MODULE 3

System Modeling

- **Context Models**
- **Interaction Models**
- **Structural Models**
- **Behavioral Models**
- **Model Driven Engineering**

****

System modeling is the process of developing abstract models of a system, with each model presenting a different view or perspective of that system. It is about representing a system using some kind of graphical notation, which is now almost always based on notations in the **Unified Modeling Language** (**UML**). Models help the analyst to understand the functionality of the system; they are used to communicate with customers.

Models can explain the system from **different 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.

Five types of UML diagrams that are the most useful for system modeling:

- **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.

Models of both new and existing system are used during **requirements engineering**. Models of the **existing systems** 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.

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** 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.

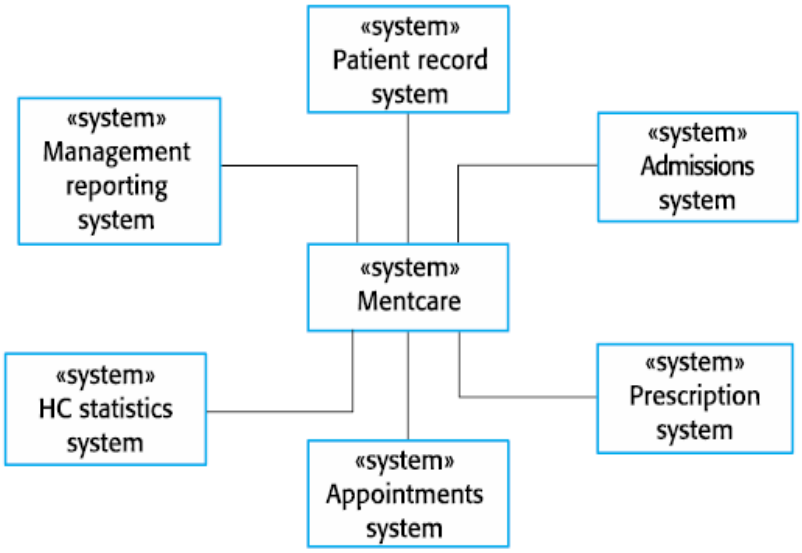

Fig: The Context of the MHC-PMC

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.

The example below shows a UML **activity diagram** describing the process of involuntary detention and the role of MHC-PMS (mental healthcare patient management system) in it.

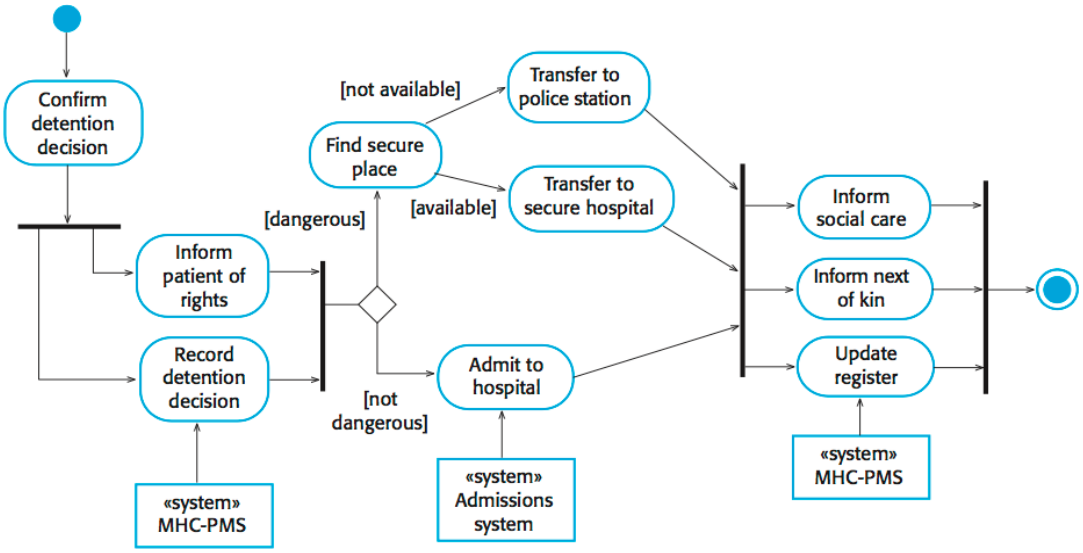

Fig: Process model of involuntary detection

- The above figure 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
- In a UML activity diagram, arrows represent the flow of work from one activity to another.
- A solid bar is used to indicate activity coordination.
- When the flow from more than one activity leads to a solid bar then all of these activities must be complete before progress is possible.
- When the flow from a solid bar leads to a number of activities, these may be executed in parallel
- Arrows may be annotated with guards that indicate the condition when that flow is taken

Interaction Models

Types of interactions that can be represented in a model:

- 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 modeling

- **Use cases** were developed originally to support requirements elicitation and now incorporated into the UML.
- Each use case represents a discrete task that involves external interaction with a system.
- Actors in a use case may be people or other systems.
- Use cases can be represented using a UML use case diagram and in a more detailed textual/tabular format.

Simple use case diagram:

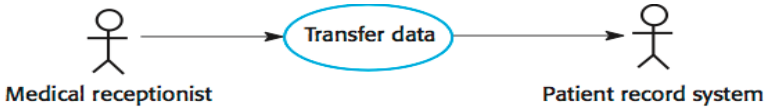

Fig: transfer data use case

Use case description in a tabular format

MHC-PMS: Transfer data	
Actors	Medical receptionist, patient records system (PRS)
Description	A receptionist may transfer data from the Mentcase system 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.

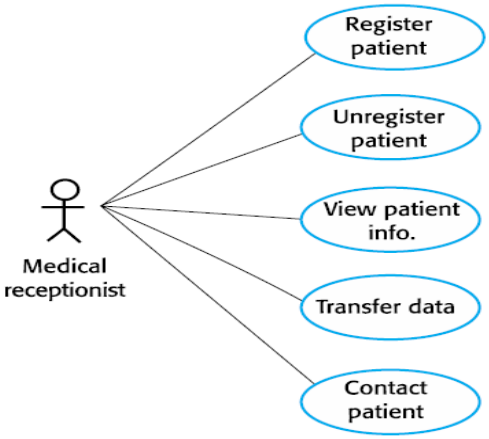

Fig: use cases involving the role 'medical receptionist'

Sequence diagrams

- **Sequence diagrams** 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.

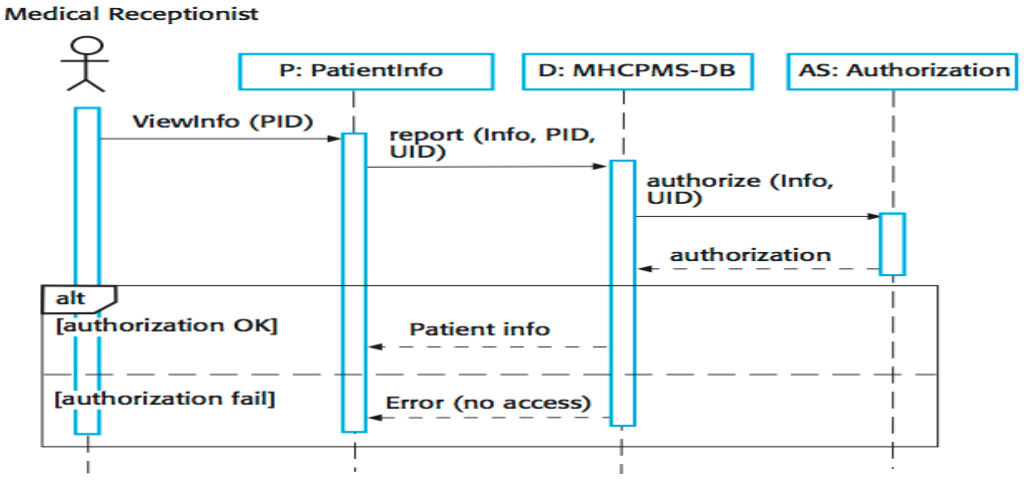

Fig: sequence diagram for view patient information

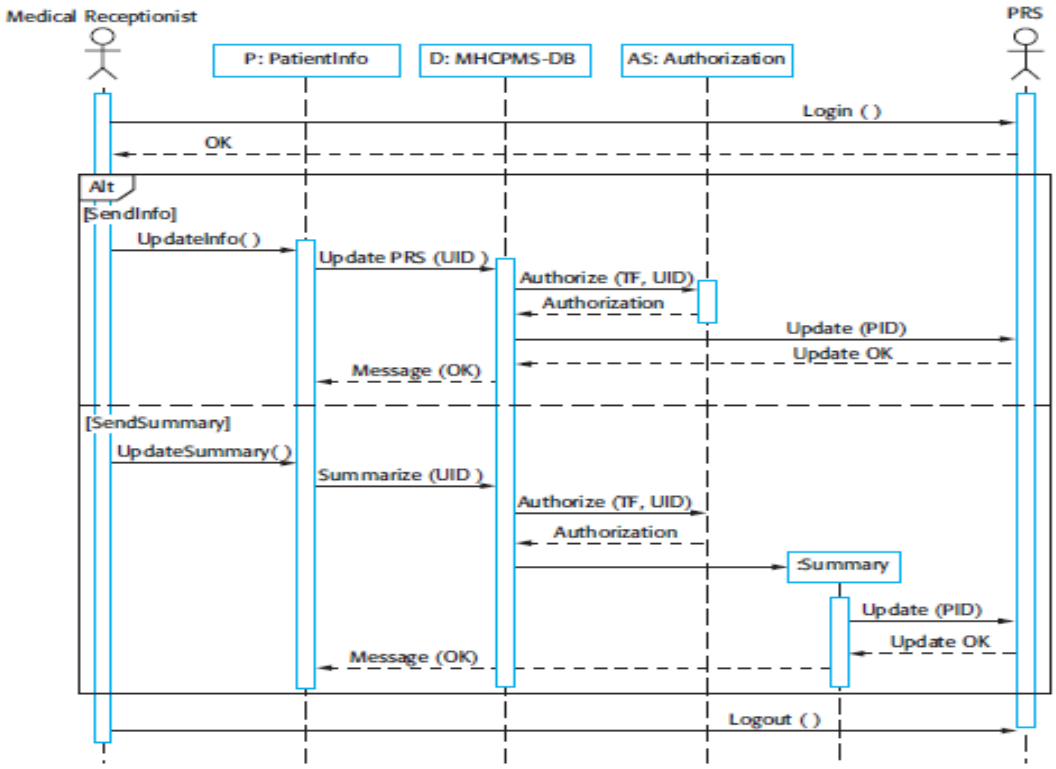

Fig: sequence diagram for transfer data

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.

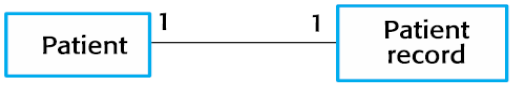

Fig: UML classes and association

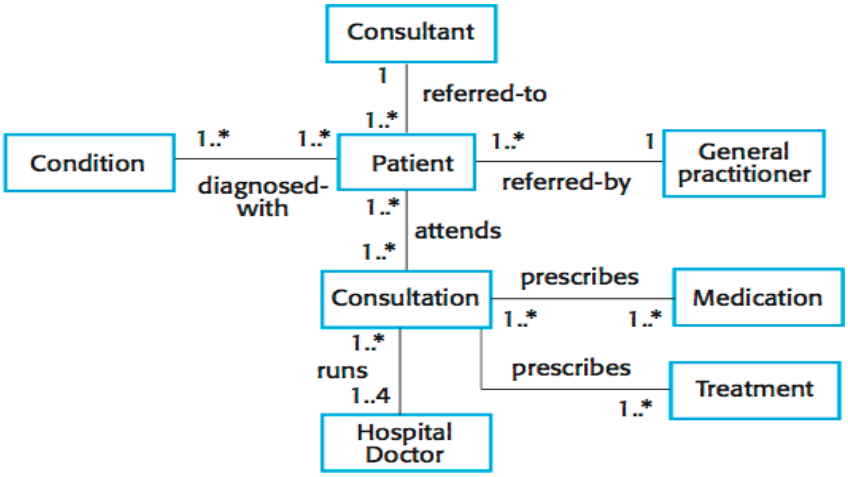

Fig: classes and association in the MHC-PMS

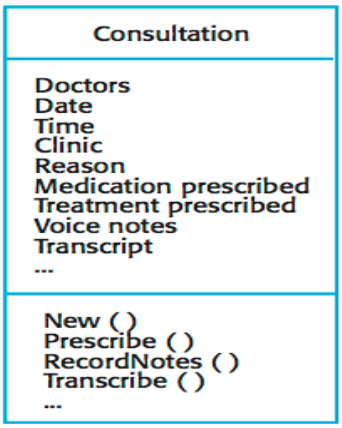

Fig: the consultation class

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 options (called methods in Java and other OO programming languages) associated with the object class are in the lower section of the rectangle.

Generalization

- **Generalization** is an everyday technique that we use to manage complexity.
- In modeling systems, it is often useful to examine the classes in a system to see if there is scope for generalization.
- 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.

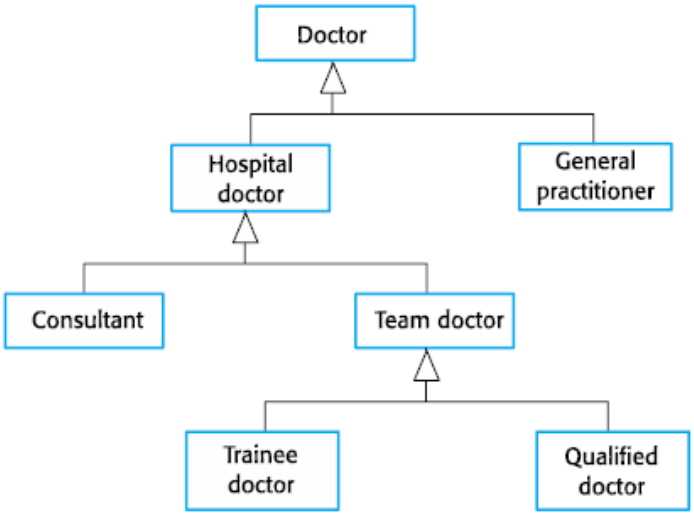

Fig: A generalization hierarchy

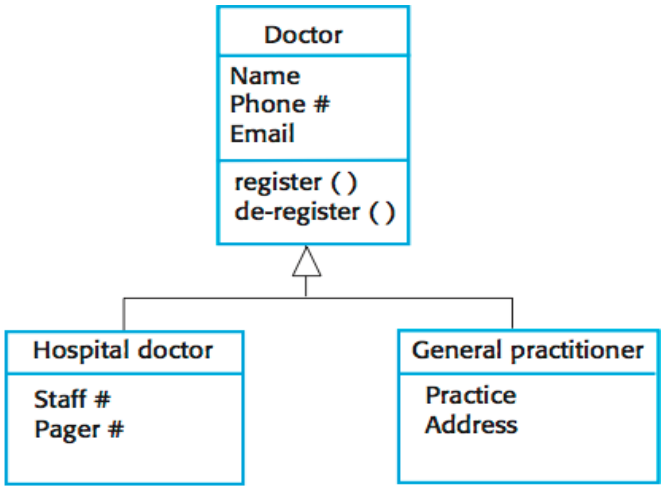

Fig: A generalization hierarchy with added detail

Aggregation

- 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.

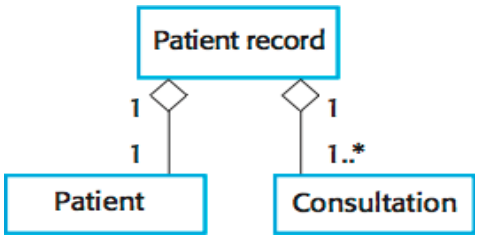

Fig: The Aggregation Association

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. Two types of stimuli:

- Some **data** arrives that has to be processed by the system.
- 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. Data-driven models can be created using UML **activity diagrams**:

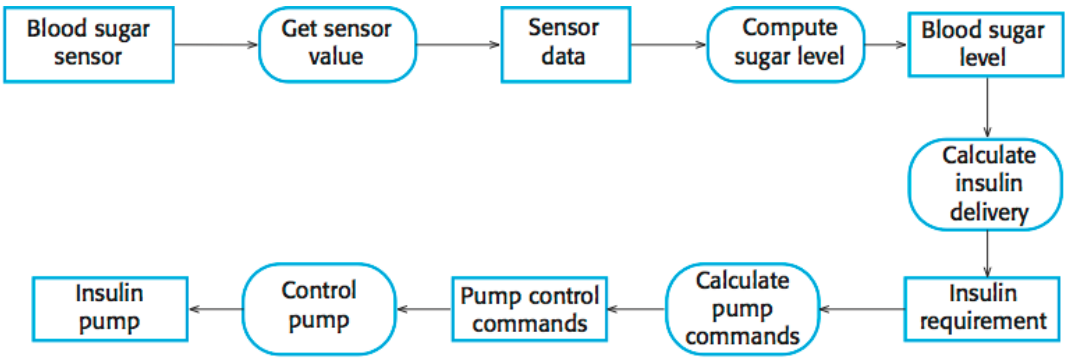

Fig: An activity model of the insulin pump's operation

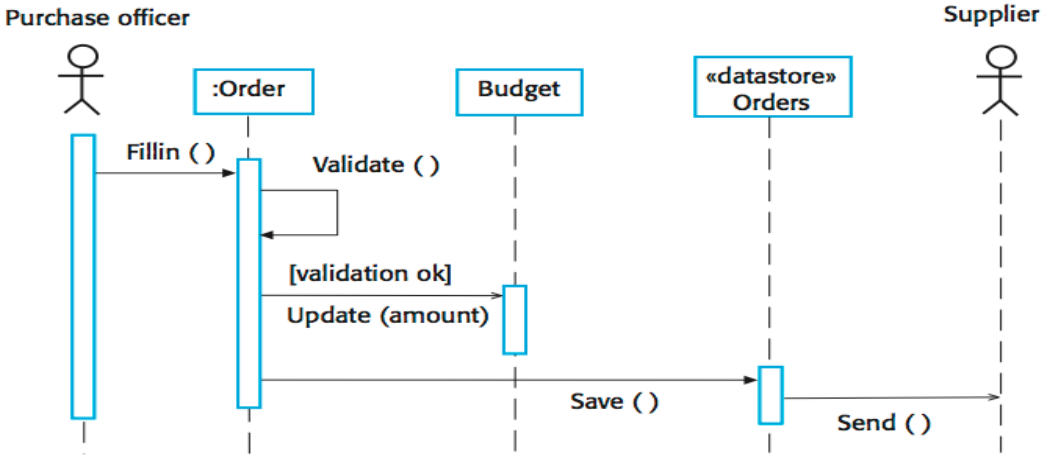

Fig: order processing

Event-driven models

- 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 models** 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. Event-driven models can be created using UML **state diagrams**:

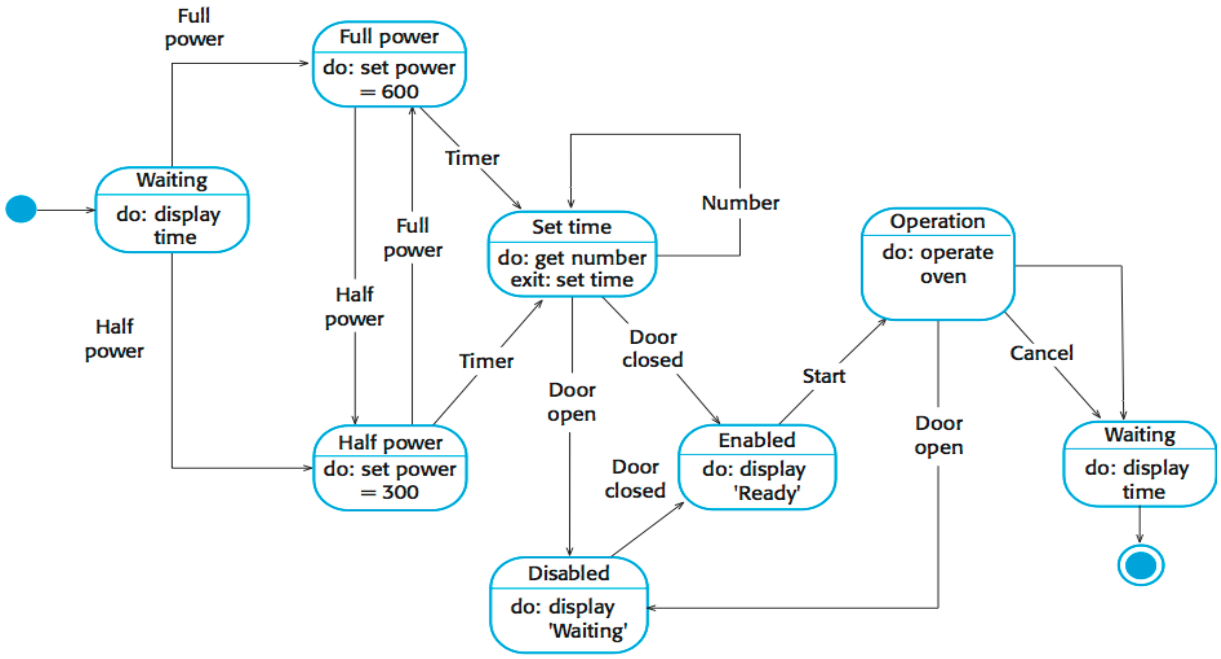

Fig: state diagram of a microwave oven

- In UML state diagrams, rounded rectangles represent system states.
- They may include a brief description (following 'do') of the actions taken in that state.
- The labelled arrows represent stimuli that force a transition from one state to another.
- You can indicate start and end states using filled circles, as in activity diagrams

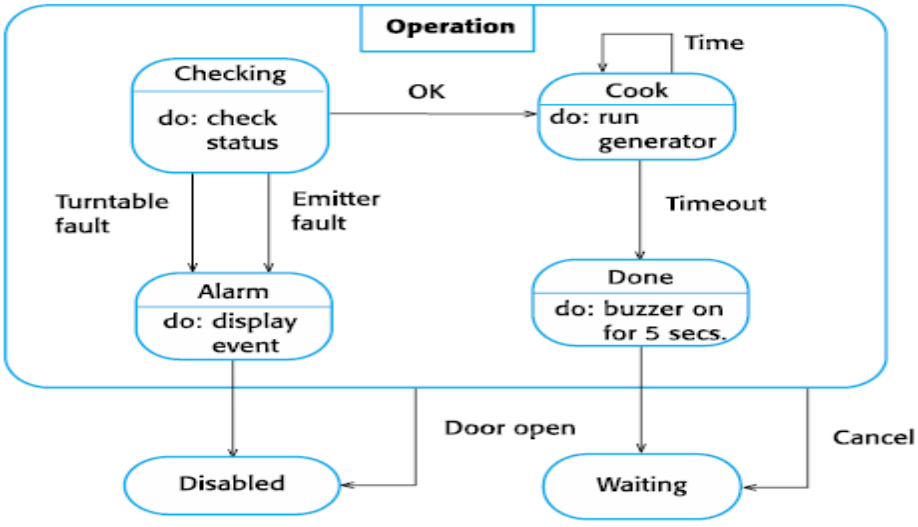

Fig: microwave oven operation

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.

Fig: States and stimuli for the microwave oven (a)

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: States and stimuli for the microwave oven (b)

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
- Engineers no longer have to be concerned with programming language details or the specifics of execution platforms.

- **The main arguments for and against MDE are:**
- *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
- By using powerful tools, system implementations can be generated for different platforms from the same model
- Therefore, to adapt the system to some new platform technology, it is only necessary to write a translator for that platform. When this is available, all platform-independent models can be rapidly rehosted on the new platform

- ***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
- Platform independence are only valid for large long-lifetime systems where the platforms become obsolete during a system's lifetime
- The MDE approach is used within large companies such as IBM and Siemens
- The techniques have been used successfully in the development of large, long-lifetime software systems such as air traffic management systems

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
- Here, 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
- **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.
- You may develop several different CIMs, reflecting different views of the system.

- For example, there may be a security CIM in which you identify important security abstractions such as an asset and a role and a patient record CIM, in which you describe abstractions such as patients, consultations, etc.
- 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.
- So, the first-level PSM could be middleware-specific but database independent.
- When a specific database has been chosen, a database specific PSM can then be generated.

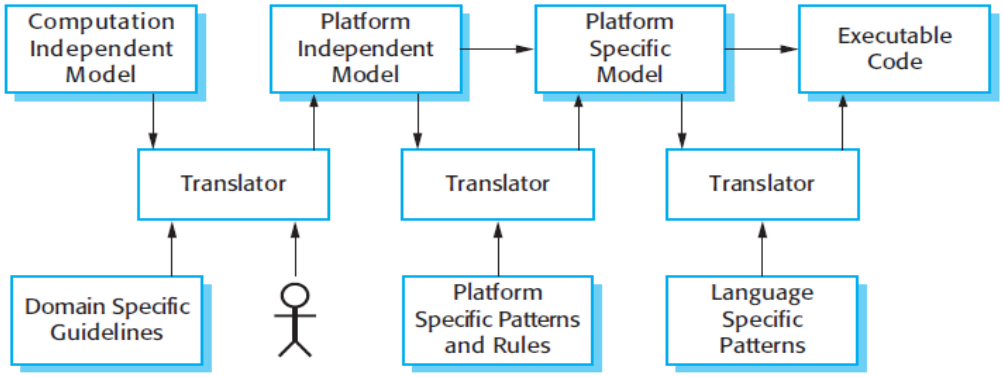

Fig: MDA transformation

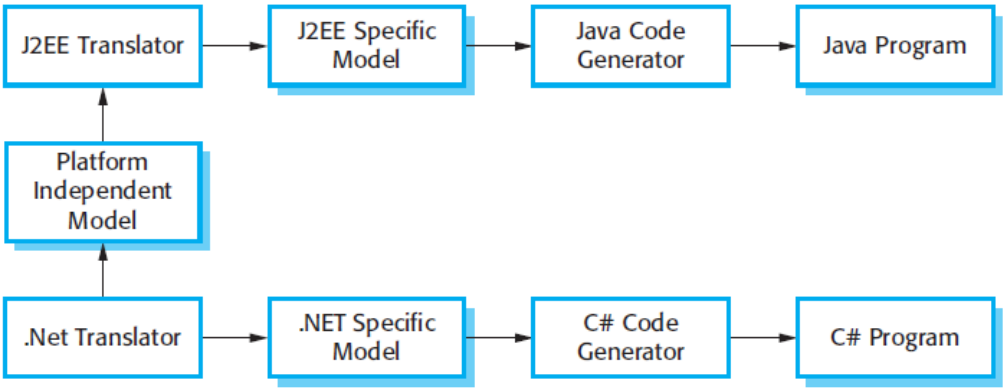

Fig: Multiple platform-specific models

- Although MDA-support tools include platform-specific translators, it is often the case that these will only offer partial support for the translation from PIMs to PSMs.
- In the vast majority of cases, the execution environment for a system is more than the standard execution platform (e.g., J2EE, .NET, etc.).
- It also includes other application systems, application libraries that are specific to a company, and user interface libraries
- As these vary significantly from one company to another, standard tool support is not available.
- Therefore, when MDA is introduced, special purpose translators may have to be created

Executable UML

- The fundamental notion behind model-driven engineering is that completely automated transformation of models to code should be possible.
- To achieve this, you have to be able to construct graphical models whose semantics are well defined
- You also need a way of adding information to graphical models about the ways in which the operations defined in the model are implemented
- This is possible using a subset of UML 2, called Executable UML or xUML
- 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:**
- Domain models identify the principal concerns in the system. These are defined using UML class diagrams that 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 lifecycle of the class.

Design and Implementation

- **Object-Oriented design using the UML**
- **Design patterns**
- **Implementation issues**
- **Open source development**

****

- Software design and implementation is the stage in the software engineering process at which an executable software system is developed
- Software design and implementation activities are invariably interleaved.
- Software design is a creative activity in which you identify software components and their relationships, based on a customer's requirements.
- Implementation is the process of realizing the design as a program
- Design and implementation are closely linked and you should normally take implementation issues into account when developing a design
- For example, using the UML to document a design may be the right thing to do if you are programming in an object-oriented language such as Java or C#.
- It is less useful, if you are developing in a dynamically typed language like Python
- One of the most important implementation decisions that has to be made at an early stage of a software project is whether or not you should buy or build the application software.
- In a wide range of domains, it is now possible to buy off-the-shelf systems (COTS) that can be adapted and tailored to the users' requirements.
- For example, if you want to implement a medical records system, you can buy a package that is already used in hospitals. It can be cheaper and faster to use this approach rather than developing a system in a conventional programming language

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
- The representation of the state is private and cannot be accessed directly from outside the object
- Object-oriented design processes involve designing object classes and the relationships between these classes
- These classes define the objects in the system and their interactions
- Object-oriented systems are easier to change than systems developed using functional approaches
- Objects include both data and operations to manipulate that data

- Changing the implementation of an object or adding services should not affect other system objects
- Because objects are associated with things, there is often a clear mapping between real-world entities and their controlling objects in the system. This improves the understandability, and hence the maintainability, of the design
- **To develop a system design from concept to detailed, object-oriented design, there are several things that you need to do:**

    1. Understand and define the context and the external interactions with the system
    2. Design the system architecture
    3. Identify the principal objects in the system
    4. Develop design models
    5. Specify interfaces

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:

    1. A **system context model** is a structural model that demonstrates the other systems in the environment of the system being developed.
    2. 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
- You may document the environment of the system using a simple block diagram, showing the entities in the system and their association

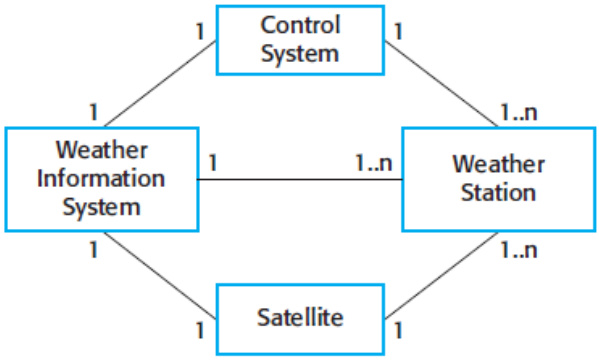

Fig: system context for the weather station

- When you model the interactions of a system with its environment you should use an abstract approach that does not include too much detail.
- One way to do this is to use a use case model
- Each possible interaction is named in an ellipse and the external entity involved in the interaction is represented by a stick figure

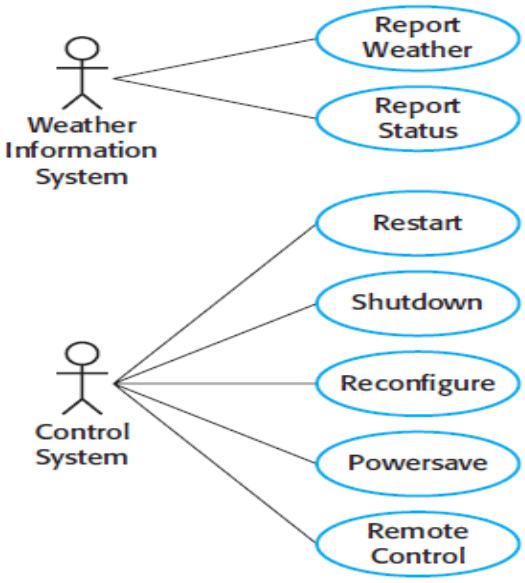

Fig: weather station use cases

- Each of these use cases should be described in structured natural language.
- This helps designers identify objects in the system and gives them an understanding of what the system is intended to do.
- The standard format for this description that clearly identifies what information is exchanged, how the interaction is initiated, and so on

Architectural design

- Once the interactions between the software system and the system's environment have been defined, you use this information as a basis for designing the system architecture.
- Of course, you need to combine this with your general knowledge of the principles of architectural design and with more detailed domain knowledge
- You identify the major components that make up the system and their interactions, and then may organize the components using an architectural pattern such as a layered or client–server model

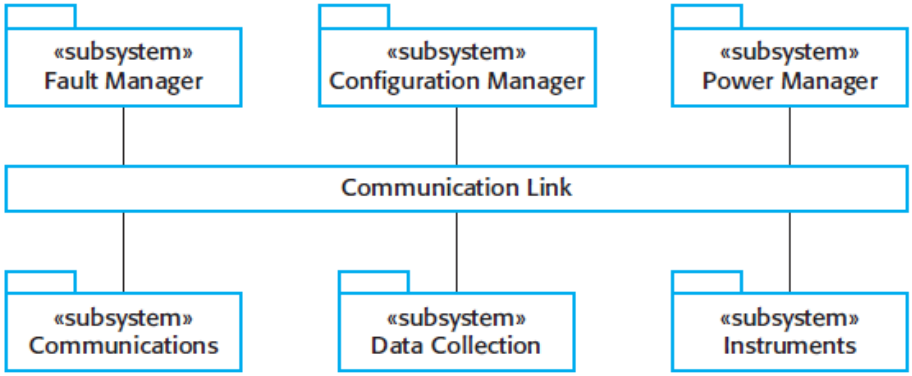

Fig: high level architecture of the weather station

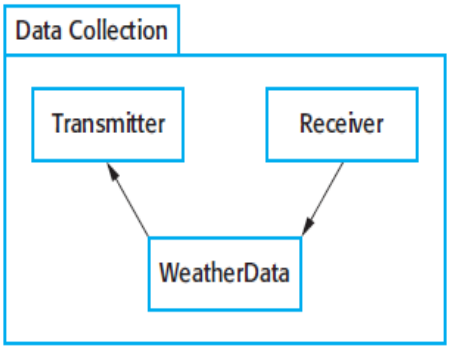

Fig: Architecture of data collection system

Object class identification

- By this stage in the design process, you should have some ideas about the essential objects in the system that you are designing.
- As your understanding of the design develops, you refine these ideas about the system objects.
- The use case description helps to identify objects and operations in the system

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

    1. 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
    2. 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
    3. Use a scenario-based analysis where various scenarios of system use are identified and analyzed in order to identify objects, attributes and operations

- In practice, you have to use several knowledge sources to discover object classes.
- Initially from the informal system description
- From application domain knowledge or scenario analysis
- This information can be collected from requirements documents, discussions with users, or from analyses of existing systems.

Weather station objects

- 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:

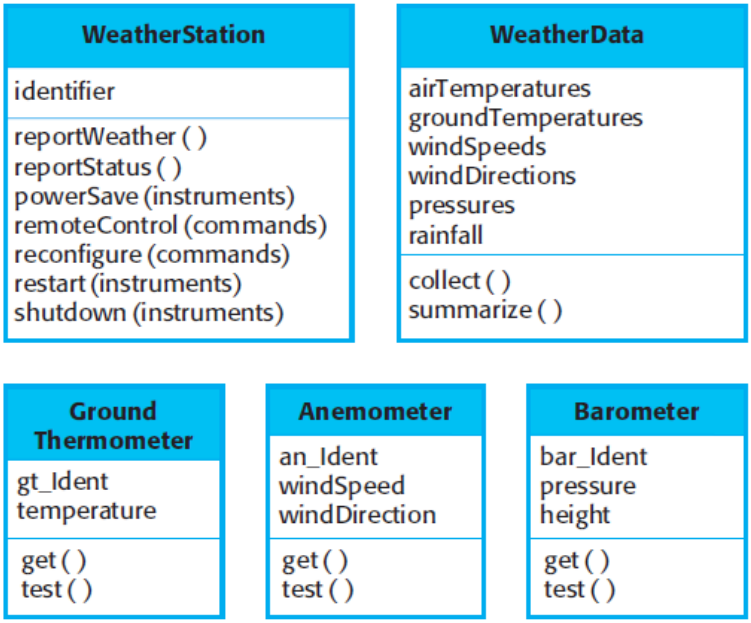

Fig: weather station objects

- Focus on identifying the objects, not on their implementation
- Once you have identified the objects, you then refine the object design
- Look for common features to achieve inheritance

Example

- Instrument super-class – defines the common features of all instruments
- Attributes like identifier
- Operations like get and test
- You can also add new attributes and operations

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
- Should be abstract to hide unnecessary details, but they should include enough details for programmers to make implementation decisions
- When you use the UML to develop a design, you will normally develop **two kinds of design model:**

1. **Structural models**

- 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**

- Describe the dynamic structure of the system and show the interactions between the system objects
- 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:** 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) models.

2. **Sequence models:** 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:** 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.

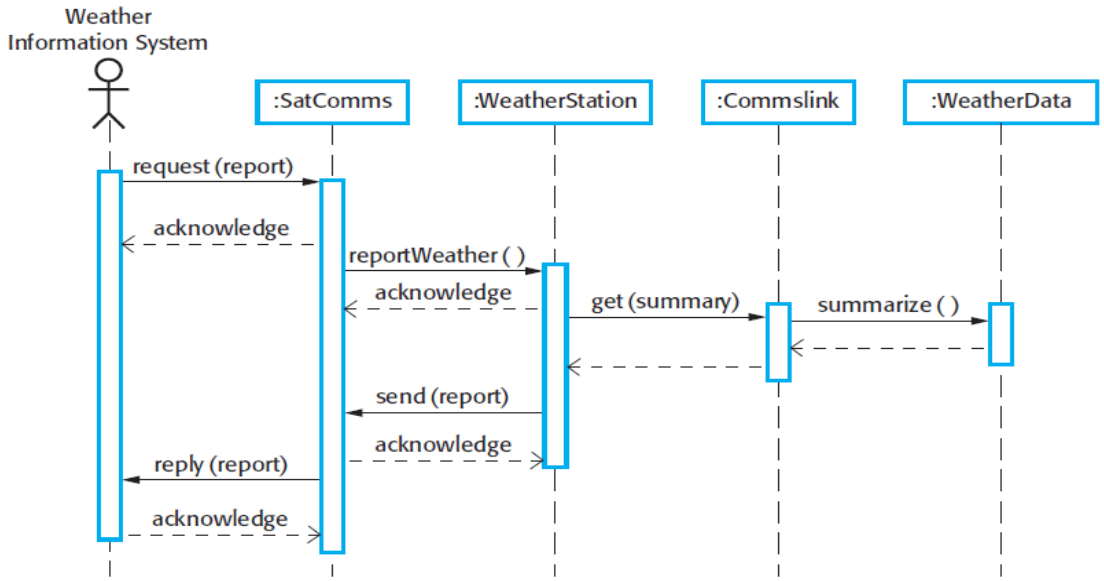

Fig: sequence diagram describing data collection

This diagram shows the sequence of interactions that takes place when an external system requests the summarized data from the weather station. You read sequence diagram from top to bottom:

1. The SatComms object receives a request from he 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 send 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 transit the summarized data to the weather information system, through the satellite communications system.

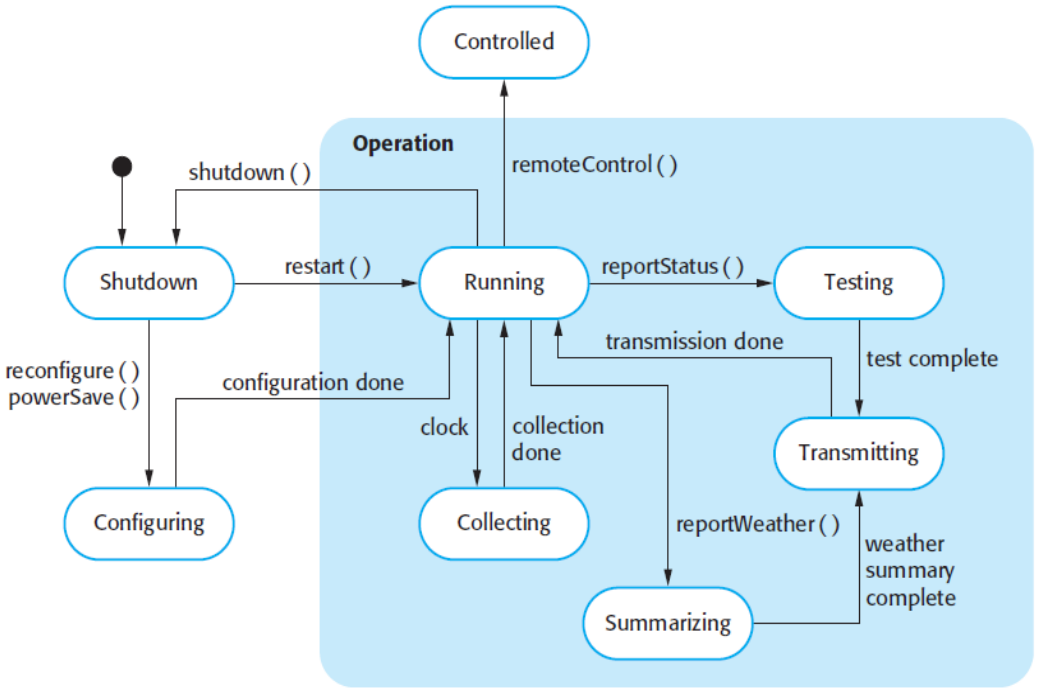

Fig: weather station state diagram

1. If the system state is Shutdown then it can respond to a restart(), a reconfigure(), or a PowerSave() message.
2. In the Running state, the system expects further message. 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. Is a reportStatus() message is received, the system moves to the Testing state, then the Transmitting state, before returning to the Running state.
5. If the signal from the clock is received, the system moves to the Collecting state, where it collects data from the instruments.
6. If a remoteControl() message is received, the system moves to a control state in which its responds to a different set of messages from the remote control room,

Interface Specification

- An important part of any design process is the specification of the interfaces between the components in the design
- 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.
- However, there is no attribute section and the UML stereotype ‹‹interface›› should be included in the name part
- You should not include details of the data representation in an interface design, as attributes are not defined in an interface specification
- However, you should include operations 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.
- For example, an array representation of a stack may be changed to a list representation without affecting other objects that use the stack

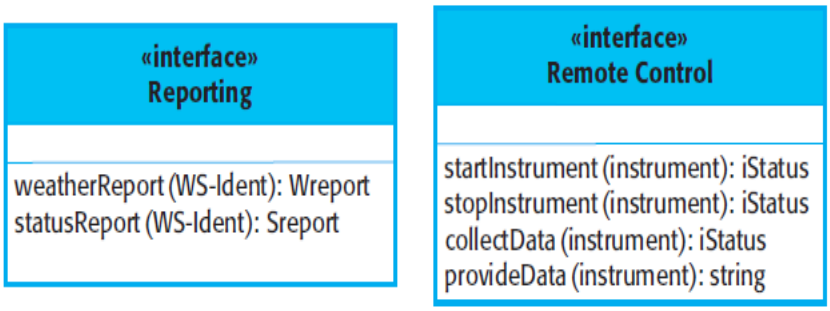

Fig: weather station interface

Design patterns

- Design patterns were derived from ideas put forward by Christopher Alexander who suggested that there were certain common patterns of building design that were inherently pleasing and effective
- 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
- Rather, you can think of it as a description of accumulated wisdom and experience, a well-tried solution to a common problem
- ***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.
- Published patterns often rely on object characteristics such as inheritance and polymorphism to provide generality
- The four essential elements of design patterns were defined by the **'Gang of Four'** in their patterns book:

1. A **name** that is a meaningful reference to the pattern
2. A **description** of the problem area that explains when the pattern may be applied.
3. 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
4. A **statement of the consequences**—the results and trade-offs

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: the observer pattern

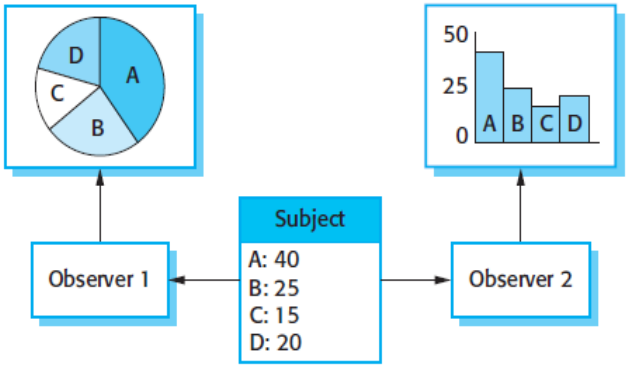

Fig: multiple display

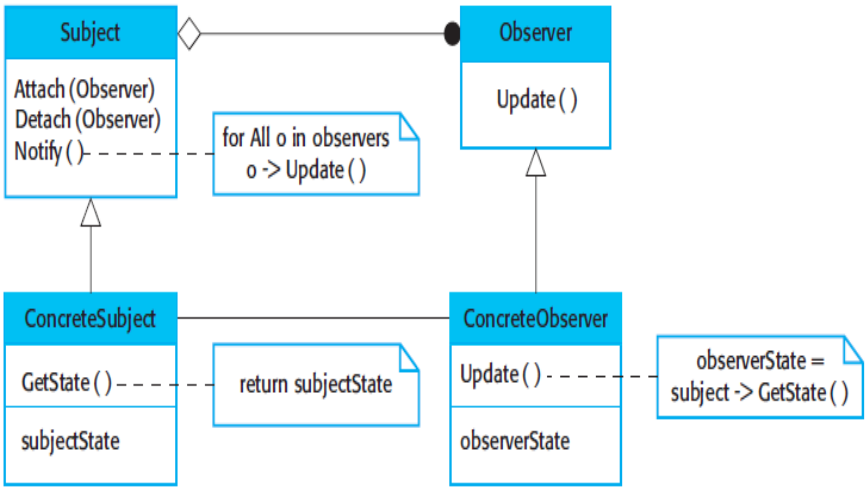

Fig: A UML model of the observer pattern

- Patterns support high-level, concept reuse.
- When you try to reuse executable components you are inevitably constrained by detailed design decisions that have been made by the implementers of these components.
- These range from the particular algorithms that have been used to implement the components to the objects and types in the component interfaces.
- When these design decisions conflict with your particular requirements, reusing the component is either impossible or introduces inefficiencies into your system.
- Using patterns means that you reuse the ideas but can adapt the implementation to suit the system that you are developing.

Implementation Issues

- Software engineering includes all of the activities involved in software development from the initial requirements of the system through to maintenance and management of the deployed system.
- A critical stage of this process is, of course, system implementation, where you create an executable version of the software.
- 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.
- Some aspects of implementation that are particularly important to software engineering

1. ***Reuse***

- Most modern software is constructed by reusing existing components or systems.
- When you are developing software, you should make as much use as possible of existing code.

2. ***Configuration management***

- *During the development process, many different* versions of each software component are created.
- If you don't keep track of these versions in a configuration management system, you are liable to include the wrong versions of these components in your system

3. ***Host-target development***

- *Production software does not usually execute on the* same computer as the software development environment.
- Rather, you develop it on one computer (the host system) and execute it on a separate computer (the target system).
- The host and target systems are sometimes of the same type but, often they are completely different

REUSE

- From the 1960s to the 1990s, most new software was developed from scratch, by writing all code in a high-level programming language.

- The only significant reuse or software was the reuse of functions and objects in programming language libraries
- However, costs and schedule pressure meant that this approach became increasingly unviable, especially for commercial and Internet-based systems
- Software reuse is possible at a number of **different levels:**

1. ***The abstraction level***

- *At this level, you don't reuse software directly but rather* use 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, you directly reuse objects 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.
- For example, if you need to process mail messages in a Java program, you may use objects and methods from a JavaMail library

3. ***The component level***

- Components are collections of objects and object classes that operate together to provide related functions and services.
- You often have to adapt and extend the component by adding some code of your own.
- An example of component-level reuse is where you build your user interface using a framework.
- This is a set of general object classes that implement event handling, display management, etc.
- You add connections to the data to be displayed and write code to define specific display details such as screen layout and colors

4. ***The system level***

- *At this level, you reuse entire application systems.*

- *This usually* involves some kind of configuration of these systems.
- This may be done by adding and modifying code 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.
- Sometimes this approach may involve reusing several different systems and integrating these to create a new system

Advantages

- By reusing existing software, you can develop new systems more quickly, with fewer development risks and also lower costs.
- As the reused software has been tested in other applications, it should be more reliable than new software

Disadvantages/costs

- The costs of the time spent in looking for software to reuse and assessing whether or not it meets your needs. You may have to test the software to make sure that it will work in your 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 you have developed. Integrating reusable software from different providers can be difficult and expensive because the providers may make conflicting assumptions about how their respective software will be reused.

Configuration management

- In software development, change happens all the time, so change management is absolutely essential.
- When a team of people are developing software, you have to make sure that team members don't interfere with each other's' work.
- That is, if two people are working on a component, their changes have to be coordinated.

- Otherwise, one programmer may make changes and overwrite the other's work.
- You also have to ensure that everyone can access the most up-to-date versions of software components, otherwise developers may redo work that has already been done
- 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 three fundamental configuration management activities:**

    1. **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
    2. **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.
    3. **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.

- Software configuration management tools support each of the above activities
- Version management may be supported using a version management system such as **Subversion**
- System integration support may be built into the language or rely on a separate toolset such as the **GNU** build system
- Bug tracking or issue tracking systems, such as **Bugzilla**, are used to report bugs and other issues and to keep track of whether or not these have been fixed

Host-target development

- Most software development is based on a host-target model.
- Software is developed on one computer (the host), but runs on a separate machine (the target).
- More generally, we can talk about a development platform and an execution platform.

- 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.
- Sometimes, the development and execution platforms are the same, making it possible to develop the software and test it on the same machine.
- More commonly, however, they are different so that you need to either move your developed software to the execution platform for testing or run a simulator on your development machine.
- Simulators are often used when developing embedded systems.
- You simulate hardware devices, such as sensors, and the events in the environment in which the system will be deployed.
- Simulators speed up the development process for embedded systems as each developer can have their own execution platform with no need to download the software to the target hardware.
- However, simulators are expensive to develop and so are only usually available for the most popular hardware architectures
- If the target system has installed middleware or other software that you need to use, then you need to be able to test the system using that software.
- It may be impractical to install that software on your development machine, even if it is the same as the target platform, because of license restrictions.
- In those circumstances, you need to transfer your developed code to the execution platform to test the system
- A software development platform should provide a range of tools to support software engineering processes. **These may include:**

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

- As well as these standard tools, your development system may include more specialized tools such as static analyzers
- Normally, development environments for teams also include a shared server that runs a change and configuration management system and, perhaps, a system to support requirements management
- 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.
- Generally, IDEs are created to support development in a specific programming language such as Java.
- The language IDE may be developed specially, or may be an instantiation of a general-purpose IDE, with specific language-support tools.
- 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.
- The best-known general-purpose IDE is the Eclipse environment.
- This environment is based on a plug-in architecture so that it can be specialized for different languages and application domains
- As part of the development process, you need to make decisions about how the developed software will be deployed on the target platform
- Issues that you have to consider in making this decision are:

1. ***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

2. ***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.

3. ***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.

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
- Its roots are in the Free Software Foundation (http://www.fsf.org), which advocates that source code should not be proprietary but rather should always be available for users to examine and modify as they wish.
- There was an assumption that the code would be controlled and developed by a small core group, rather than users of the code
- Open source software extended this idea by using the Internet to recruit a much larger population of volunteer developers. Many of them are also users of the code.
- In principle at least, any contributor to an open source project may report and fix bugs and propose new features and functionality.
- However, in practice, successful open source systems still rely on a core group of developers who control changes to the software
- LINUX, JAVA, the APACHE Web Server, mySQL dbms
- IBM and Sun
- It is usually fairly cheap or free to acquire open source software.
- You can normally download open source software without charge.
- However, if you want documentation and support, then you may have to pay for this, but costs are usually fairly low.
- The other key benefit of using open source products is that mature open source systems are usually very reliable.

- The reason for this is that they have a large population of users who are willing to fix problems themselves rather than report these problems to the developer and wait for a new release of the system
- Bugs are discovered and repaired more quickly than is usually possible with proprietary software.
- For a company involved in software development, there are two open source issues that have to be considered:
    1. Should the product that is being developed make use of open source components?
    2. 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
- If you are developing a software product for sale, then time to market and reduced costs are critical. If you are developing in a domain in which there are high-quality open source systems available, you can save time and money by using these systems
- However, if you are developing software to a specific set of organizational requirements, then using open source components may not be an option
- Companies believe that involving the open source community will allow software to be developed more cheaply, more quickly, and will create a community of users for the software.
- Many companies believe that adopting an open source approach will reveal confidential business knowledge to their competitors and so are reluctant to adopt this development model

Open source licensing

- Although a fundamental principle of open-source development is that source code should be freely available, this does not mean that anyone can do as they wish with that code.
- Legally, the developer of the code (either a company or an individual) still owns the code.
- They can place restrictions on how it is used by including legally binding conditions in an open source software license

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 you can write components that link to open source code without having to publish the source of these components.
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.

Bayersdorfer suggests that companies managing projects that use open source should:

1. Establish a system for maintaining information about open source components that are downloaded and used. You have to keep a copy of the license for each component that was valid at the time the component was used. Licenses may change so you need to know the conditions that you have agreed to.
2. Be aware of the different types of licenses and understand how a component is licensed before it is used.
3. Be aware of evolution pathways for components. You need to know a bit about the open source project where components are developed to understand how they might change in future.
4. Educate people about open source. It's not enough to have procedures in place to ensure compliance with license conditions. You also need to educate developers about open source and open source licensing
5. Have auditing systems in place. Developers, under tight deadlines, might be tempted to break the terms of a license. If possible, you should have software in place to detect and stop this
6. Participate in the open source community. If you rely on open source products, you should participate in the community and help support their development