Unit 5: System Modeling (6 Hrs.)

5.1 Introduction to System Modeling

System modeling is the process of creating abstract representations of a system, with each model providing a different perspective or view. These models help in understanding, designing, and documenting a system's structure, behavior, and interactions. While system modeling is often associated with graphical notations like the **Unified Modeling Language (UML)**, it can also involve formal mathematical models for precise system specifications.

Purposes of System Modeling

Models serve various roles throughout the system development lifecycle:

1. **During Requirements Engineering** – Models of an existing system help clarify its functionality and identify strengths and weaknesses. Models of a proposed system aid in explaining new requirements to stakeholders.
2. **During Design** – Engineers use models to discuss design proposals and document system architecture for implementation.
3. **After Implementation** – Models serve as documentation, explaining the system's structure and operation.
4. **Model-Driven Engineering (MDE)** – In some cases, system models can automatically generate partial or complete implementations.

System modeling mainly helps in identifying and validating the requirements of the new system by fining scope and limitation of the existing system.

UML as a Modeling Standard

The Unified Modeling Language (UML) is a widely adopted standard for object-oriented modeling. While UML includes 13 diagram types, five are most essential:

1. **Activity Diagrams** – Show process flows and data processing activities.
2. **Use Case Diagrams** – Illustrate interactions between the system and external actors.
3. **Sequence Diagrams** – Depict interactions between actors and system components over time.
4. **Class Diagrams** – Represent system classes and their relationships.
5. **State Diagrams** – Model system behavior in response to events.

5.2 Context Models

Context models are used in the early stages of system specification to define the **system boundaries**—what is included within the system and what lies outside in its environment. Establishing these boundaries helps clarify the system's scope, dependencies, and interactions with external entities. Context models also help to reduce the costs and the time needed for understanding the system requirements and design.

Representing Context with Models

A **context model** visually represents the system and its interactions with external entities.

Example: The Mentcare System

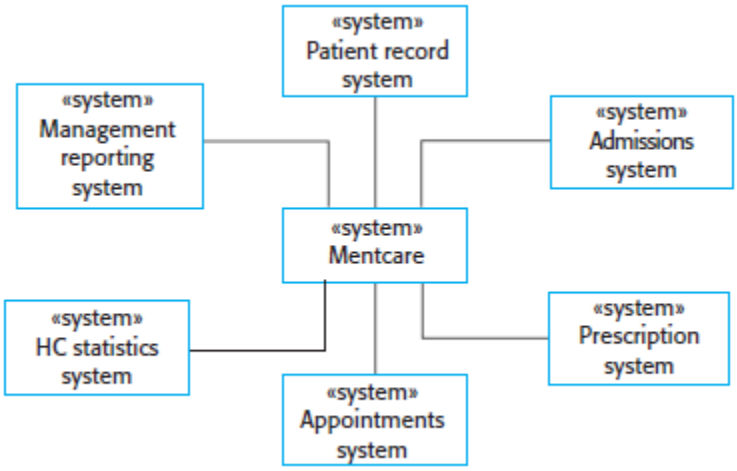

Figure 5.1: The context of the Mentcare system

The model shows **Mentcare** and its connected systems:

- **Patient Record System** (shares patient data).
- **Appointments System** (manages scheduling).
- **Admissions System** (handles hospital intake).
- **Prescription System** (generates medication orders).
- **Management Reporting & HC Statistics Systems** (support decision-making and research).

Limitations of Simple Context Models

While useful, context models do not show:

- **Nature of relationships** (data flow, dependencies, network connections).
- **Physical distribution** (co-located vs. remote systems).
- **Detailed interactions** (requiring additional models like **UML activity diagrams**).

5.3 Interaction Models

Interaction models are essential for understanding how different entities communicate within and around a system. These models help capture:

- **User interactions** (inputs/outputs with human users)
- **System-to-system interactions** (communication between software systems)
- **Component interactions** (internal exchanges between system modules)

Approaches to Interaction Modeling

Two primary methods are used to model interactions:

5.3.1 Use Case Modeling

Originally developed by Ivar Jacobsen, use cases describe **discrete tasks** where external actors interact with the system.

Key Elements of Use Cases

- **Actors**: Human users or external systems.
- **Use Case**: A specific interaction.
- **Relationships**: Lines connect actors to use cases.

Example: "Transfer Data" Use Case (Figure 5.4)

- **Actors**: Medical receptionist (initiator) and Patient Records System (PBS).
- **Description**: A receptionist transfers patient data (personal info or treatment summary) to a central database.
- **Key Details**:
    - **Stimulus**: User command from the receptionist.
    - **Response**: Confirmation of PBS update.
    - **Security**: Requires proper access permissions.

Use Case Diagrams

- Provide a **high-level overview** of interactions.

- Can be **composite** (showing multiple related use cases) or **focused** (specific actor interactions).
- Often supplemented with **detailed descriptions** (text, tables, or sequence diagrams).

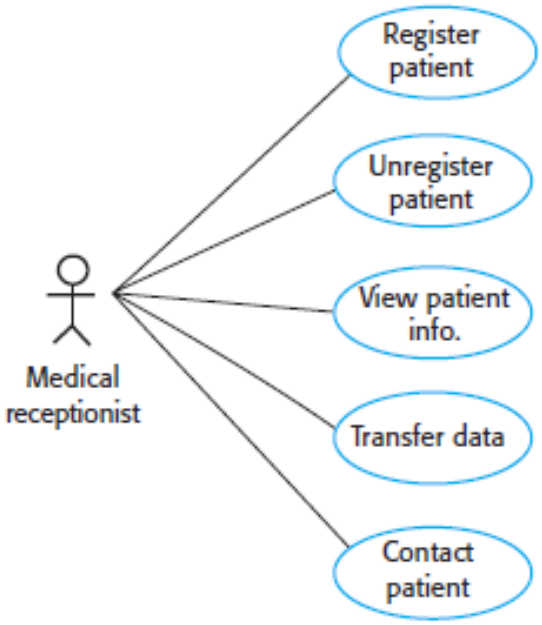

Figure 5.2: Use cases involving the role of medical receptionist

Limitations:

- Do not show the **sequence** of interactions.
- May become cluttered if too many use cases are included.

5.3.2 Sequence Diagrams

A sequence diagram is a type of interaction diagram that shows how objects or components in a system interact in a time-ordered sequence to perform a specific function or scenario (often tied to a use case). While use cases provide a **static view**, sequence diagrams model the **dynamic flow** of interactions over time.

Features of sequence diagrams:

- Show **message exchanges** between components/actors.
- Illustrate **order of operations** and **parallel processes**.
- Useful for **detailed system design** and **performance analysis**.

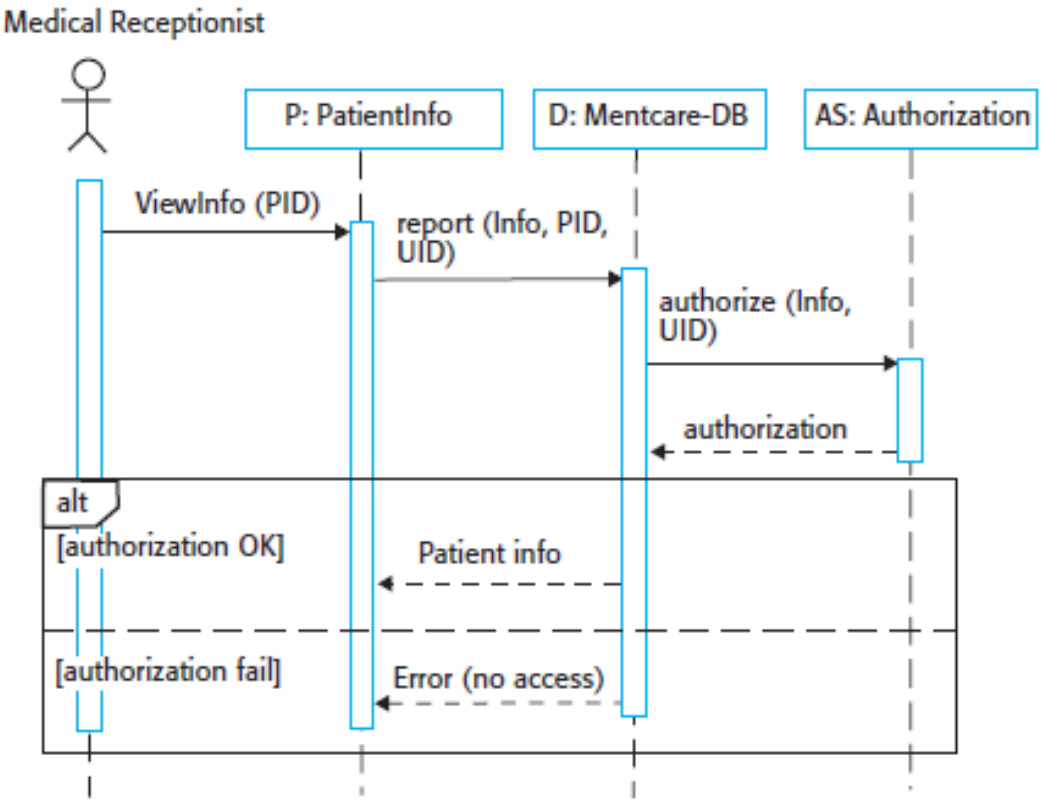

Figure 5.3: sequence diagram for view patient information

Figure 5.3 is an example of a sequence diagram that illustrates the view patient information. This sequence diagram illustrates the interactions between a **Medical Receptionist**, a **Patient Information (P) system**, a **Database (D)**, and an **Authorization System (AS)** when accessing patient records.

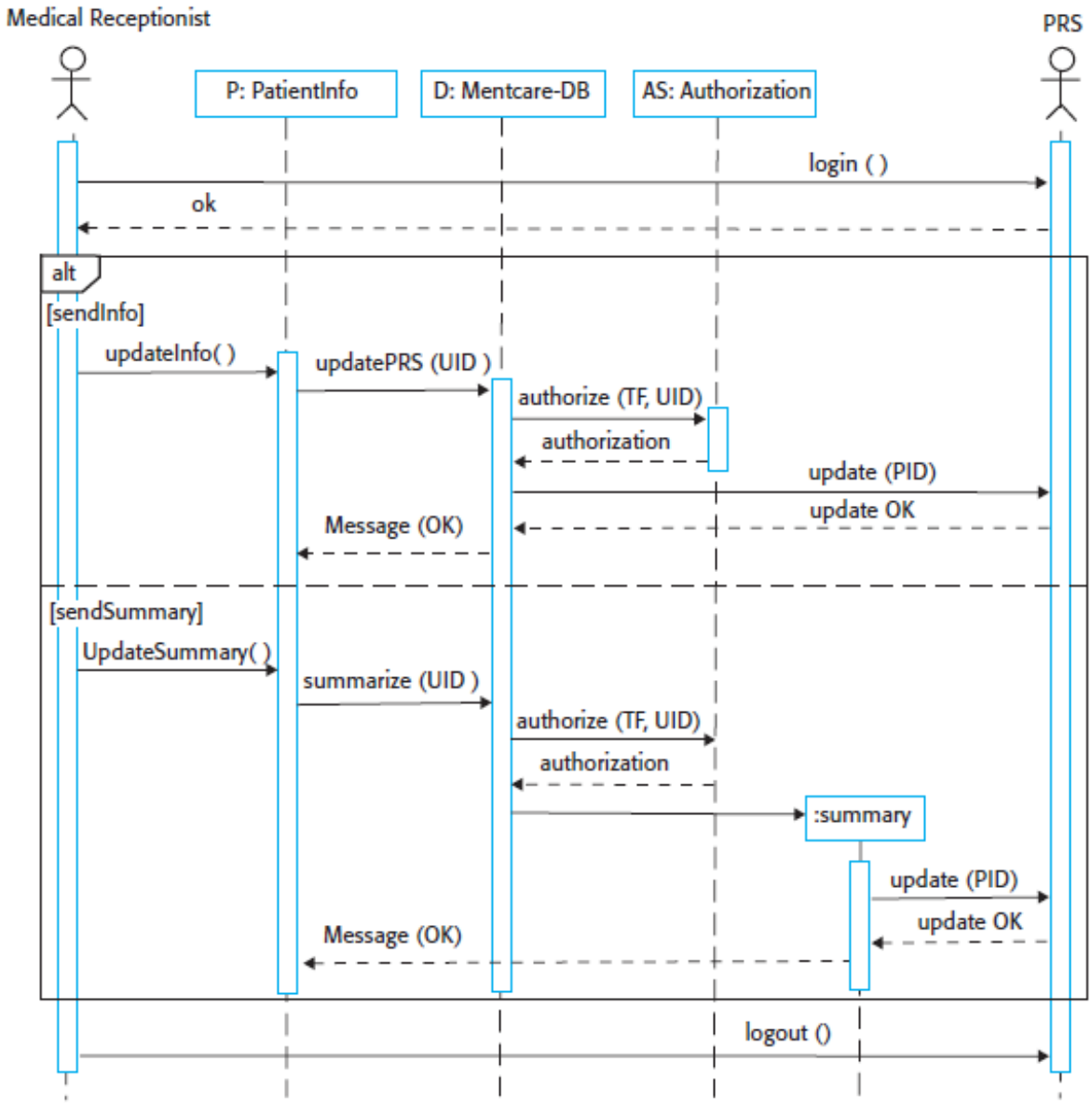

Figure 5.4: Sequence diagram for transfer data

Figure 5.4 shows a more complex workflow for data transfer:

1. **Receptionist** logs into the **Patient Records System (PRS)**.
2. **Two Transfer Options** (via alt fragment):
    - **Option 1**: Direct update via updateInfo() → updatePRS(UID).
    - **Option 2**: Summary creation via summarize(UID) → sendSummary().
3. **Authorization** is checked for both paths (authorize(TF, UID)).
4. **PRS** confirms completion with Message (OK).

When to Use Sequence Diagrams

- To clarify **complex workflows** (e.g., the "Involuntary Detention" process in Mentcare).
- To validate **system architecture** (e.g., ensuring components communicate efficiently).

Complementarity with Use Cases

- Use cases define **what** interactions occur.
- Sequence diagrams explain **how** they occur.

5.4 Structural Models

Structural models represent the **static organization** of a system's components (e.g., classes, objects, databases) and their relationships. They are essential for:

- **Designing system architecture** (e.g., class hierarchies, database schemas).
- **Clarifying relationships** between entities (e.g., patients, doctors, consultations).
- **Supporting reuse and modularity** (e.g., inheritance, aggregation).

Two key types of structural models are:

1. **Static Models**: Show system design at rest (e.g., class diagrams).
2. **Dynamic Models**: Show runtime interactions (e.g., sequence diagrams).

5.4.1 Class Diagrams

Class diagrams are the backbone of structural modeling, depicting:

- **Classes**: Represent entities (e.g., Patient, Doctor, Consultation).
- **Associations**: Relationships between classes (e.g., "diagnosed-with," "prescribes").
- **Multiplicities**: Cardinality constraints (e.g., 1..*, 1).

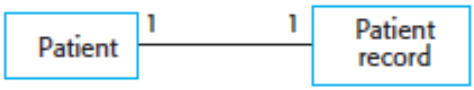

Figure 5.5: Classes and associations

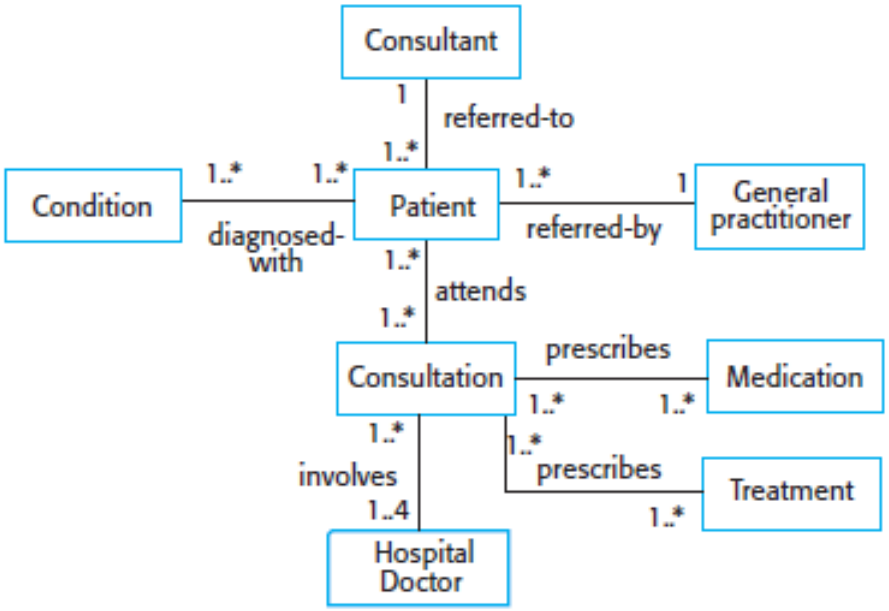

Figure 5.6: Classes and associations in the Ment care system

Features of Class diagrams

1. **Basic Notation**:
    - **Class**: Rectangle with name (e.g., Consultation).
    - **Attributes/Operations**: Listed in middle/bottom sections (e.g., Consultation class in **Figure 5.6** includes Date, Prescribe (), etc.).
2. **Associations**:
    - Named links

      e.g., referred-by between Patient and General Practitioner in **Figure 5.5**
    - Multiplicities:
        - 1: One-to-one

          e.g., Patient ↔ Patient Record in **Figure 5.5**
        - 1.. *: One-to-many

          e.g., Patient ↔ Condition in **Figure 5.6**
3. **Semantic Data Modeling**:
    - Similar to database schemas but with operations

      (e.g., ChangeAddress() in patient).

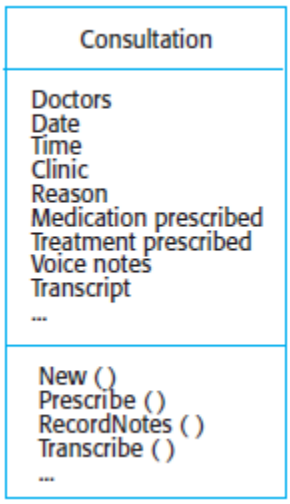

Figure 5.7: Consultation class

5.4.2 Generalization (Inheritance)

Generalization reduces complexity by grouping shared attributes/behaviors into superclasses.

Example: Doctor Hierarchy

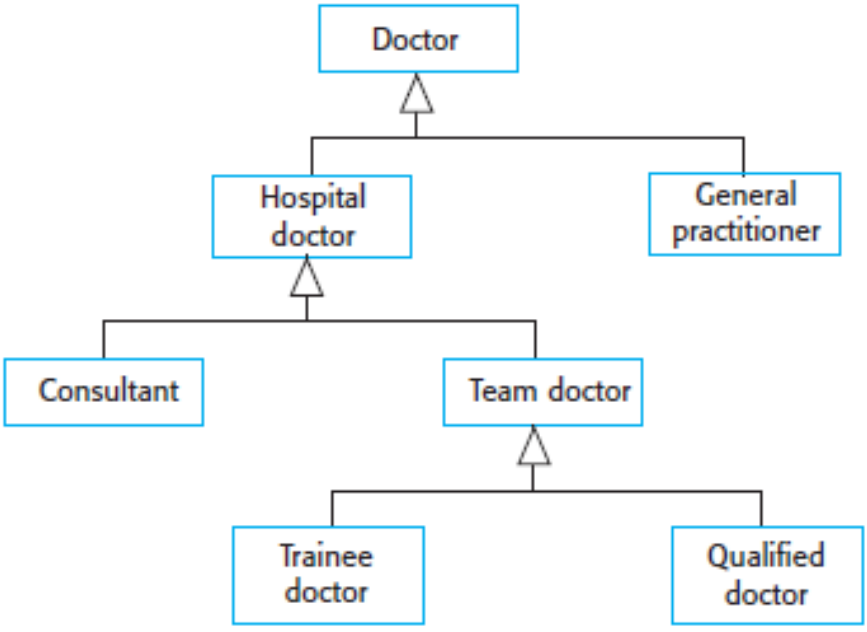

Figure 5.8: Generalization Hierarchy

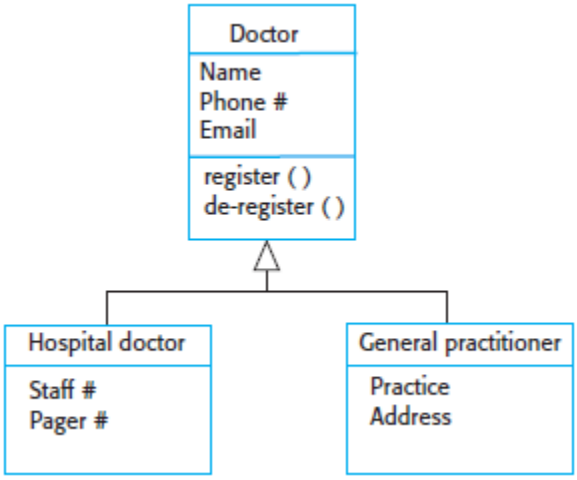

Figure 5.9: Generalization hierarchy with added detail

- **Superclass**: Doctor (common attributes: Name, Phone #).
- **Subclasses**:
  - HospitalDoctor (adds Staff #, Pager #).
  - GeneralPractitioner (adds Practice Address).
- **UML Notation**: Arrow from subclass to superclass (e.g., Consultant → Doctor).

Benefits:

- **Reusability**: Subclasses inherit superclass properties.
- **Maintainability**: Changes to Doctor propagate to all subclasses.

5.4.3 Aggregation

Aggregation models "whole-part" relationships where parts can exist independently.

Example: Patient Record

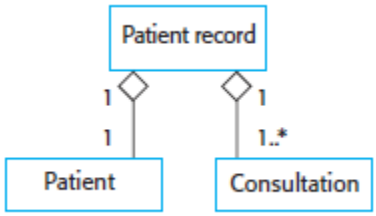

Figure 5.10: The aggregation association

- **Whole**: PatientRecord (aggregate).

- **Parts**: Patient (1) + Consultation (1..*).
- **UML Notation**: Diamond on the "whole" side (e.g., PatientRecord ◇─ Consultation).

Contrast with Composition:

- **Aggregation**: Parts can exist without the whole (e.g., Consultation can be archived separately).
- **Composition**: Parts die with the whole (e.g., Room cannot exist without a Building).

Practical Applications

1. **Database Design**: Class diagrams map to tables (e.g., Patient → SQL table).
2. **API Development**: Methods like Prescribe() define service endpoints.
3. **Security**: Authorization checks (e.g., Consultation access) can be modeled.

5.5 Behavioral Models

Behavioral models extend **structural models** (e.g., class diagrams) by defining **how the system behaves** in response to interactions, events, or state changes. Behavioral models are essential tools for understanding and designing the dynamic behavior of software systems as they execute. While structural models answer "What are the components?", behavioral models answer "How do they work together?".

Types of Behavioral Models

1. Data-Driven Modeling

Data-driven models focus on the sequence of actions involved in processing input data and generating corresponding outputs. These models are particularly useful for:

- Showing end-to-end processing in a system
- Illustrating the complete sequence from initial input to system response
- Understanding data flow through various processing steps

Examples:

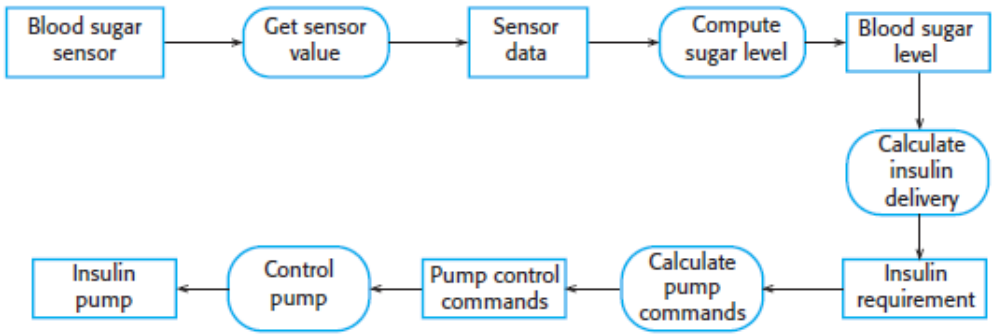

Figure 5.11: An activity model of the insulin pump's operation

Figure 5.11 shows an activity model of an insulin pump's operation, demonstrating how sensor data leads to insulin delivery

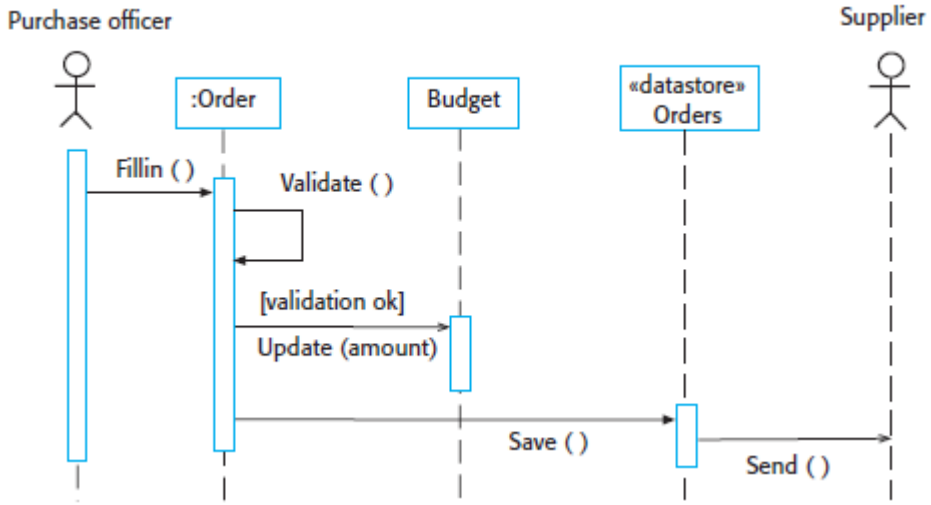

Figure 5.12: Order processing

Figure 5.12 presents a sequence model of order processing, highlighting the steps from order placement to sending to supplier

Note: Data-flow diagrams (DFDs) are traditional data-driven models that have evolved into UML activity diagrams. These are often more intuitive for non-technical stakeholders to understand compared to sequence diagrams, which engineers tend to prefer.

2. Event-Driven Modeling

Event-driven models show how systems respond to both external and internal events, based on the concept that systems have finite states and events trigger transitions between these states. These models are particularly valuable for:

- Real-time systems
- Systems with clearly defined operational states
- Safety-critical systems where state transitions must be carefully controlled

Examples:

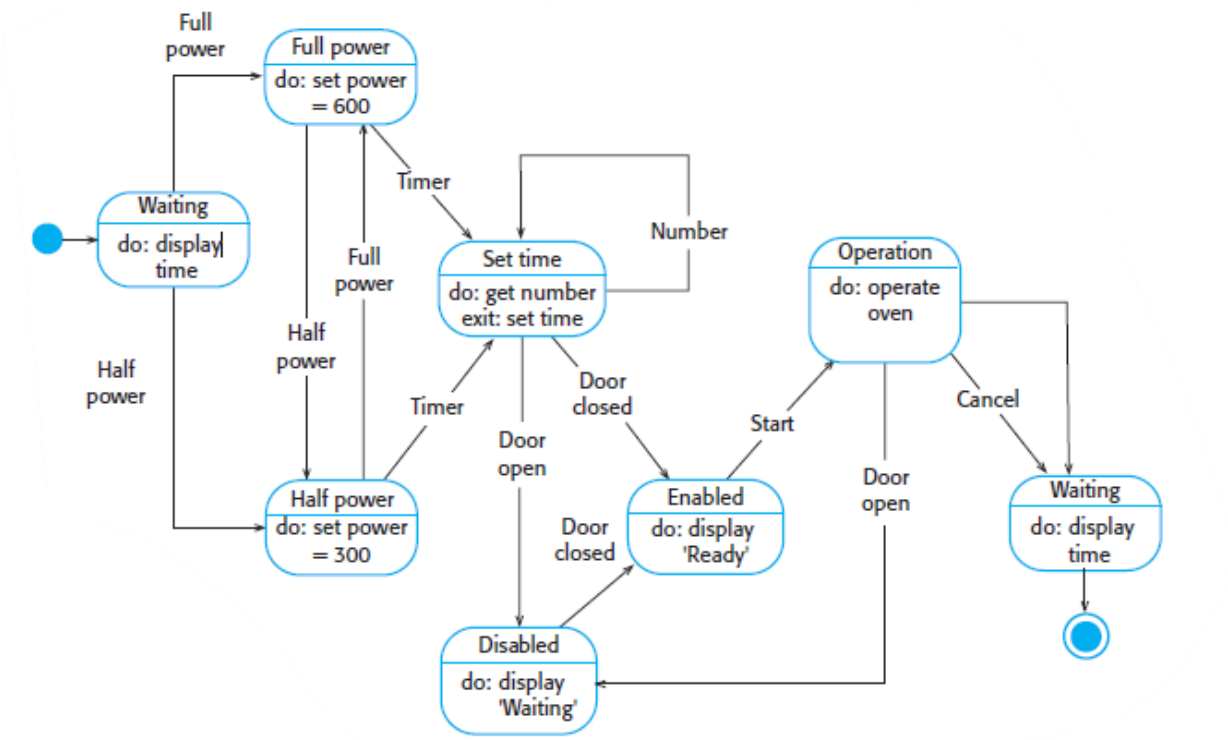

Figure 5.13: A state diagram of a microwave oven

- Figure 5.13 shows a state diagram of a microwave oven, illustrating states like "Waiting," "Full power," and "Operation"

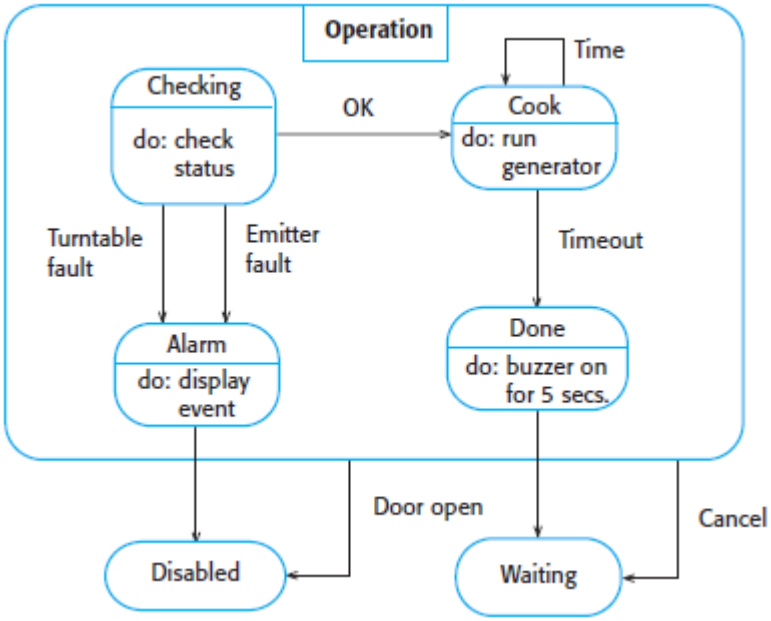

Figure 5.14: A state model of the operation state

- Figure 5.14 expands on the "Operation" superstate, revealing substates like "Checking," "Cook," and "Alarm"

The concept of "superstates" helps manage complexity by encapsulating multiple substates that can be expanded when more detail is needed.

Model-Driven Engineering (MDE)

MDE represents an approach where models rather than programs are the primary outputs of development:

- Programs are automatically generated from models
- Raises the level of abstraction, reducing focus on programming language details
- Evolved from Model-Driven Architecture (MDA) proposed by the Object Management Group
- While promising, adoption has been slow due to various practical challenges

5.6 Model-Driven Architecture

Model-Driven Architecture (MDA) is an innovative approach to software development that emphasizes models as the primary artifacts throughout the development process. This methodology, developed by the Object Management Group (OMG), represents a paradigm shift from traditional code-centric development to model-centric engineering.

The Three-Layered MDA Approach

MDA operates through three fundamental levels of abstraction:

1. **Computation Independent Model (CIM)**
    - Represents the highest level of abstraction
    - Focuses on domain concepts and business requirements rather than computation
    - Often called "domain models"
    - May include multiple CIMs for different system aspects (e.g., security CIM, patient record CIM)
    - Captures important domain abstractions and their relationships
2. **Platform Independent Model (PIM)**
    - Describes system operation without implementation details
    - Typically created using UML diagrams
    - Shows static system structure and dynamic behavior
    - Remains neutral regarding execution platforms
3. **Platform Specific Model (PSM)**
    - Transforms PIM into platform-specific implementations
    - May involve multiple layers (e.g., middleware-specific then database-specific)
    - One PSM generated for each target platform
    - Incorporates platform-specific patterns and rules

The Transformation Process

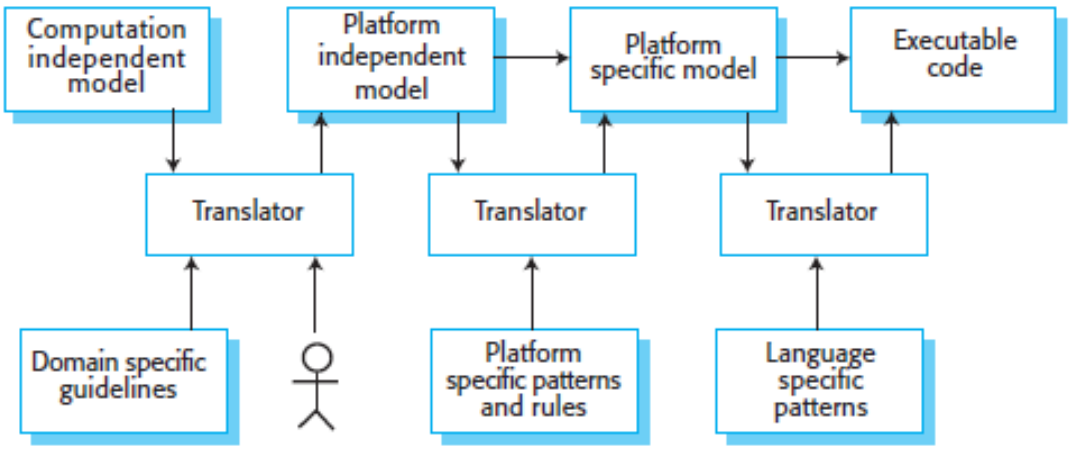

Figure 5.15: MDA transformations

The MDA workflow, as illustrated in Figure 5.15, involves automated transformations between these model layers:

1. **CIM to PIM Transformation**: The most challenging step, often requiring human intervention to map domain concepts
2. **PIM to PSM Transformation**: More straightforward, supported by commercial tools
3. **PSM to Executable Code**: Final transformation generating platform-specific code

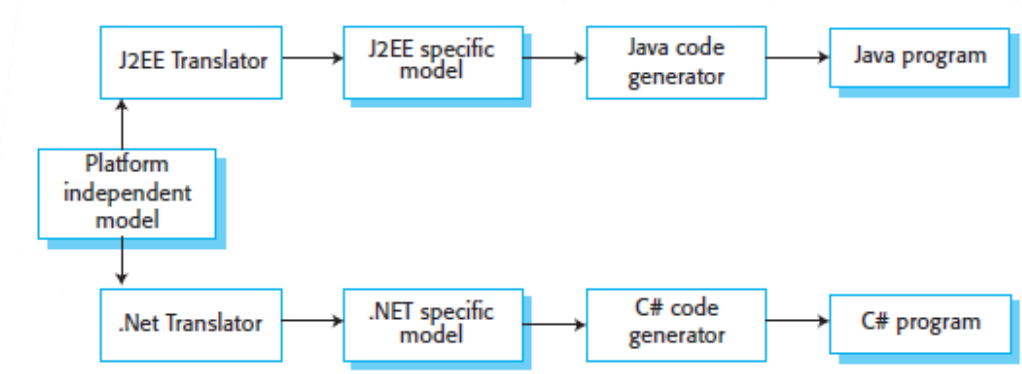

Figure 5.16: Multiple platform-specific models

Figure 5.16 demonstrates how a single PIM can generate multiple PSMs for different platforms (e.g., J2EE and .NET), enabling true platform independence.

Benefits of MDA

1. **Higher Abstraction Level**: Engineers focus on system design rather than implementation details
2. **Reduced Errors**: Fewer manual coding errors through automated generation
3. **Faster Development**: Accelerated design and implementation cycles
4. **Platform Independence**: Single model can target multiple platforms
5. **Improved Maintainability**: Easier adaptation to new technologies
6. **Enhanced Reusability**: Platform-independent models can be reused across projects

Challenges and Limitations

Despite its potential, MDA adoption has faced several obstacles:

1. **Translation Difficulties**:
    - Fully automated CIM to PIM transformation remains a research challenge
    - Concept mapping between different CIMs often requires human expertise

    - Specialized translators may be needed for company-specific environments
2. **Practical Concerns**:
    - Useful design abstractions may not align with implementation needs
    - Implementation is often not the most challenging part of development
    - Platform independence benefits are most relevant for long-lifecycle systems
    - Costs of MDA introduction may outweigh benefits for standard platform development
3. **Cultural Factors**:
    - Rise of agile methods has shifted focus away from extensive upfront modeling
    - Perceived conflict between MDA's model-heavy approach and agile principles