MODULE 2

SYSTEM MODELS, DESIGN AND IMPLEMENTATION

SYSTEM MODELS

→ 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 generally come to mean representing the system using some kind of graphical notation, which is now almost always based on notations in the Unified Modeling Language (UML).

→ Models are used during the requirements engineering process to help derive the requirements for a system, during the design process to describe the system to engineers implementing the system and after implementation to document the system's structure and operation.

2.1 Context Models

→ At an early stage in the specification of a system, it is necessary to decide on the system boundaries.

→ This involves working with system stakeholders to decide what functionality should be included in the system and what is provided by the system's environment.

→ A decision might be taken about a automated support for some business processes should be implemented but others should be manual processes or supported by different systems.

→ Possible overlaps must also be noted in functionality with existing systems and decide where new functionality should be implemented.

→ These decisions should be made early in the process to limit the system costs and the time needed for understanding the system requirements and design.

→ Fig 2.1 is a simple context model that shows the patient information system and the other systems in its environment.

→ From fig 2.1, it is seen that the MHC-PMS is connected to an appointments system and a more general patient record system with which it shares data.

→ The system is also connected to systems for management reporting and hospital bed allocation and a statistics system that collects information for research.

→ Finally, it makes use of a prescription system to generate prescriptions for patients' Medication.

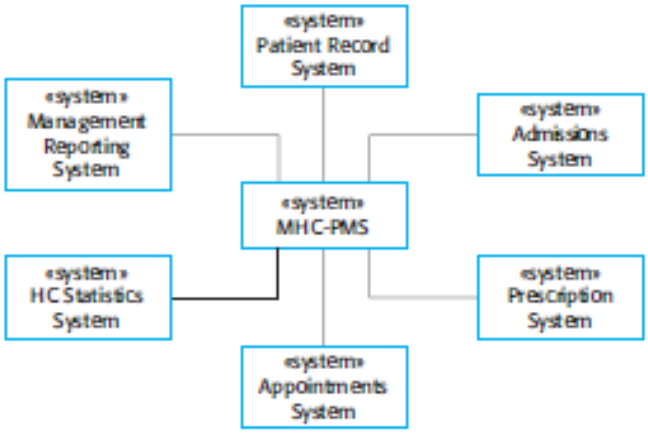

Fig 2.1: The context of the MHC-PMS

→ Fig 2.2 is a model of an important system process that shows the processes in which the MHC-PMS is used.

→ Fig 2.2 is a UML activity diagram.

→ Activity diagrams are intended to show the activities that make up a system process and the flow of control from one activity to another.

→ The start of a process is indicated by a filled circle; the end by a filled circle inside another circle.

→ Rectangles with round corners represent activities, that is, the specific sub-processes that must be carried out.

→ Objects can be included in activity charts.

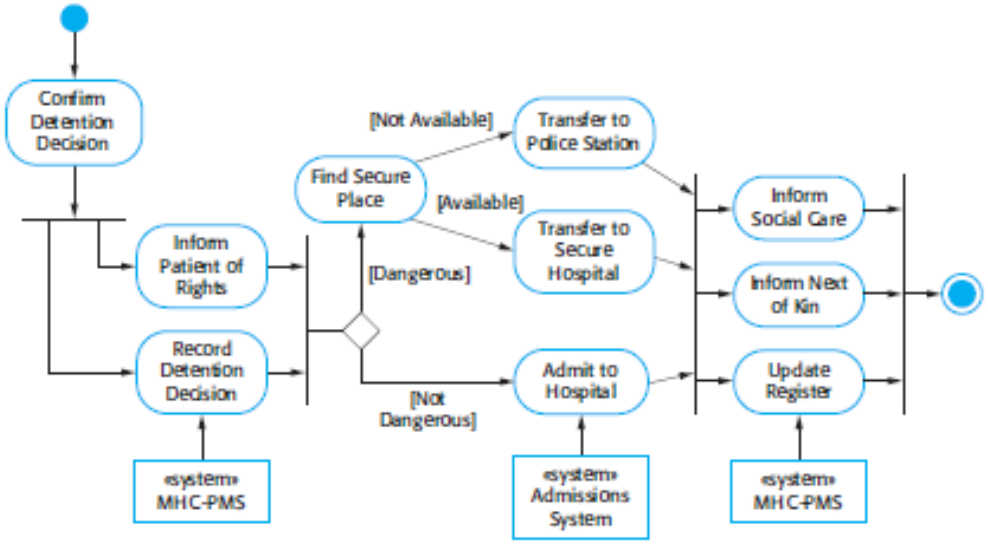

Fig 2.2: Process model of involuntary detention

→ In fig 2.2, it can be seen that guards showing the flows for patients who are dangerous and not dangerous to society.

→ Patients who are dangerous to society must be detained in a secure facility. However, patients who are suicidal and so are a danger to themselves may be detained in an appropriate ward in a hospital.

2.2 Interaction Models

→ All systems involve interaction of some kind.

→ This can be user interaction, which involves user inputs and outputs, interaction between the systems being developed and other systems or interaction between the components of the system.

→ Modeling system to system interaction highlights the communication problems that may arise.

→ There are 2 approaches to interaction modeling:

1. Use case modeling, which is mostly used to model interactions between a system and external actors (users or other systems).
2. Sequence diagrams, which are used to model interactions between system components, although external agents may also be included.

2.2.1 Use Case Modeling

→ Each use case represents a discrete task that involves external interaction with a system.

→ In its simplest form, a use case is shown as an ellipse with the actors involved in the use case represented as stick figures.

→ Fig 2.3 shows a use case from the MHC-PMS that represents the task of uploading data from the MHC-PMS to a more general patient record system.

Fig 2.3: Transfer data use case

→ This more general system maintains summary data about a patient rather than the data about each consultation, which is recorded in the MHC-PMS.

→ There are two actors in this use case: the operator who is transferring the data and the patient record system.

→ Use case diagrams give a fairly simple overview of an interaction so more details will have to be added in order to understand what is involved.

→ This detail can either be a simple textual description, a structured description in a table, or a sequence diagram.

→ Fig 2.4 shows a 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.

Fig 2.4: Tabular description of the transfer data use case

→ Figure 2.5 shows all of the use cases in the MHC-PMS in which the actor 'Medical Receptionist' is involved

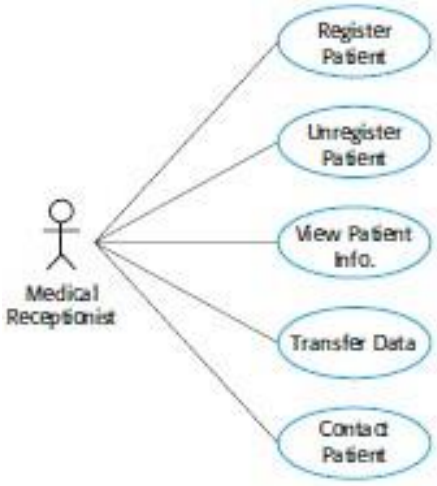

Fig 2.5: Use cases involving the role "medical receptionist"

2.2.2 Sequence Diagrams

→ Sequence diagrams in the UML are primarily used to model the interactions between the actors and the objects in a system and the interactions between the objects themselves.

→ A sequence diagram shows the sequence of interactions that take place during a particular use case or use case instance.

→ Fig 2.6 is an example of a sequence diagram that illustrates the basics of the notation.

→ This diagram models the interactions involved in the View patient information use case, where a medical receptionist can see some patient information.

→ Fig 2.6 can be read as follows:

1. The medical receptionist triggers the ViewInfo method in an instance P of the PatientInfo object class, supplying the patient's identifier, PID. P is a user interface object, which is displayed as a form showing patient information.
2. The instance P calls the database to return the information required, supplying the receptionist's identifier to allow security checking.
3. The database checks with an authorization system that the user is authorized for this action.
4. If authorized, the patient information is returned and a form on the user's screen is filled in. If authorization fails, then an error message is returned.

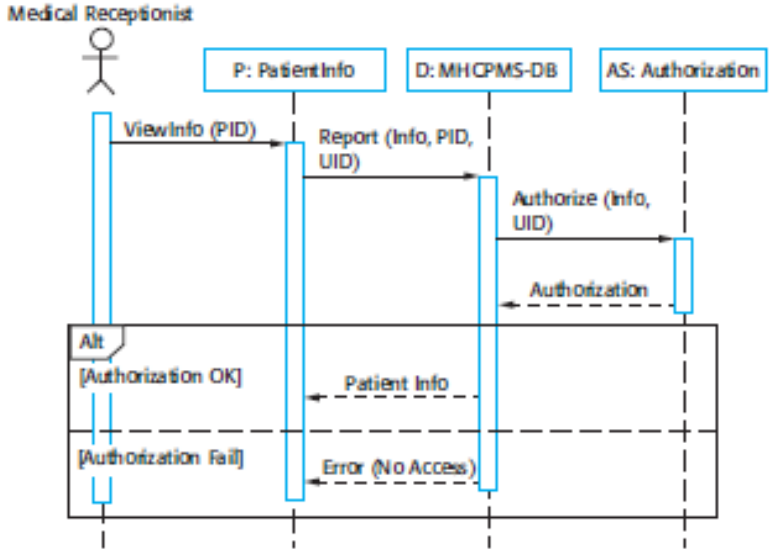

Fig 2.6: Sequence diagram for View patient information

→ Fig 2.7 is a second example of a sequence diagram from the same system that illustrates two additional features.

→ These are the direct communication between the actors in the system and the creation of objects as part of a sequence of operations.

→ Diagram below can be read as:

1. The receptionist logs on to the PRS.

2. There are two options available. These allow the direct transfer of updated patient information to the PRS and the transfer of summary health data from the MHC-PMS to the PRS.
3. In each case, the receptionist's permissions are checked using the authorization system.
4. Personal information may be transferred directly from the user interface object to the PRS. Alternatively, a summary record may be created from the database and that record is then transferred.
5. On completion of the transfer, the PRS issues a status message and the user logs off.

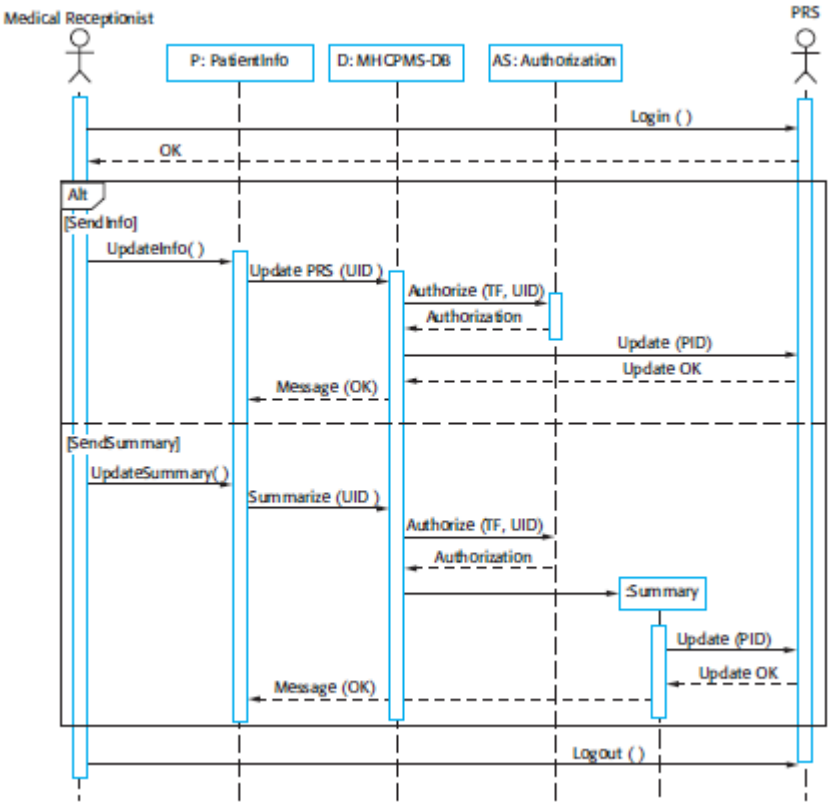

Fig 2.7: Sequence diagram for transfer data

2.3 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.

2.3.1 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 association is a link between classes that indicates that there is a relationship between these classes.

→ Class diagrams in the UML can be expressed at different levels of detail The simplest way of writing these is to write the class name in a box.

→ Note the existence of an association. by drawing a line between classes.

→ For example, Figure 2.8 is a simple class diagram showing two classes: Patient and Patient Record with an association between them.

→ In Fig 2.8, each end of the association is annotated with a 1, meaning that there is a 1:1 relationship between objects of these classes.

→ That is, each patient has exactly one record and each record maintains information about exactly one patient

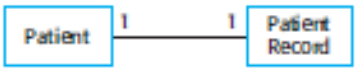

Fig 2.8: UML classes and association

→ Fig 2.9 develops this type of class diagram to show that objects of class Patient are also involved in relationships with a number of other classes.

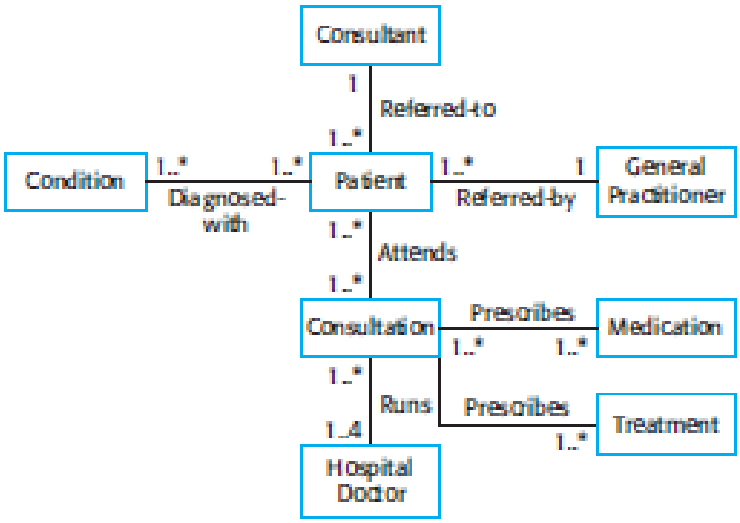

Fig 2.9: classes and associations in the MHC-PMS

→ At this level of detail, class diagrams look like semantic data models. Semantic data models are used in database design.

→ They show the data entities, their associated attributes, and the relations between these entities.

→ When showing the associations between classes, it is convenient to represent these classes in the simplest possible way.

→ To define them in more detail, information can be added about their attributes (the characteristics of an object) and operations .

→ For example, a Patient object will have the attribute Address and you may include an operation called ChangeAddress, which is called when a patient indicates that they have moved from one address to another.

→ In the UML, attributes and operations can be shown by extending the simple rectangle that represents a class. This is illustrated in Figure 2.10 where:

1. The name of the object class is in the top section.
2. The class attributes are in the middle section. This must include the attribute names and, optionally, their types.
3. The operations (called methods in Java and other OO programming languages) associated with the object class are in the lower section of the rectangle

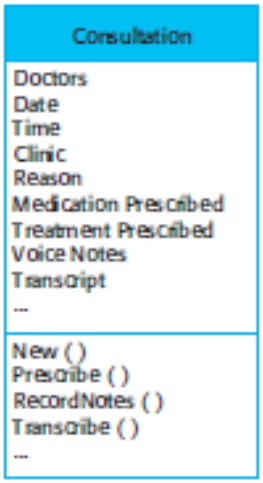

Fig 2.10: The consultation class

2.3.2 Generalization

→ This allows us to infer that different members of these classes have some common characteristics.

→ In modeling systems, it is often useful to examine the classes in a system to see if there is scope for generalization.

→ This means that common information will be maintained in one place only.

→ In object-oriented languages, such as Java, generalization is implemented using the class inheritance mechanisms built into the language.

→ The UML has a specific type of association to denote generalization, as illustrated in Fig 2.11.

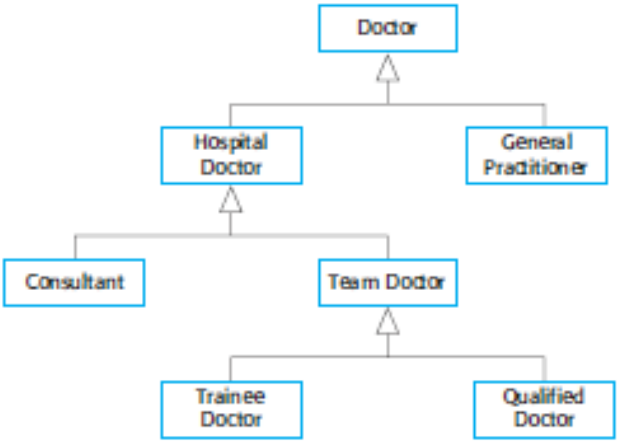

Fig 2.11: A generalization hierarchy

→ In a generalization, the attributes and operations associated with higher-level classes are also associated with the lower-level classes.

→ The generalization is shown as an arrowhead pointing up to the more general class.

→ This shows that general practitioners and hospital doctors can be generalized as doctors and that there are three types of Hospital Doctor— those that have just graduated from medical school and have to be supervised (Trainee Doctor); those that can work unsupervised as part of a consultant's team (Registered Doctor); and consultants, who are senior doctors with full decision making responsibilities.

→ Fig 2.12, shows part of the generalization hierarchy extended with class attributes.

→ The operations associated with the class Doctor are intended to register and de-register that doctor with the MHC-PMS.

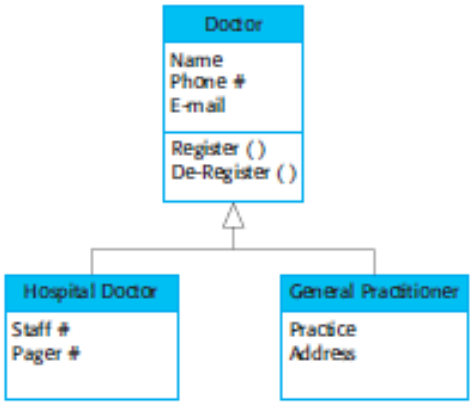

Fig 2.12: A generalization hierarchy with added detail

2.3.3 Aggregation

→ The UML provides a special type of association between classes called aggregation that means that one object (the whole) is composed of other objects (the parts).

→ To show this, a diamond shape is used next to the class that represents the whole. This is shown in Fig 2.13, which shows that a patient record is a composition of Patient and an indefinite number of Consultations.

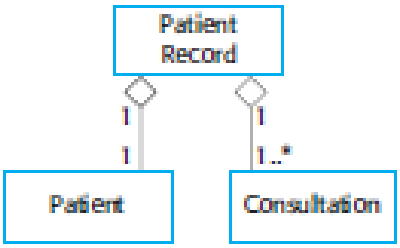

Fig 2.13: The aggregation association

2.4 Behavioral Models

→ Behavioral models are models of the dynamic behavior of the 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.

→ There are 2 types:

1. Data: Some data arrives that has to be processed by the system
2. Events: Some event happens that triggers system processing. Events may have associated data but this is not always the case.

2.4.1 Data-driven modeling

→ 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.

→ Data-flow models are useful because tracking and documenting how the data associated with a particular process moves through the system helps analysts and designers understand what is going on.

→ Data-flow diagrams (DFD's) are simple and intuitive and it is usually possible to explain them to potential system users who can then participate in validating the model.

→ Fig 2.14 shows the chain of processing involved in the insulin pump software.

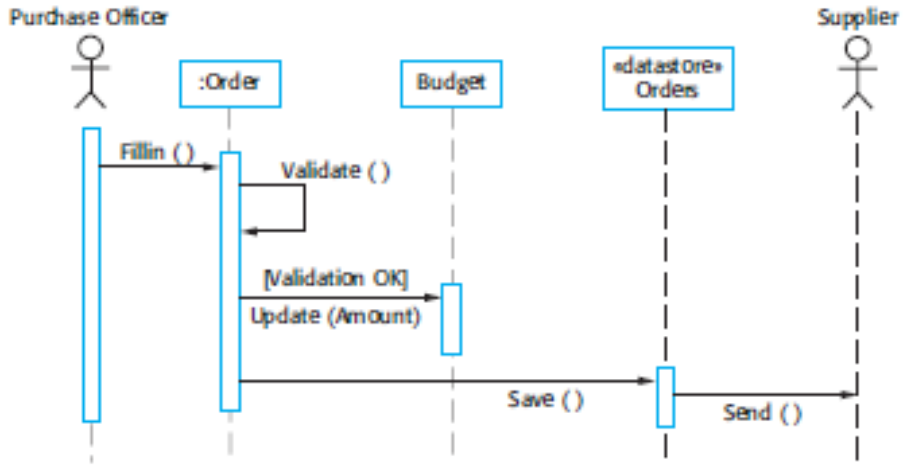

Fig 2.14: An activity model of the insulin pump's operation

→ Fig 2.15 illustrates the use of sequence model of the processing of an order and sending it to a supplier.

→ Sequence models highlight objects in a system, whereas data-flow diagrams highlight the functions.

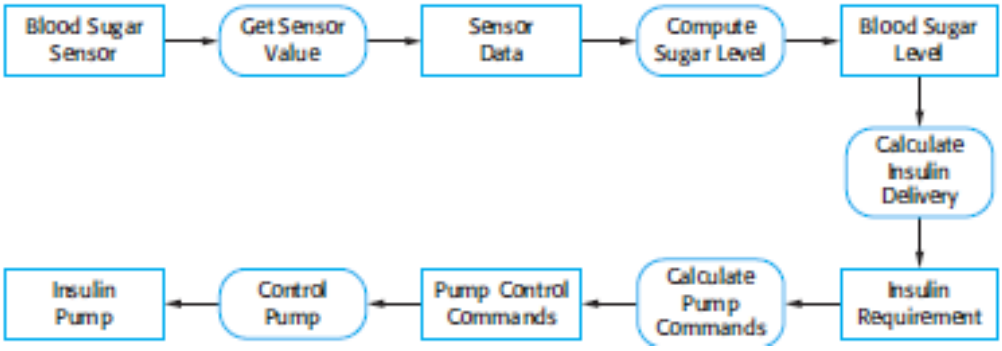

Fig 2.15: Order processing

2.4.2 Event driven modeling

→ 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.

→ For example, a system controlling a valve may move from a state 'Valve open' to a state 'Valve closed' when an operator command (the stimulus) is received.

→ The UML supports event-based modeling using state diagrams, which were based on Statecharts.

→ State diagrams show system states and events that cause transitions from one state to another.

→ They do not show the flow of data within the system but may include additional information on the computations carried out in each state.

→ The sequence of actions in using the microwave is:

→ 1. Select the power level (either half power or full power).

→ 2. Input the cooking time using a numeric keypad.

→ 3. Press Start and the food is cooked for the given time.

→ From fig 2.16, it can be seen that the system starts in a waiting state and responds initially to either the full-power or the half-power button.

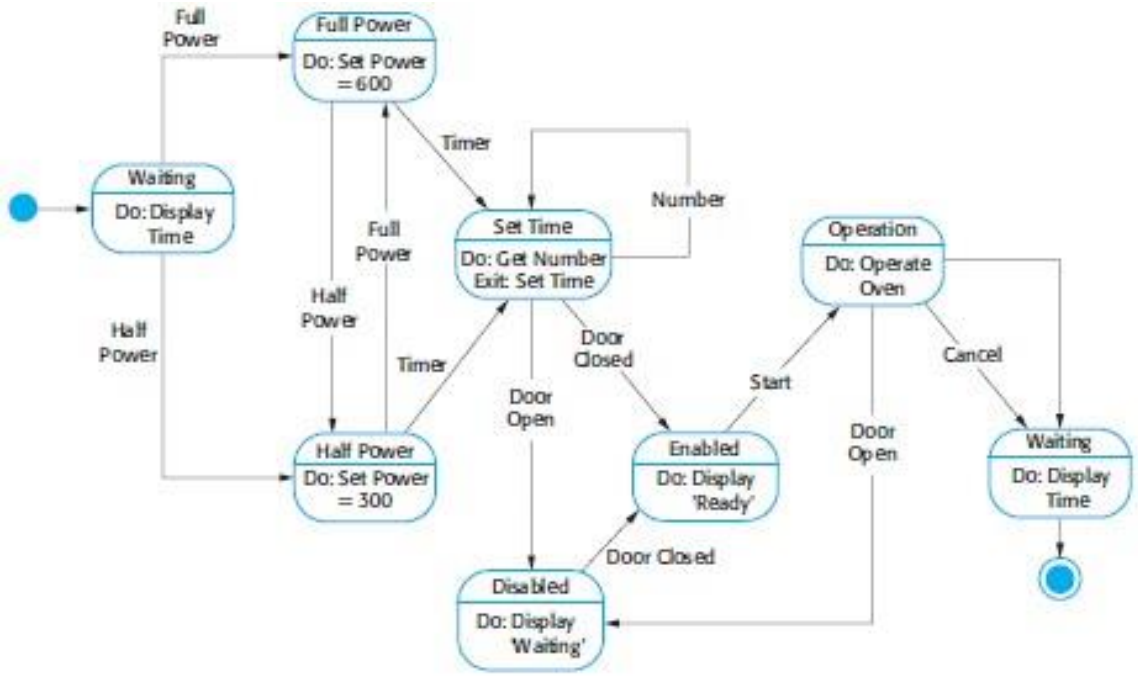

Fig 2.16: State diagram of a microwave oven

→ Users can change their mind after selecting one of these and press the other button.

→ The time is set and, if the door is closed, the Start button is enabled. Pushing this button starts the oven operation and cooking takes place for the specified time.

→ This is the end of the cooking cycle and the system returns to the waiting state.

→ Figure 2.17, shows a tabular description of each state and how the stimuli that force state transitions are generated.

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.
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.

Fig 2.17: States and stimuli for the microwave oven

→ The fig 2.18 below shows the microwave oven operation.

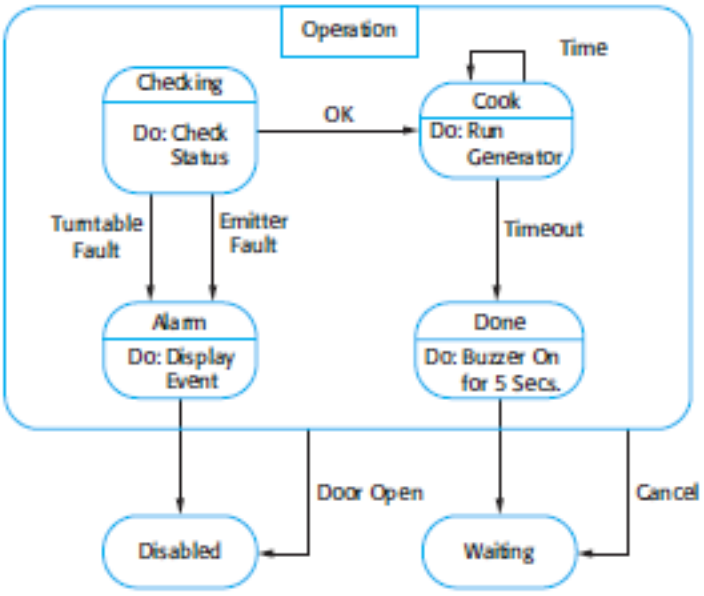

Fig 2.18: Microwave oven operation

2.5 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.

→ Model-driven engineering has its roots in model-driven architecture (MDA) which was proposed by the Object Management Group (OMG) in 2001 as a new software development paradigm.

→ The main arguments for and against MDE are:

1. **For MDE:**
   - Model-based engineering allows engineers to think about systems at a high level of abstraction, without concern for the details of their implementation.
   - This reduces the likelihood of errors, speeds up the design and implementation process, and allows for the creation of reusable, platform-independent application models.
2. **Against MDE:**
   - Models are a good way of facilitating discussions about a software design.
   - However, it does not always follow that the abstractions that are supported by the model are the right abstractions for implementation.
   - So, users may create informal design models but then go on to implement the system using an off-the-shelf, configurable package.
   - Furthermore, the arguments for platform independence are only valid for large long-lifetime systems where the platforms become obsolete during a system's lifetime.

2.5.1 Model-driven architecture

→ Model-driven architecture is a model-focused approach to software design and implementation that uses a sub-set of UML models to describe a system.

→ The MDA method recommends that three types of abstract system model should be produced:

→ A computation independent model (CIM) that models the important domain abstractions used in the system. CIMs are sometimes called domain models.

→ A platform independent model (PIM) that models 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) which 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.

→ Fig 2.19 shows a final level of automatic transformation. A transformation is applied to the PSM to generate executable code that runs on the designated software platform.

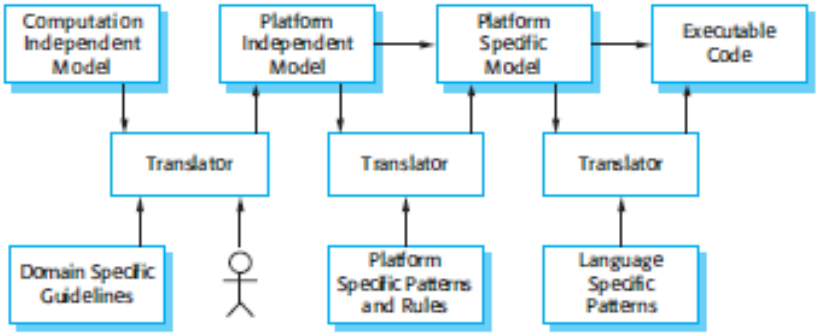

Fig 2.19: MDA Transformations

→ The fig 2.20 below shows multiple platform specific models

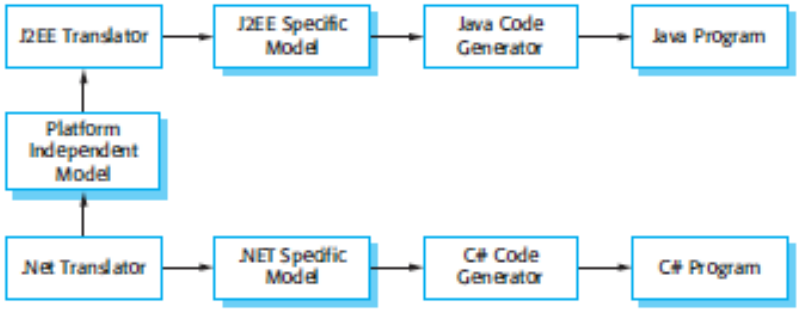

Fig 2.20: Multiple platform-specific models

2.5.2 Executable UML

→ UML was designed as a language for supporting and documenting software design, not as a programming language.

→ The designers of UML were not concerned with semantic details of the language but with its expressiveness.

→ They introduced useful notions such as use case diagrams that help with the design but which are too informal to support execution.

→ To create an executable sub-set of UML, the number of model types has therefore been dramatically reduced to three key model types:

1. Domain models identify the principal concerns in the system. These are defined using UML class diagrams that include objects, attributes, and associations.
2. Class models, in which classes are defined, along with their attributes and operations.
3. State models, in which a state diagram is associated with each class and is used to describe the lifecycle of the class.

DESIGN AND IMPLEMENTATION

2.6 Introduction to RUP (Rational Unified Process)

→ The RUP recognizes that conventional process models present a single view of the process.

→ In contrast, the RUP is normally described from three perspectives:

1. A dynamic perspective, which shows the phases of the model over time.
2. A static perspective, which shows the process activities that are enacted.
3. A practice perspective, which suggests good practices to be used during the process.

→ Fig 2.21 shows the phases in the RUP. These are:

1. **Inception:**
   - Goal: To establish a business case for the system.
   - It is necessary to identify all external entities (people and systems) that will interact with the system and define these interactions.
   - This information can then be used to assess the contribution that the system makes to the business.
2. **Elaboration:**
   - Goal: To develop an understanding of the problem domain, establish an architectural framework for the system, develop the project plan, and identify key project risks.
3. **Construction:**
   - Involves system design, programming, and testing.

* Parts of the system are developed in parallel and integrated during this phase.
* On completion of this phase, you should have a working software system and associated documentation that is ready for delivery to users.

4. Transition:

* It is concerned with moving the system from development community to the user community and making it work in a real environment.
* On completion of this phase, you should have a documented software system that is working correctly in its operational environment.

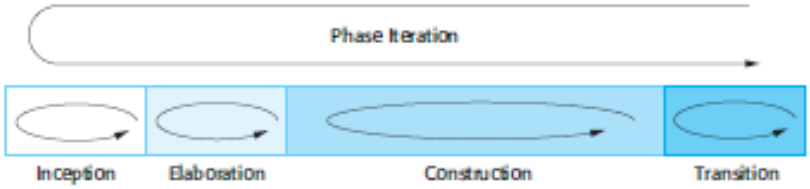

Fig 2.21: Phases in Rational Unified process

→ The core engineering and support workflows are described in Figure 2.22

Workflow	Description
Business modelling	The business processes are modelled using business use cases.
Requirements	Actors who interact with the system are identified and use cases are developed to model the system requirements.
Analysis and design	A design model is created and documented using architectural models, component models, object models, and sequence models.
Implementation	The components in the system are implemented and structured into implementation sub-systems. Automatic code generation from design models helps accelerate this process.
Testing	Testing is an iterative process that is carried out in conjunction with implementation. System testing follows the completion of the implementation.
Deployment	A product release is created, distributed to users, and installed in their workplace.
Configuration and change management	This supporting workflow manages changes to the system (see Chapter 25).
Project management	This supporting workflow manages the system development (see Chapters 22 and 23).
Environment	This workflow is concerned with making appropriate software tools available to the software development team.

Fig 2.22: Static workflows in the rational unified process

→ The practice perspective on the RUP describes good software engineering practices that are recommended for use in systems development.

→ Six fundamental best practices are recommended:

1. **Develop Software Iteratively:** Plan increments of the system based on customer priorities and develop the highest-priority system features early in the development process.
2. **Manage Requirements:** Explicitly document the customer's requirements and keep track of changes to these requirements. Analyze the impact of changes on the system before accepting them.
3. **Use Component-based Architectures:** Structure the system architecture into components.
4. **Visually Model Software:** Use graphical UML models to present static and dynamic views of the software.
5. **Verify Software Quality:** Ensure that the software meets the organizational quality standards.
6. **Control Changes to Software:** Manage changes to the software using a change management system and configuration management procedures and tools.

2.7 Design Principles

→ The design of a system is *correct* if a system built precisely according to the design satisfies the requirements of that system.

→ Clearly, the goal during the design phase is to produce correct designs.

→ The goal of the design process is not simply to produce *a* design for the system. Instead, the goal is to find the *best* possible design within the limitations imposed by the requirements and the physical and social environment in which the system will operate.

→ A design should clearly be verifiable, complete (implements all the specifications), and traceable (all design elements can be traced to some requirements).

→ Two most important properties that concern designers: efficiency and simplicity.

→ Efficiency of any system is concerned with the proper use of scarce resources by the system.

→ The design of a system is one of the most important factors affecting the maintainability of a system.

→ During maintenance, the first step a maintainer has to undertake is to understand the system to be maintained.

→ Only after a maintainer has a thorough understanding of the different modules of the system, how they are interconnected, and how modifying one will affect the others should the modification be undertaken.

2.7.1 Problem Partitioning and Hierarchy

→ For software design, therefore, the goal is to divide the problem into manageably small pieces that can be solved separately.

→ It is this restriction of being able to solve each part separately that makes dividing into pieces a complex task and that many methodologies for system design aim to address.

→ The different pieces cannot be entirely independent of each other, as they together form the system.

→ The different pieces have to cooperate and communicate to solve the larger problem.

→ This communication adds complexity, which arises due to partitioning and may not have existed in the original problem.

→ As the number of components increases, the cost of partitioning, together with the cost of this added complexity, may become more than the savings achieved by partitioning.

→ It is at this point that no further partitioning needs to be done. The designer has to make the judgment about when to stop partitioning.

→ Problem partitioning, which is essential for solving a complex problem, leads to hierarchies in the design. That is, the design produced by using problem partitioning can be represented as a hierarchy of components.

→ The relationship between the elements in this hierarchy can vary depending on the method used.

2.7.2 Abstraction

→ It is a tool that permits a designer to consider a component at an abstract level without worrying about the details of the implementation of the component.

→ Any component or system provides some services to its environment. An abstraction of a component describes the external behaviour of that component without bothering with the internal details that produce the behavior.

→ Abstraction is an indispensable part of the design process and is essential for problem partitioning.

→ Partitioning essentially is the exercise in determining the components of a system.

→ However, these components are not isolated from each other; they interact with each other, and the designer has to specify how a component interacts with other components.

→ Abstraction is used for existing components as well as components that are being designed.

→ Abstraction of existing components plays an important role in the maintenance phase.

→ To modify a system, the first step is understanding what the system does and how.

→ The process of comprehending an existing system involves identifying the abstractions of subsystems and components from the details of their implementations.

→ Using these abstractions, the behavior of the entire system can be understood. This also helps determine how modifying a component affects the system.

→ There are two common abstraction mechanisms for software systems: ***functional abstraction*** **and** ***data abstraction***.

→ In *functional abstraction,* a module is specified by the function it performs. For example, a module to compute the log of a value can be abstractly represented by the function log.

→ The second unit for abstraction is *data abstraction.* Any entity in the real world provides some services to the environment to which it belongs. Often the entities provide some fixed predefined services. The case of data entities is similar.

→ Certain operations are required from a data object, depending on the object and the environment in which it is used. Data abstraction supports this view.

→ Data is not treated simply as objects, but is treated as objects with some predefined operations on them.

2.7.3 Modularity

→ Modularity is a clearly a desirable property in a system.

→ Modularity helps in system debugging—isolating the system problem to a component is easier if the system is modular; in system repair—changing a part of the system is easy as it affects few other parts; and in system building—a modular system can be easily built by "putting its modules together."

→ A software system cannot be made modular by simply chopping it into a set of modules.

→ For modularity, each module needs to support a well defined abstraction and have a clear interface through which it can interact with other modules.

→ Modularity is where abstraction and partitioning come together.

2.7.4 Top-Down and Bottom-Up Strategies

→ A system consists of components, which have components of their own; indeed a system is a hierarchy of components.

→ The highest-level component corresponds to the total system. To design such a hierarchy there is two possible approaches: top-down and bottom-up.

→ A top-down design approach starts by identifying the major components of the system, decomposing them into their lower-level components and iterating until the desired level of detail is achieved.

→ Top-down design methods often result in some form of *stepwise refinement* Starting from an abstract design, in each step the design is refined to a more concrete level, until we reach a level where no more refinement is needed and the design can be implemented directly.

→ A bottom-up design approach starts with designing the most basic or primitive components and proceeds to higher-level components that use these lower-level components.

→ Bottom-up methods work with *layers of abstraction.* Starting from the very bottom, operations that provide a layer of abstraction are implemented.

→ The operations of this layer are then used to implement more powerful operations and a still higher layer of abstraction, until the stage is reached where the operations supported by the layer are those desired by the system.

→ A common approach to combine the two approaches is to provide a layer of abstraction for the application domain of interest through libraries of functions, which contains the functions of interest to the application domain.

→ Then use a top-down approach to determine the modules in the system, assuming that the abstract machine available for implementing the system provides the operations supported by the abstraction layer.

2.8 Object-oriented design using the UML

→ An object-oriented system is made up of interacting objects that maintain their own local state and provide operations on that state.

→ Object-oriented systems are easier to change than systems developed using functional approaches.

→ Objects include both data and operations to manipulate that data.

→ They may therefore be understood and modified as stand-alone entities.

→ Changing the implementation of an object or adding services should not affect other system objects.

→ To develop a system design from concept to detailed, object-oriented design, there are several things that you need to do:

* Understand and define the context and the external interactions with the system.
* Design the system architecture.
* Identify the principal objects in the system.
* Develop design models.
* Specify interfaces.

2.8.1 System Context and Interactions

→ The first stage in any software design process is to develop an understanding of the relationships between the software that is being designed and its external environment.

→ This is essential for deciding how to provide the required system functionality and how to structure the system to communicate with its environment.

→ Understanding of the context also lets you establish the boundaries of the system.

→ Setting the system boundaries helps you decide what features are implemented in the system being designed and what features are in other associated systems.

→ System context models and interaction models present complementary views of the relationships between a system and its environment:

* A system context model is a structural model that demonstrates the other systems in the environment of the system being developed.
* An interaction model is a dynamic model that shows how the system interacts with its environment as it is used.

→ The context model of a system may be represented using associations.

→ Associations simply show that there are some relationships between the entities involved in the association.

→ The environment of the system can be represented using a simple block diagram showing the entities in the system and their associations.

→ This is illustrated in fig 2.23, which shows that the systems in the environment of each weather station are a weather information system, an onboard satellite system, and a control system.

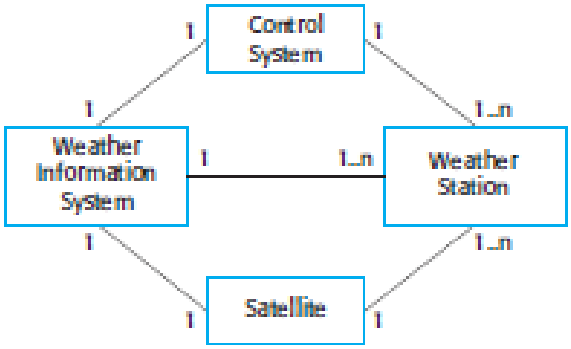

Fig 2.23: System context for the weather station

→ The cardinality information on the link shows that there is one control system but several weather stations, one satellite, and one general weather information system.

→ The use case model for the weather station is shown in fig 2.24.

→ This shows that the weather station interacts with the weather information system to report weather data and the status of the weather station hardware

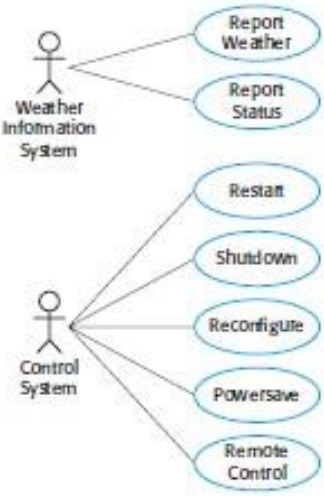

Fig 2.24: Weather station use cases

→ The use case description is shown in fig 2.25.

System	Weather station
Use case	Report weather
Actors	Weather information system, Weather station
Dat	The weather station sends a summary of the weather data that has been collected from the instruments in the collection period to the weather information system. The data sent are the maximum, minimum, and average ground and air temperatures; the maximum, minimum, and average air pressures; the maximum, minimum, and average wind speeds; the total rainfall; and the wind direction as sampled at five-minute intervals.
Stimulus	The weather information system establishes a satellite communication link with the weather station and requests transmission of the data.
Response	The summarized data are sent to the weather information system.
Comments	Weather stations are usually asked to report once per hour but this frequency may differ from one station to another and may be modified in the future.

Fig 2.25: Use case description – Report weather

2.8.2 Architectural Design

→ Once the interactions between the software system and the system's environment have been defined, this information is used as a basis for designing the system architecture.

→ The high-level architectural design for the weather station software is shown in fig 2.26.

→ The weather station is composed of independent subsystems that communicate by broadcasting messages on a common infrastructure, shown as the Communication link in fig 2.26.

→ Each subsystem listens for messages on that infrastructure and picks up the messages that are intended for them.

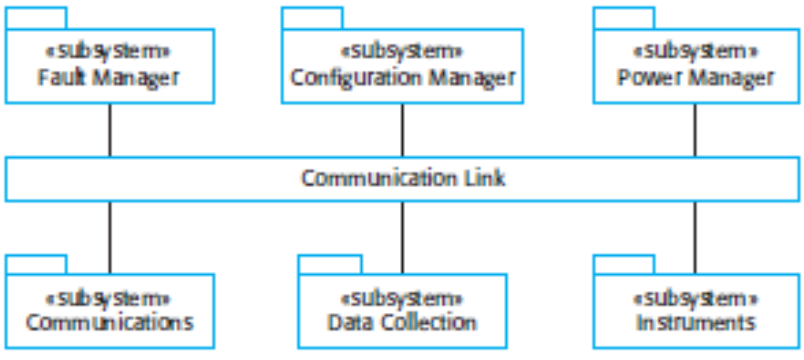

Fig 2.26: High level architecture of the weather station

→ Fig 2.27 shows the architecture of the data collection subsystem, which is included in fig 2.26.

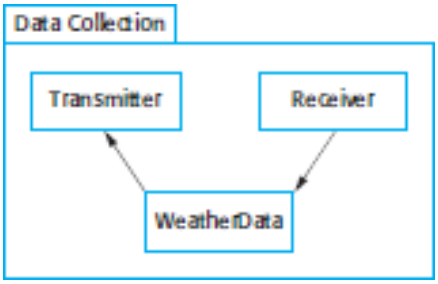

Fig 2.27: Architecture of data collection system

→ The Transmitter and Receiver objects are concerned with managing communications and the WeatherData object encapsulates the information that is collected from the instruments and transmitted to the weather information system.

→ This arrangement follows the producer-consumer pattern

→ There have been various proposals made about how to identify object classes in object oriented systems:

* Use a grammatical analysis of a natural language description of the system to be constructed. Objects and attributes are nouns; operations or services are verbs.
* Use tangible entities (things) in the application domain such as aircraft, roles such as manager or doctor, events such as requests, interactions such as meetings, locations such as offices, organizational units such as companies, and so on.
* Use a scenario-based analysis where various scenarios of system use are identified and analyzed in turn.

→ There are five object classes in fig 2.28.

→ The Ground thermometer, Anemometer, and Barometer objects are application domain objects, and the WeatherStation and WeatherData objects have been identified from the system description and the scenario (use case) description:

* The WeatherStation object class provides the basic interface of the weather station with its environment.
* The WeatherData object class is responsible for processing the report weather command. It sends the summarized data from the weather station instruments to the weather information system.
* The Ground thermometer, Anemometer, and Barometer object classes are directly related to instruments in the system. They reflect tangible hardware

entities in the system and the operations are concerned with controlling that hardware. These objects operate autonomously to collect data at the specified frequency and store the collected data locally. This data is delivered to the WeatherData object on request.

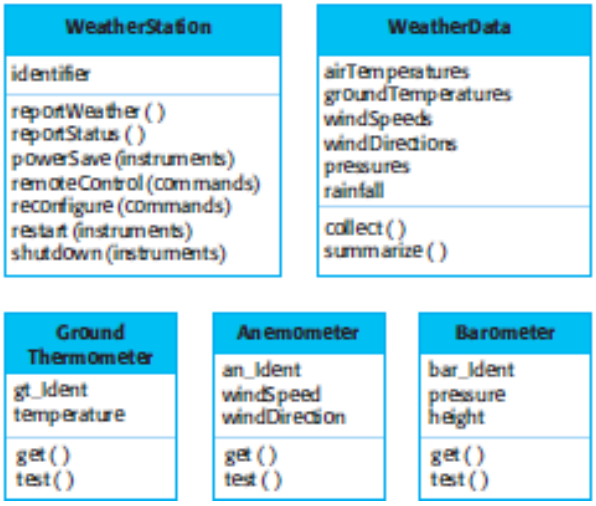

Fig 2.28: Weather station objects

2.8.3 Design models

→ Design or system models, show the objects or object classes in a system. They also show the associations and relationships between these entities.

→ These models are the bridge between the system requirements and the implementation of a system.

→ An important step in the design process, therefore, is to decide on the design models needed and the level of detail required in these models.

→ This depends on the type of system that is being developed.

→ When UML is used to develop a design, there are two kinds of design models to be developed.

1. Structural models, which describe the static structure of the system using object classes and their relationships. Important relationships that may be documented at this stage are generalization (inheritance) relationships, uses/used-by relationships, and composition relationships.
2. Dynamic models, which describe the dynamic structure of the system and show the interactions between the system objects. Interactions that may be documented include the sequence of service requests made by objects and the state changes that are triggered by these object interactions.

→ In the early stages of the design process, there are three models that are particularly useful for adding detail to use case and architectural models:

1. Subsystem models, which that show logical groupings of objects into coherent subsystems. These are represented using a form of class diagram with each subsystem shown as a package with enclosed objects. Subsystem models are static (structural) model
2. Sequence models, which show the sequence of object interactions. These are represented using a UML sequence or a collaboration diagram. Sequence models are dynamic models.
3. State machine model, which show how individual objects change their state in response to events. These are represented in the UML using state diagrams. State machine models are dynamic models.

→ Fig 2.29 is an example of a sequence model, shown as a UML sequence diagram.

→ This diagram shows the sequence of interactions that take place when an external system requests the summarized data from the weather station. Sequence diagrams are read from top to bottom:

1. The SatComms object receives a request from the weather information system to collect a weather report from a weather station. It acknowledges receipt of this request. The stick arrowhead on the sent message indicates that the external system does not wait for a reply but can carry on with other processing.
2. SatComms sends a message to WeatherStation, via a satellite link, to create a summary of the collected weather data. Again, the stick arrowhead indicates that SatComms does not suspend itself waiting for a reply.
3. WeatherStation sends a message to a Commslink object to summarize the weather data. In this case, the squared-off style of arrowhead indicates that the instance of the WeatherStation object class waits for a reply.
4. Commslink calls the summarize method in the object WeatherData and waits for a reply.
5. The weather data summary is computed and returned to WeatherStation via the Commslink object.

6. WeatherStation then calls the SatComms object to transmit the summarized data to the weather information system, through the satellite communications system.

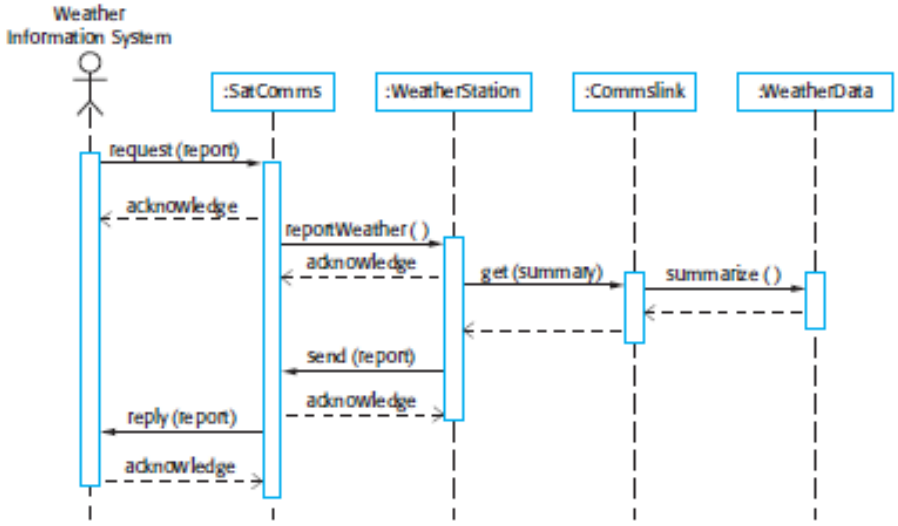

Fig 2.29: Sequence diagram describing data collection

→ Fig 2.30 is a state diagram for the weather station system that shows how it responds to requests for various services.

→ This diagram can be read as follows:

1. If the system state is Shutdown then it can respond to a restart(), a reconfigure(), or a powerSave() message. The unlabeled arrow with the black blob indicates that the Shutdown state is the initial state. A restart() message causes a transition to norma operation. Both the powerSave() and reconfigure() messages cause a transition to a state in which the system reconfigures itself. The state diagram shows that reconfiguration is only allowed if the system has been shut down.
2. In the Running state, the system expects further messages. If a shutdown() message is received, the object returns to the shutdown state.
3. If a reportWeather() message is received, the system moves to the Summarizing state. When the summary is complete, the system moves to a Transmitting state where the information is transmitted to the remote system. It then returns to the Running state.
4. If a reportStatus() message is received, the system moves to the Testing state, then the Transmitting state, before returning to the Running state.

5. If a signal from the clock is received, the system moves to the Collecting state, where it collects data from the instruments. Each instrument is instructed in turn to collect its data from the associated sensors.
6. If a remoteControl() message is received, the system moves to a controlled statein which it responds to a different set of messages from the remote control room. These are not shown on this diagram.

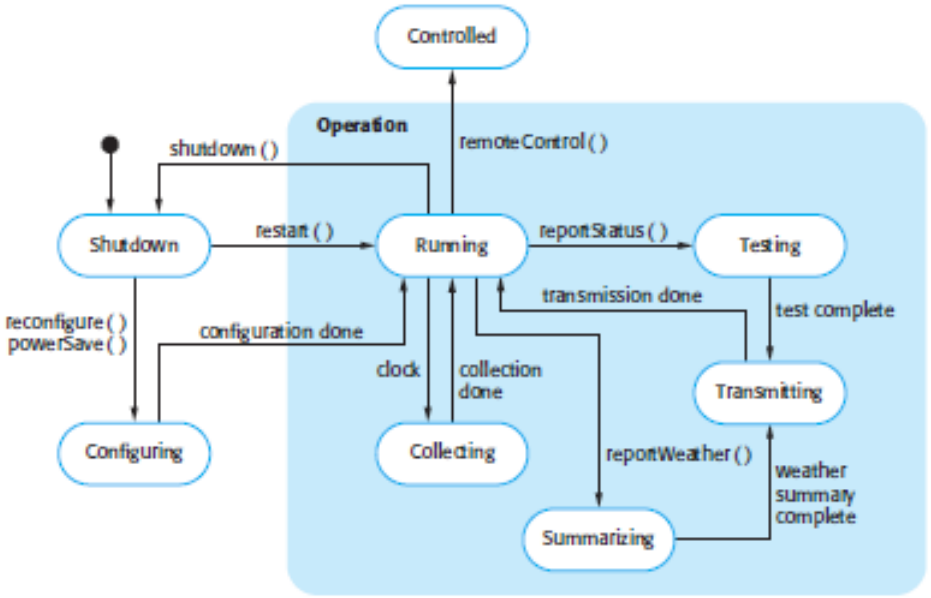

Fig 2.30: Weather station state diagram

2.8.4 Interface Specification

→ Interface design is concerned with specifying the detail of the interface to an object or to a group of objects.

→ This means defining the signatures and semantics of the services that are provided by the object or by a group of objects.

→ Interfaces can be specified in the UML using the same notation as a class diagram

→ Details of the data representation should not be included in an interface design, as attributes are not defined in an interface specification.

→ However, operations can be included to access and update data.

→ As the data representation is hidden, it can be easily changed without affecting the objects that use that data. This leads to a design that is inherently more maintainable.

→ Fig 2.31 shows two interfaces that may be defined for the weather station.

→ The left-hand interface is a reporting interface that defines the operation names that are used to generate weather and status reports.

→ These maps directly to operations in the WeatherStation object. The remote control interface provides four operations, which map onto a single method in the WeatherStation object.

Fig 2.31: Weather station interfaces

2.9 Design Patterns

→ The pattern is a description of the problem and the essence of its solution, so that the solution may be reused in different settings.

→ The pattern is not a detailed specification.

→ **Patterns and Pattern Languages are ways to describe best practices, good designs, and capture experience in a way that it is possible for others to reuse this experience.**

→ Design patterns are usually associated with object-oriented design.

→ The general principle of encapsulating experience in a pattern is one that is equally applicable to any kind of software design

→ The four essential elements of design patterns were defined by the 'Gang of Four' in their patterns book:

* A name that is a meaningful reference to the pattern.
* A description of the problem area that explains when the pattern may be applied.
* A solution description of the parts of the design solution, their relationships, and their responsibilities. This is not a concrete design description. It is a template for a design solution that can be instantiated in different ways. This is often expressed graphically and shows the relationships between the objects and object classes in the solution.
* A statement of the consequences—the results and trade-offs—of applying the pattern. This can help designers understand whether or not a pattern can be used in a particular situation.

→ Observer pattern is as shown in fig 2.32. This pattern can be used in situations where different presentations of an object's state are required.

Pattern name: Observer
Description: Separates the display of the state of an object from the object itself and allows alternative displays to be provided. When the object state changes, all displays are automatically notified and updated to reflect the change.
Problem description: In many situations, you have to provide multiple displays of state information, such as a graphical display and a tabular display. Not all of these may be known when the information is specified. All alternative presentations should support interaction and, when the state is changed, all displays must be updated.

This pattern may be used in all situations where more than one display format for state information is required and where it is not necessary for the object that maintains the state information to know about the specific display formats used.
Solution description: This involves two abstract objects, Subject and Observer, and two concrete objects, ConcreteSubject and ConcreteObject, which inherit the attributes of the related abstract objects. The abstract objects include general operations that are applicable in all situations. The state to be displayed is maintained in ConcreteSubject, which inherits operations from Subject allowing it to add and remove Observers (each observer corresponds to a display) and to issue a notification when the state has changed.

The ConcreteObserver maintains a copy of the state of ConcreteSubject and implements the Update() interface of Observer that allows these copies to be kept in step. The ConcreteObserver automatically displays the state and reflects changes whenever the state is updated.
The UML model of the pattern is shown in Figure 7.12.
Consequences: The subject only knows the abstract Observer and does not know details of the concrete class. Therefore there is minimal coupling between these objects. Because of this lack of knowledge, optimizations that enhance display performance are impractical. Changes to the subject may cause a set of linked updates to observers to be generated, some of which may not be necessary.

Fig 2.32: The Observer pattern

→ It separates the object that must be displayed from the different forms of presentation which is shown in fig 2.33.

→ Fig 2.34 is the representation in UML of the Observer pattern.

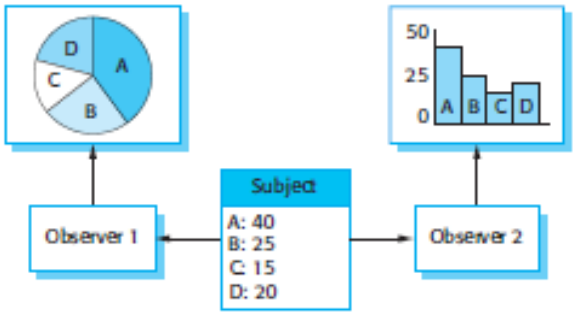

Fig 2.33: Multiple displays

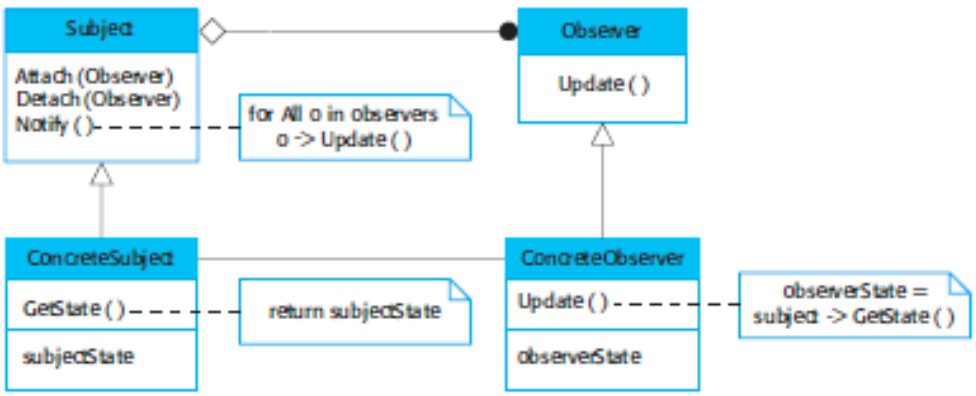

Fig 2.34: A UML model of the observer pattern

2.10 Implementation Issues

→ Implementation may involve developing programs in high- or low-level programming languages or tailoring and adapting generic, off-the-shelf systems to meet the specific requirements of an organization.

→ Few aspects of implementation that are particularly important to software engineering that are often not covered in programming texts. These are:

1. **Reuse:** Most modern software is constructed by reusing existing components or systems. When the software is being developed, the existing code must be used as much as possible.
2. **Configuration management:** During the development process, many different versions of each software component are created. If records of these versions are not maintained in a configuration management system, there is probability of including wrong versions of these components in the system.
3. **Host-target development:** Production software does not usually execute on the same computer as the software development environment. The host and target systems are sometimes of the same type but, often they are completely different.

2.10.1 Reuse

→ Software reuse is possible at a number of different levels:

1. **The abstraction level:** At this level, software is not reused directly but ratheruse knowledge of successful abstractions in the design of your software. Design patterns and architectural patterns are ways of representing abstract knowledge for reuse.
2. **The object level:** At this level, objects are directly reused from a library rather than writing the code yourself. To implement this type of reuse, you have to find appropriate libraries and discover if the objects and methods offer the functionality that you need.
3. **The component level:** Components are collections of objects and object classes that operate together to provide related functions and services. It is required to adapt and extend the component by adding some code. An example of component-level reuse is where you build your user interface using a framework.

4. **The system level:** At this level, the entire application systems are reused. This usually involves some kind of configuration of these systems. This may be done by adding and modifying code (if you are reusing a software product line) or by using the system's own configuration interface. Most commercial systems are now built in this way where generic COTS (commercial off-the-shelf) systems are adapted and reused.

→ There are costs associated with reuse:

* The costs of the time spent in looking for software to reuse and assessing whether or not it meets your needs. The software will have to be tested to make sure that it will work in own environment, especially if this is different from its development environment.
* Where applicable, the costs of buying the reusable software. For large off-the shelf systems, these costs can be very high.
* The costs of adapting and configuring the reusable software components or systems to reflect the requirements of the system that you are developing.
* The costs of integrating reusable software elements with each other (if you are using software from different sources) and with the new code that has been developed.

2.10.2 Configuration Management

→ Configuration management is the name given to the general process of managing a changing software system.

→ The aim of configuration management is to support the system integration process so that all developers can access the project code and documents in a controlled way, find out what changes have been made, and compile and link components to create a system.

→ There are, therefore, three fundamental configuration management activities:

* Version management, where support is provided to keep track of the different versions of software components. Version management systems include facilities to coordinate development by several programmers. They stop one developer overwriting code that has been submitted to the system by someone else.
* System integration, where support is provided to help developers define what versions of components are used to create each version of a system. This

description is then used to build a system automatically by compiling and linking the required components.

* Problem tracking, where support is provided to allow users to report bugs and other problems, and to allow all developers to see who is working on these problems and when they are fixed.

2.10.3 Host-target development

→ A platform is more than just hardware.

→ It includes the installed operating system plus other supporting software such as a database management system or, for development platforms, an interactive development environment. Simulators are often used when developing embedded systems.

→ Hardware devices can be simulated, such as sensors, and the events in the environment in which the system will be deployed.

→ If the target system has installed middleware or other software that can be used, then it is necessary to test the system using that software.

→ It may be impractical to install that software on the development machine, even if it is the same as the target platform, because of license restrictions.

→ A software development platform should provide a range of tools to support software engineering processes. These may include:

* An integrated compiler and syntax-directed editing system that allows users to create, edit, and compile code.
* A language debugging system.
* Graphical editing tools, such as tools to edit UML models.
* Testing tools, such as JUnit that can automatically run a set of tests on a new version of a program.
* Project support tools that help you organize the code for different development projects.

→ Software development tools are often grouped to create an integrated development environment (IDE).

→ An IDE is a set of software tools that supports different aspects of software development, within some common framework and user interface

→ A general-purpose IDE is a framework for hosting software tools that provides data management facilities for the software being developed, and integration mechanisms, that allow tools to work together.

→ For distributed systems, it is essential to decide on the specific platforms where the components will be deployed.

→ Issues that should be considered in making this decision are:

* **The hardware and software requirements of a component:** If a component is designed for a specific hardware architecture, or relies on some other software system, it must obviously be deployed on a platform that provides the required hardware and software support.
* **The availability requirements of the system:** High-availability systems may require components to be deployed on more than one platform. This means that, in the event of platform failure, an alternative implementation of the component is available.
* **Component communications**: If there is a high level of communications traffic between components, it usually makes sense to deploy them on the same platform or on platforms that are physically close to one other. This reduces communications latency, the delay between the time a message is sent by one component and received by another.

2.11 Open Source Development

→ Open source development is an approach to software development in which the source code of a software system is published and volunteers are invited to participate in the development process.

→ Open source software extended this idea by using the Internet to recruit a much larger population of volunteer developers.

→ For a company involved in software development, there are two open source issues that have to be considered:

* Should the product that is being developed make use of open source components?
* Should an open source approach be used for the software's development?

→ The answers to these questions depend on the type of software that is being developed and the background and experience of the development team.

2.11.1 Open source licensing

→ Most open source licenses are derived from one of three general models:

1. **The GNU General Public License (GPL).** This is a so-called 'reciprocal' license that, simplistically, means that if you use open source software that is licensed under the GPL license, then you must make that software open source.
2. **The GNU Lesser General Public License (LGPL).** This is a variant of the GPL license where it is possible to write components that link to open source code without having to publish the source of these components. However, if the licensed component is changed, then it must be published as open source.
3. **The Berkley Standard Distribution (BSD) License.** This is a non-reciprocal license, which means you are not obliged to republish any changes or modifications made to open source code. The code can be included in proprietary systems that are sold.

→ Companies managing projects that use open source should:

* Establish a system for maintaining information about open source components that are downloaded and used.
* Be aware of the different types of licenses and understand how a component is licensed before it is used.
* Be aware of evolution pathways for components.
* Educate people about open source. It's not enough to have procedures in place to ensure compliance with license conditions.
* Have auditing systems in place. Developers, under tight deadlines, might be tempted to break the terms of a license.
* Participate in the open source community. If you rely on open source products, you should participate in the community and help support their development