing.

Non-Functional Testing (Key for Dependability and Availability):

- **Performance Testing**: Measures the responsiveness and speed of the application under normal and peak load.
- **Stress Testing**: Evaluates the software's behavior under extreme conditions, such as heavy traffic or limited resources.
- **Reliability Testing**: Ensures the software consistently performs well over time, including during failures.
- **Availability Testing**: Ensures the system is available without downtime or disruptions

Development Testing for Quality Assurance

Unit Testing: Ensures individual components perform as expected, critical for building reliable software.

Integration Testing: Verifies interactions between components, ensuring dependability when modules integrate.

Reliability and Availability Considerations in Development:

•**Mocking External Services**: Mock external systems (e.g., databases, APIs) to test how components behave when they fail or become unavailable.

•**Dependency Injection**: Helps test components in isolation and ensures reliability under different configurations

Software Testing Lifecycle

Phases:

- **Requirement Analysis**: Understanding user requirements, including reliability, performance, and availability needs.
- **Test Planning**: Including strategies for reliability testing, stress testing, and availability verification.
- **Test Design**: Writing test cases that account for the software's dependability under normal and extreme conditions.
- **Test Execution**: Running functional, non-functional, and reliability tests.
- **Defect Reporting**: Logging defects with a focus on system availability and performance.
- **Test Closure**: Summarizing test results with emphasis on reliability and availability metrics.

Test-Driven Development (TDD) for Reliable Software

What is TDD?: A practice that improves software reliability by writing tests before coding.

TDD Cycle:

1. Write a test.
2. Run the test (it fails).
3. Write code to make the test pass.
4. Refactor.

TDD for Reliability:

• TDD helps identify defects early and ensures that each component meets the expected behavior, reducing the risk of failures later in production.

Performance, Stress, and Reliability Testing

Performance Testing: Measures how the software performs under expected load.

Stress Testing: Tests the software beyond its limits to see how it fails or recovers.

- **Example**: Increasing the load on an e-commerce website during a Black Friday sale to see how it handles a high volume of users.

Reliability Testing: Ensures the software remains stable and performs consistently over time.

- **Key Metrics**: Mean Time Between Failures (MTBF), Mean Time to Repair (MTTR).

Availability Testing

What is Availability Testing?: Ensures that the system is available when required, without downtime or disruptions.

Types of Availability Testing:

Failover Testing: Verifies that the system can switch to backup resources if a primary resource fails.

Disaster Recovery Testing: Ensures the system can recover quickly after a catastrophic failure.

Load Balancing Testing: Ensures that traffic is distributed evenly across servers for continuous availability.

Availability Testing

Cloud Application Availability:

How cloud systems use redundancy to ensure high availability, and how testing can ensure these measures are working.

Best Practices for Assuring Software Quality

1. **Test Early, Test Often**: Start testing early in development to catch issues early and ensure reliability.
2. **Automate Regression Testing**: Use automation tools to run tests frequently and consistently.
3. **Monitor Post-Release**: Implement monitoring tools to ensure availability and performance in production (e.g., APM tools like New Relic).
4. **Simulate Real-World Scenarios**: Include reliability, stress, and performance tests to simulate how the software behaves in the real world.

Designing a Test Plan

Objective: Design a testing plan for a fictional e-commerce system, ensuring the system is tested for reliability, performance, and availability.

Key Areas to Cover:

1. Unit testing and integration testing.
2. Load and stress testing for performance.
3. Availability and failover testing.
4. Disaster recovery and recovery time tests.

Conclusion and Takeaways

Key Points:

1. Assuring software quality is not just about functionality but also performance, reliability, and availability.
2. Techniques like performance testing, stress testing, and failover testing are essential for ensuring that the software is dependable and available.
3. Testing early and often, and automating key tests, are best practices for maintaining high-quality, reliable systems.

Thank You

Ensuring Software Quality: Attributes, Assurance, and Metrics

Understanding the Core Principles of Quality in Software Development

Introduction to Software Quality

- **What is Software Quality?**

- Software quality refers to the degree to which a software system satisfies customer requirements, works reliably, and performs efficiently. It encompasses various dimensions of a system's functionality and performance.

- **Why is Software Quality Important?**

- Good quality software is user-friendly, dependable, and performs optimally, resulting in greater customer satisfaction, fewer defects, and long-term sustainability.

- Poor quality software leads to bugs, security vulnerabilities, increased maintenance costs, and decreased user trust.

Quality Attributes of Software

- **What Are Quality Attributes?**

- Quality attributes are characteristics that define the quality of software. They include:

1. Functionality: The ability of software to perform the required functions correctly.
2. Reliability: The ability of software to maintain performance under expected conditions over time.
3. Usability: How easy and intuitive the software is for users.
4. Efficiency: How well the software performs with respect to resource usage, such as memory and CPU.
5. Maintainability: How easily the software can be modified, updated, or extended.
6. Portability: The ability of software to be transferred or run in different environments (e.g., operating systems, hardware).

Quality Attributes of Software

- **Additional Attributes**:

- Availability: The degree to which the system is operational and accessible when needed.

- Scalability: The software's ability to handle growth, such as increasing load or data.

- Security: Protection against unauthorized access and data breaches.

Key Focus Areas for Software Quality Assurance (SQA)

- **What is Software Quality Assurance (SQA)?**

SQA is a set of activities aimed at ensuring that software meets the required quality standards throughout its lifecycle. It is a proactive process that focuses on preventing defects rather than finding them after they occur.

Key Focus Areas for Software Quality Assurance (SQA)

SQA Processes

1. Planning: Defining quality goals, creating a testing strategy, and determining required resources.

2. Standards and Guidelines: Establishing industry or company-specific standards for development, design, and testing.

3. Reviews and Audits: Regularly reviewing requirements, design, and code to identify potential quality issues.

4. Testing: Implementing various testing strategies to ensure quality attributes are met (functional testing, non-functional testing, regression testing).

5. Defect Management: Tracking, reporting, and resolving defects throughout the development process.

The Role of SQA in Different Phases

Phase 1: Requirements and Design

SQA starts with early involvement in requirements gathering to ensure that functional and non-functional requirements (like reliability, security, performance) are well-defined.

Phase 2: Implementation

Code reviews and static analysis tools help catch defects early, ensuring the software adheres to best coding practices and quality standards.

The Role of SQA in Different Phases

Phase 3: Testing

Focus on continuous testing (unit testing, integration testing, performance testing) and the role of "automated testing" in ensuring quality throughout development.

Phase 4: Deployment and Maintenance

Post-deployment monitoring to assess real-world quality (e.g., uptime, performance, user feedback).

Metrics for Measuring Software Quality

What Are Product Metrics?

Product metrics are quantitative measures used to assess various aspects of software quality. These metrics help identify areas for improvement, measure progress, and track overall product health.

Types of Product Metrics

1. Code Quality Metrics

Cyclomatic Complexity: Measures code complexity to identify difficult-to-maintain sections of the code.

Code Churn: Tracks how often code is changed; excessive churn might indicate instability.

Code Coverage: Percentage of code covered by automated tests. Higher coverage generally correlates with fewer defects.

Metrics for Measuring Software Quality

2. **Defect Metrics**

    Defect Density: Number of defects per unit of software (e.g., per 1,000 lines of code).

    Defect Severity: Classifies defects based on their impact (critical, major, minor).

    Mean Time to Failure (MTTF): Measures the average time the system operates before it experiences a failure.

    Defect Rejection Ratio: Percentage of reported defects that are rejected as non-issues.

3. **Performance Metrics**

    Response Time: The time it takes for the software to respond to user input or requests.

    Throughput: The amount of work the system can process in a given time frame.

    Resource Utilization: Tracks CPU and memory usage during runtime.

Linking Metrics to Quality Attributes

Mapping Metrics to Attributes

- Reliability: Measured by MTTF, defect density, and test coverage.
- Usability: Measured by user satisfaction surveys, usability testing results.
- Performance: Measured by response time, throughput, and resource utilization.
- Maintainability: Measured by cyclomatic complexity, code churn, and code review outcomes.

- Balancing Metrics:
- Focusing on one quality attribute might affect others (e.g., improving performance might increase complexity and reduce maintainability).

Software Quality Assurance Frameworks

Common SQA Frameworks:

- CMMI (Capability Maturity Model Integration): A process improvement framework focusing on continuous improvement.
- ISO 9001: A set of standards for quality management and continuous improvement.
- Six Sigma: Focuses on reducing defects and variation to improve process quality.

Integrating SQA in Agile:

- Agile methodologies incorporate SQA activities (e.g., continuous integration, automated testing, regular retrospectives).

Best Practices for Ensuring Software Quality

- Early Involvement of Quality Assurance: Begin quality checks during requirements gathering, not after development.
- Automation: Invest in automated testing and continuous integration to maintain high software quality throughout the lifecycle.
- Regular Reviews and Refactoring: Continuously improve code quality with peer reviews and refactoring to reduce defects and maintain maintainability.
- User-Centric Testing: Focus on usability testing and feedback loops to ensure software meets user expectations.

Conclusion and Key Takeaways

- Software quality is defined by multiple attributes (functionality, reliability, performance, usability, etc.), and it must be assured throughout the lifecycle using proactive SQA techniques.
- Metrics provide objective data to measure and manage software quality, helping to identify areas for improvement.

Thank You

System Dependability and Reliability Engineering in Software Engineering

Exploring the Foundations of Building Trustworthy Systems

Introduction

What is Dependability?

- Dependability is the ability of a system to deliver its services dependably and correctly, even when there are internal or external disturbances.
- Dependability is essential for building trust in systems, ensuring their continuity, and making sure they function properly across their lifecycle.

Key Dependability Dimensions:

- Reliability
- Availability
- Safety
- Confidentiality
- Integrity
- Maintainability

Introduction

Why Focus on Dependability in Software?

- Software failures can have severe consequences (e.g., system crashes, data loss, financial loss, safety hazards).
- In critical systems (e.g., healthcare, aerospace, autonomous vehicles), dependability is non-negotiable.

Dependability Properties

- Reliability - The probability that a system will function without failure for a given period of time under specified conditions.
- Key Metrics
  - MTTF (Mean Time to Failure): Average time the system operates before failing.
  - MTBF (Mean Time Between Failures): Average time between two consecutive failures.
  - Failure Rate: How often the system fails (often expressed in failures per hour or failures per year).
- Reliability vs. Availability: While reliability is about a system performing its intended function without failure, availability is concerned with whether the system is ready for use when needed.
- Availability - The proportion of time a system is operational and accessible when required. (Uptime / (Uptime + Downtime)

Dependability Properties

- Key Factors Affecting Availability:
  - Uptime** (when the system is running as expected).
  - Downtime** (when the system is non-functional due to failures, maintenance, etc.).
- Types of Availability:
  - Hardware Availability: Ensuring hardware is operational.
  - Software Availability: Ensuring software services are accessible to users.
- Safety - The system should not cause harm to people, the environment, or other systems, even when faults occur.
- Confidentiality and Integrity
  - Confidentiality: Ensuring that system data is protected from unauthorized access.
  - Integrity: Ensuring that data is accurate, consistent, and uncorrupted during operations.

Dependability Properties

- Maintainability - The ease with which a system can be modified or repaired to correct faults or accommodate new functionality.
- Diagnostic capabilities, modular design, and clear documentation all contribute to maintainability.

Reliability Engineering in Software

- What is Reliability Engineering?
- Reliability engineering involves identifying potential risks and implementing strategies to ensure that a system remains reliable over time.
- It involves creating and evaluating metrics, models, and methods to predict, measure, and improve reliability.
- Key Methods for Improving Reliability:
- Fault Prevention: Minimizing the possibility of faults through careful design, rigorous testing, and preventive measures.
  - Code Reviews and Static Analysis tools help detect defects early.
  - Formal Methods (e.g., model checking) ensure that systems behave correctly under all conditions.

Reliability Engineering in Software

- Fault Tolerance: Designing systems that can continue functioning even when one or more components fail.
  - Techniques like redundancy (e.g., duplicate servers, backup power supplies) or error correction (e.g., RAID storage for data redundancy) ensure the system remains operational despite individual failures.

- Fault Detection and Diagnosis: Mechanisms to detect, log, and recover from failures.
  - Health Monitoring Systems: Monitoring system states and detecting anomalies in real time.
  - Automated System Recovery: Failover mechanisms or automatic restarts.

Reliability Engineering in Software

- Reliability Testing:
  - Stress Testing: Pushing the system beyond its expected limits to see how it behaves under extreme conditions.
  - Failure Mode and Effect Analysis (FMEA): Analyzing potential failure points in a system and their consequences.
  - Fault Injection: Deliberately introducing faults in a controlled environment to observe how the system reacts.

Availability and Redundancy Engineering

- High Availability (HA) Systems:
  - Aim for minimal downtime, typically through redundancy, failover systems, and continuous monitoring.
  - Example: An e-commerce website uses load balancing across multiple data centers to ensure availability even if one server fails.
- Availability Modeling:
  - Availability Models help quantify how system configurations impact availability. Common models include:
  - Series-Parallel System Models: Use reliability block diagrams (RBD) to visualize system redundancy.
  - Markov Models: Used to model different system states (e.g., operational, failed, under maintenance) and compute availability over time.
- Redundancy Techniques
  - Active-Standby Redundancy:
  - One component is active, and another component is on standby, ready to take over if the active one fails.
  - Example: Web servers or database systems.
  - N + 1 Redundancy:
  - One extra component (N + 1) is added to a system to ensure that if one component fails, there is always a backup.
- Load Balancing: - Distributing workloads across multiple servers or systems to improve performance and ensure availability.

Dependability Engineering Life Cycle

Specification:

- Clearly define the dependability requirements (e.g., what level of availability, reliability, safety).
- Example: A financial trading system might require 99.999% availability to avoid losing transactions.

Design:

- Implement fault-tolerant architecture (e.g., using redundancy or replication techniques).
- Choose reliable components (e.g., selecting hardware and software with proven reliability).

Implementation:

- Ensure robustness in the code with practices like defensive programming, automated unit tests, and integration testing.

Dependability Engineering Life Cycle

Verification and Validation:

- Test the system under real-world and extreme conditions using methods like stress testing, simulation, and fault injection.

Operation and Maintenance:

- Continuously monitor system performance using health monitoring tools and incident management systems.
- Implement automated recovery mechanisms and regular updates to patch vulnerabilities.

Case Study: Reliability in Critical Systems

Example 1: NASA's Mars Rover

- Challenge: The Mars Rover, operating millions of kilometers away, must function reliably in a harsh environment with no direct human intervention.
- Approach:
- Redundant systems and components ensure critical functions continue in the event of failure.
- Extensive pre-launch testing to simulate real-world conditions and uncover potential failure modes.
- Autonomous fault detection and diagnostic systems to address issues remotely.

Example 2: Air Traffic Control Systems

- Challenge: Air traffic control systems must maintain extremely high availability and safety, as failures could lead to catastrophic consequences.
- Approach:
- Dual-redundant systems ensure that if one system fails, the other takes over immediately.
- Continuous health monitoring of critical systems.
- Regular disaster recovery drills to ensure readiness in case of system failure.

Tools and Techniques for Dependability Engineering

Failure Mode and Effect Analysis (FMEA)

- A structured approach to identifying all potential failure modes in a system, their causes, and their effects on the overall system.
- Example: In a software application, FMEA could identify potential failure points (e.g., a database crash) and mitigate risks by implementing backup systems.

Markov Chains and Models

- Used to model the state transitions of a system, especially for predicting reliability and availability.
- Example: A simple Markov chain could model the transition between system states (operational, failure, repair), helping predict how long a system is expected to be operational.

Tools and Techniques for Dependability Engineering

Reliability Block Diagrams (RBD)

- A graphical tool used to visualize and analyze system reliability.
- Example: In a redundant power supply system, an RBD could show how different components (e.g., power sources, switches, backup batteries) contribute to the overall system's reliability.

Automated Monitoring Tools

- Tools like **Prometheus** or **Nagios** help continuously monitor system performance and detect failures in real time.

Conclusion

Key Takeaways:

- Dependability in software engineering ensures that systems are reliable, available, and safe.
- Engineering dependability requires understanding failure modes and implementing strategies like redundancy, monitoring, and fault-tolerance.
- Reliability, availability, and safety are interdependent and require a comprehensive approach from design to operation.

- **Final Thought**: Building dependable systems is a continuous process that requires attention to design, testing, monitoring, and maintenance.

Thank You

Project Planning

- **Effort estimation -** The managers estimate efforts in terms of personnel requirement and man-hour required to produce the software. For effort estimation software size should be known. This can either be derived by managers' experience, organization's historical data or software size can be converted into efforts by using some standard formulae.
- **Time estimation -** Once size and efforts are estimated, the time required to produce the software can be estimated. Efforts required is segregated into sub categories as per the requirement specifications and interdependency of various components of software. Software tasks are divided into smaller tasks, activities or events by Work Breakthrough Structure (WBS). The tasks are scheduled on day-to-day basis or in calendar months.
- The sum of time required to complete all tasks in hours or days is the total time invested to complete the project.

Project Communication Management

- Effective communication plays vital role in the success of a project. It bridges gaps between client and the organization, among the team members as well as other stake holders in the project such as hardware suppliers.
- Communication can be oral or written.

Project Communication Management

Communication management process may have the following steps:

- **Planning** - This step includes the identifications of all the stakeholders in the project and the mode of communication among them. It also considers if any additional communication facilities are required.
- **Sharing** - After determining various aspects of planning, manager focuses on sharing correct information with the correct person on correct time. This keeps every one involved the project up to date with project progress and its status.
- **Feedback** - Project managers use various measures and feedback mechanism and create status and performance reports. This mechanism ensures that input from various stakeholders is coming to the project manager as their feedback.
- **Closure** - At the end of each major event, end of a phase of SDLC or end of the project itself, administrative closure is formally announced to update every stakeholder by sending email, by distributing a hardcopy of document or by other mean of effective communication.

Configuration Management

- Configuration management is a process of tracking and controlling the changes in software in terms of the requirements, design, functions and development of the product.
- IEEE defines it as “the process of identifying and defining the items in the system, controlling the change of these items throughout their life cycle, recording and reporting the status of items and change requests, and verifying the completeness and correctness of items”.
- Generally, once the SRS is finalized there is less chance of requirement of changes from user. If they occur, the changes are addressed only with prior approval of higher management, as there is a possibility of cost and time overrun.

- A phase of SDLC is assumed over if it baselined, i.e. baseline is a measurement that defines completeness of a phase. A phase is baselined when all activities pertaining to it are finished and well documented. If it was not the final phase, its output would be used in next immediate phase.

- Configuration management is a discipline of organization administration, which takes care of occurrence of any change (process, requirement, technological, strategical etc.) after a phase is baselined. CM keeps check on any changes done in software.