Module 3

System Models: Context models (Sec 5.1). Interaction models (Sec 5.2). Structural models (Sec 5.3). Behavioral models (Sec 5.4). Model-driven engineering (Sec 5.5).

Design and Implementation: Introduction to RUP (Sec 2.4), Design Principles (Chap 17). Object-oriented design using the UML (Sec 7.1). Design patterns (Sec 7.2). Implementation issues (Sec 7.3). Open source development (Sec 7.4).

Introduction

- ★ System modeling is the process of developing abstract models of a system, with each model presenting a different view or perspective of that system.
- ★ Generally, system model is represent using unified modeling language (UML)
- ★ Models of development
  - ✓ Models of the existing system are used during requirements engineering. They help clarify what the existing system does and can be used as a basis for discussing its strengths and weaknesses. These then lead to requirements for the new system.
  - ✓ Models of the new system are used during requirements engineering to help explain the proposed requirements to other system stakeholders. Engineers use these models to discuss design proposals and to document the system for implementation. In a model-driven engineering process, it is possible to generate a complete or partial system implementation from the system model.
- ★ Different perspectives models
  - o An ***external perspective***, where you model the context or environment of the system.
  - o An ***interaction perspective*** where you model the interactions between a system and its environment or between the components of a system.
  - o A ***structural perspective***, where you model the organization of a system or the structure of the data that is processed by the system.
  - o A ***behavioral perspective***, where you model the dynamic behavior of the system and how it responds to events.
- ★ UML five diagram types
  - o ***Activity diagrams***, which show the activities involved in a process or in data processing.
  - o ***Use case diagrams***, which show the interactions between a system and its environment.
  - o ***Sequence diagrams***, which show interactions between actors and the system and between system components.
  - o ***Class diagrams,*** which show the object classes in the system and the associations between these classes.
  - o ***State diagrams***, which show how the system reacts to internal and external events

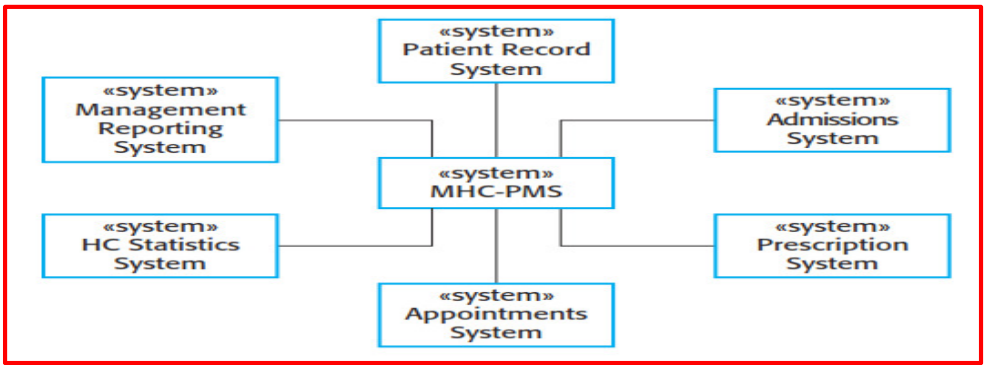
Context models

- ★ 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
  - o The definition of a system boundary is not a value-free judgment. Social and organizational concerns may mean that the position of a system boundary may be determined by non-technical factors
    - For example, a system boundary may be positioned so that the analysis process can all be carried out on one site; it may be chosen so that a particularly difficult manager need not be consulted; it may be positioned so that the system cost is increased, and the system development division must therefore expand to design and implement the system
- ★ Architectural models show the system and its relationship with other systems

System Boundary

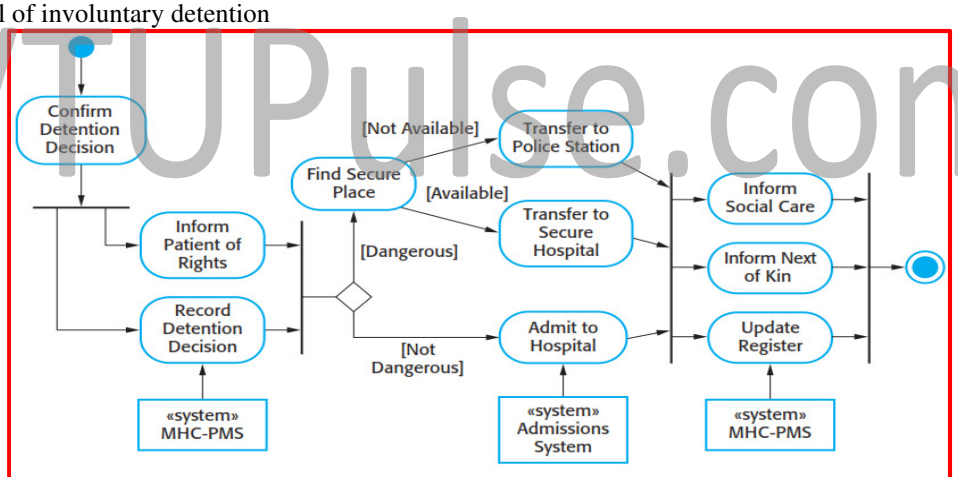
- ★ System boundaries are established to define what is inside and what is outside the system
  - o They show other systems that are used or depend on the system being developed.
- ★ The position of the system boundary has a profound effect on the system requirements.
- ★ Defining a system boundary is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

★ The context of the MHC-PMS (A patient management system for mental health care) system



- MHC-PMS is connected to an appointments system and a more general patientrecord 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.
- Context models normally show that the environment includes several other automated systems.
- However, they do not show the types of relationships between the systems in the environment and the system that is being specified.
- External systems might produce data for or consume data from the system. They might share data with the system, or they might be connected directly, through a network or not connected at all.
- They might be physically co-located or located in separate buildings. All of these relations may affect the requirements and design of the system being defined and must be taken into account.
- Simple context models are used along with other models, such as business process models. These describe human and automated processes in which particular software systems are used.

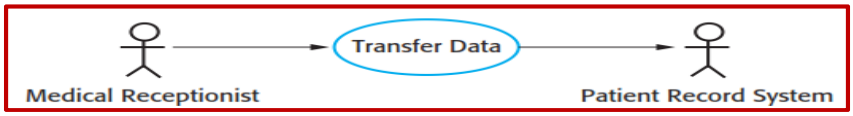
★ Process model of involuntary detention



- Activity diagrams are intended to show theactivities that make up a system process and the flow of control from one activity toanother.
- 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.
- 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.
- The activities to inform social care and the patient's next of kin and to update the detention register may be concurrent.
- 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.
- Patients who are suicidal and so are a danger to them may be detained in an appropriate ward in a hospital.

Interaction models

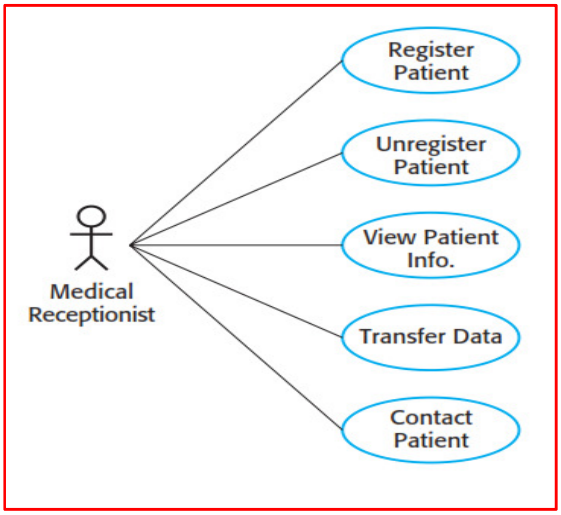
- ★ This can be user interaction model.
- ★ It involves user inputs and outputs, interaction between the systems being developedand other systems or interaction between the components of the system.
- ★ The user interaction is to identify user requirements and understand if proposed system structures like input and output.
- ★ There are two interaction modeling:
  - o Use case modeling, which is mostly used to model interactions between a system and external actors (users or other systems).
  - o Sequence diagrams, which are used to model interactions between system components, although external agents may also be included.
  - ➢ Use case modeling
    - o Use cases were developed originally to support requirements elicitation and now incorporated into the UML
    - o Each use case represents a discrete task that involves external interaction with a system.
    - o Actors in a use case may be people or other systems.
    - o Represented diagrammatically to provide an overview of the use case and in a more detailed textual form.
    - o Use case is shown as an ellipse with the actors involved in the use case represented as stick



Tabular description of the 'Transfer data' use case

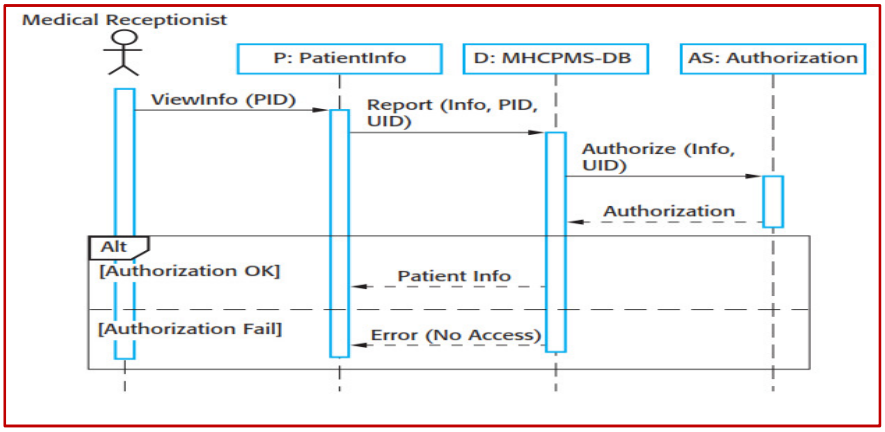
MHC-PMS: Transfer data	
Actors	Medical receptionist, patient records system (PRS)
Description	A receptionist may transfer data from the MHC-PMS to a general patient record database that is maintained by a health authority. The information transferred may either be updated personal information (address,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

Use cases involving the role 'medical receptionist'



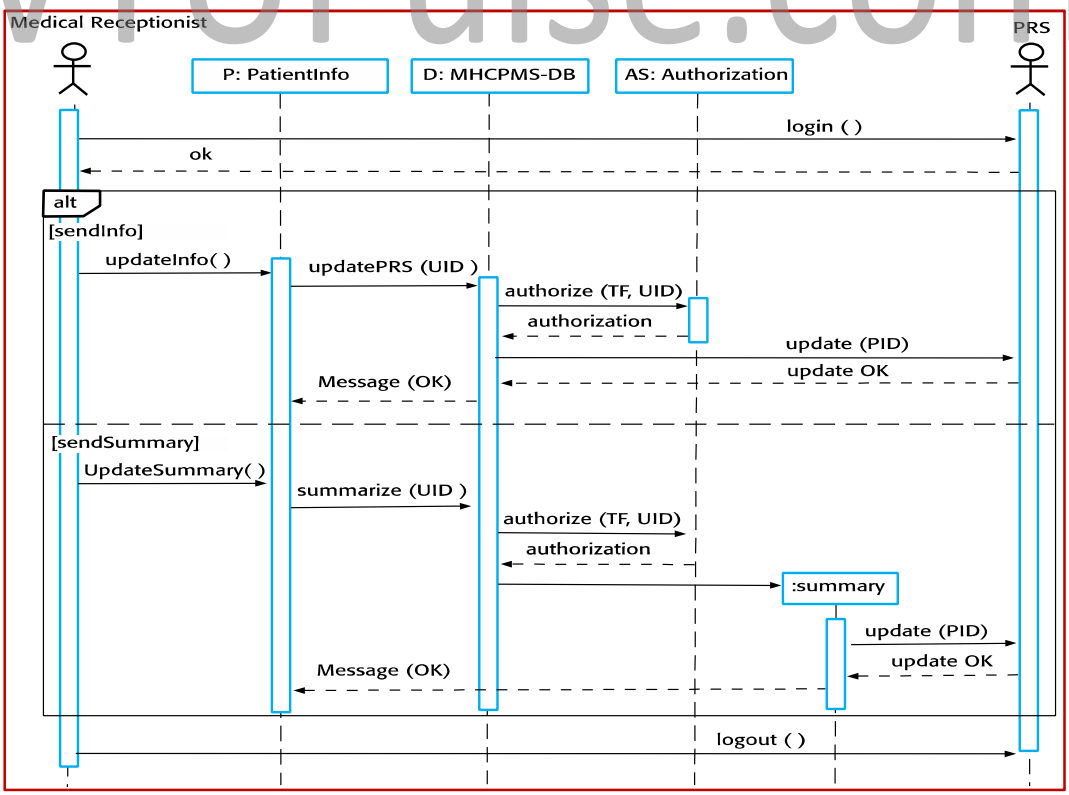
Sequence diagrams

- ✧ Sequence diagrams are part of the UML and are used to model the interactions between the actors and the objects within a system.
- ✧ A sequence diagram shows the sequence of interactions that take place during a particular use case or use case instance.
- ✧ The objects and actors involved are listed along the top of the diagram, with a dotted line drawn vertically from these.
- ✧ Interactions between objects are indicated by annotated arrows (calls to the objects, their parameters, and the return values).
- ✧ A box named alt is used with the conditions indicated in square brackets.
- ✧ Sequence diagram for View patient information as follows:



1. The medical receptionist triggers the ViewInfo method in an instance P of thePatientInfo object class, supplying the patient's identifier, PID. P is a user interfaceobject, which is displayed as a form showingpatientinformation.
2. The instance P calls the database to return the information required, supplyingthe receptionist's identifier to allow security checking (at this stage, we do notcare where this UID comes from).
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.

Sequence diagram for transfer data



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.

Structural models

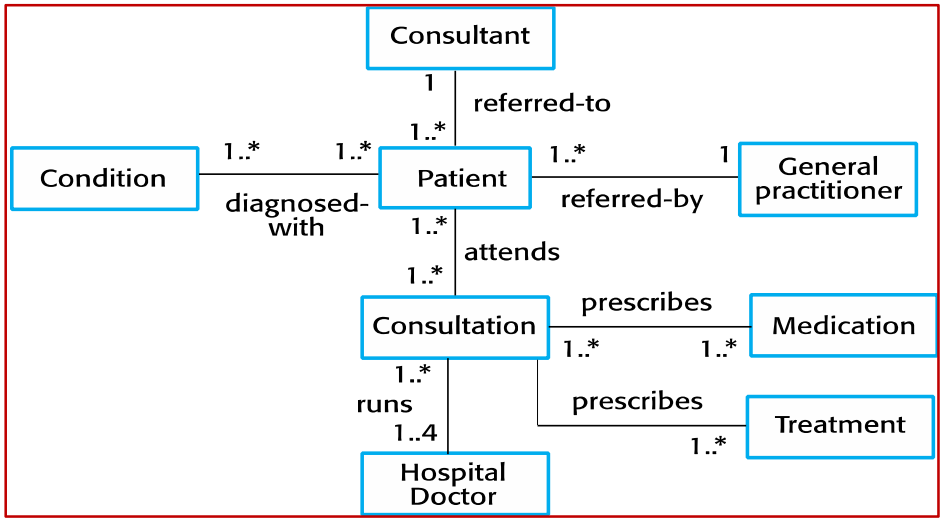
1. Structural models of software display the organization of a system in terms of the components that make up that system and their relationships
2. Structural models may be static models, which show the structure of the system design, or dynamic models, which show the organization of the system when it is executing.
3. 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.
- ✧ Example: UML classes and association as shown below

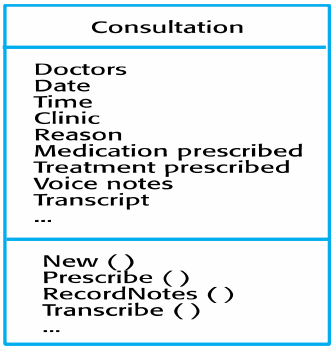
  - ○ Each end of the association is annotated with a 1, meaning that there is a 1:1 relationship between objects of these classes. That is, each patient has exactly one record and each record maintains information about exactly one patient.

- ✧ Classes and associations in the MHC-PMS



- Name associations to give the reader an indication of the type ofrelationship that exists.
- The UML also allows the role of the objects participating in the association to be specified.
- Class diagrams look like semantic data models which are used in database design.
- They show the data entities, their associated attributes, and the relations between these entities.
- The UML does not include a specific notation for this database modeling as it assumes an object-oriented development process and models data using objects and their relationships.

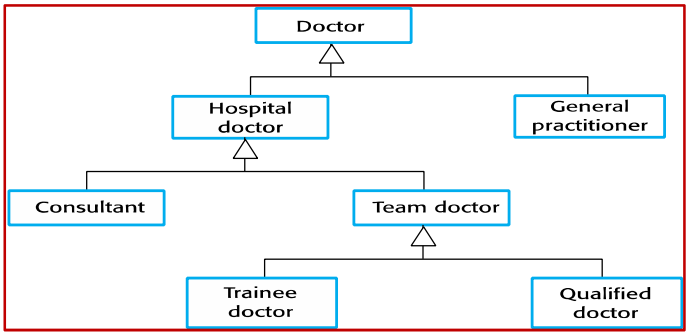
The Consultation class



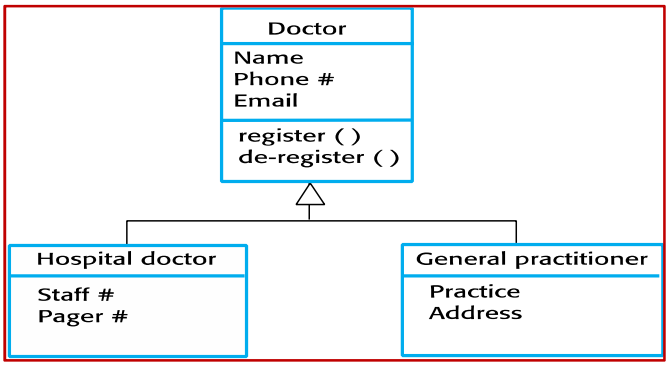
1. The name of the object class is in the top section.
2. The class attributes are in the middle section. This must include the attribute names and, optionally, their types.
3. The operations (called methods in Java and other OO programming languages) associated with the object class are in the lower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

Generalization

- ✧ Generalization is an everyday technique that we use to manage complexity.
- ✧ Rather than learn the detailed characteristics of every entity that we experience, we place these entities in more general classes (animals, cars, houses, etc.) and learn the characteristics of these classes.
- ✧ This allows us to infer that different members of these classes have some common characteristics e.g. squirrels and rats are rodents.
- ✧ In modeling systems, it is often useful to examine the classes in a system to see if there is scope for generalization. If changes are proposed, then you do not have to look at all classes in the system to see if they are affected by the change.
- ✧ In object-oriented languages, such as Java, generalization is implemented using the class inheritance mechanisms built into the language.
- ✧ In a generalization, the attributes and operations associated with higher-level classes are also associated with the lower-level classes.
- ✧ The lower-level classes are subclasses inherit the attributes and operations from their super classes. These lower-level classes then add more specific attributes and operations.
- ✧ Example: A generalization hierarchy

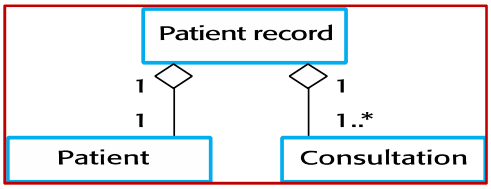


- ✧ A generalization hierarchy with added detail as shown below the diagram

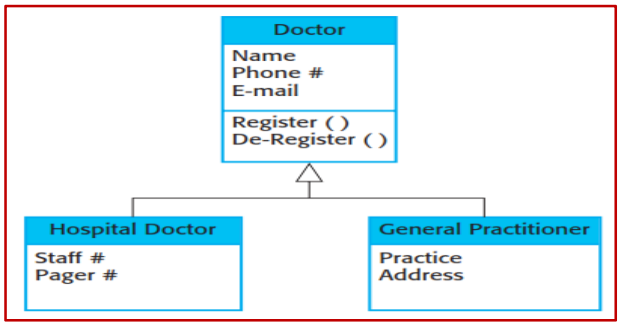


Object class aggregation models

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



- 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.



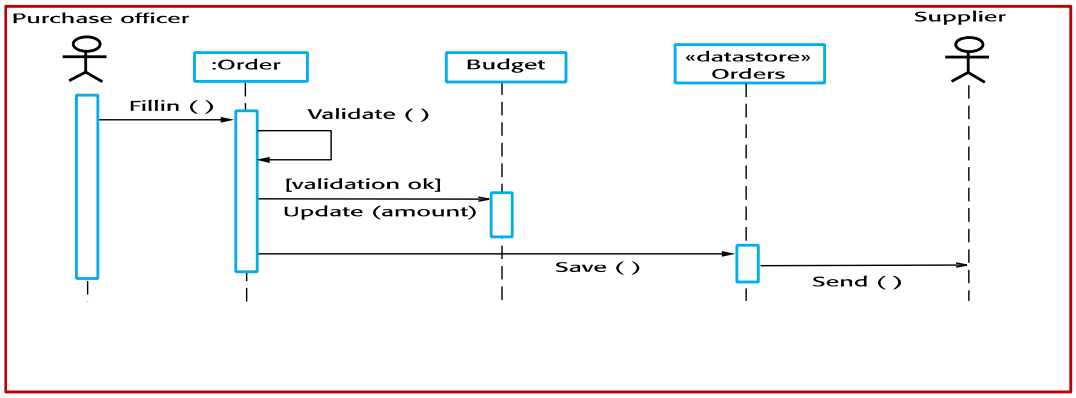
- The operations associated with the class Doctor are intended to register and de-register that doctor with the MHC-PMS.

Behavioral models

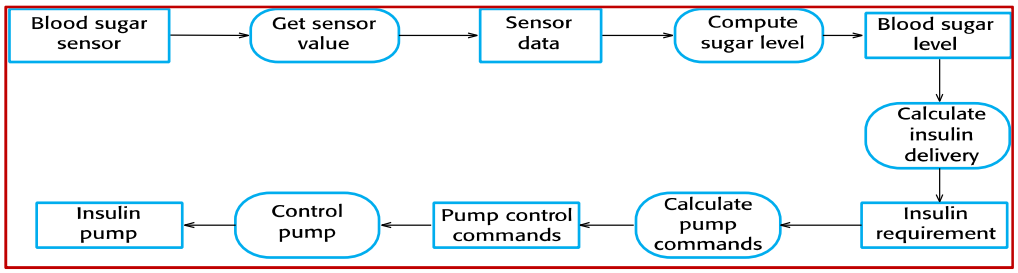
- ✧ Behavioral models are models of the dynamic behavior of a system as it is executing. They show what happens or what is supposed to happen when a system responds to a stimulus from its environment
- ✧ You can think of these stimuli as being of two types:
  - Data: Some data arrives that has to be processed by the system.
  - Events: Some event happens that triggers system processing. Events may have associated data, although this is not always the case.

Data-driven modeling

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



- ✧ Example: An activity model of the insulin pump's operation



Event-driven modelling

- ✧ 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' bygenerating a dial tone.
- ✧ Event-driven modeling shows how a system responds to external and internal events.
- ✧ It is based on the assumption that a system has a finite number of states and that events (stimuli) may cause a transition from one state to another.

State machine models

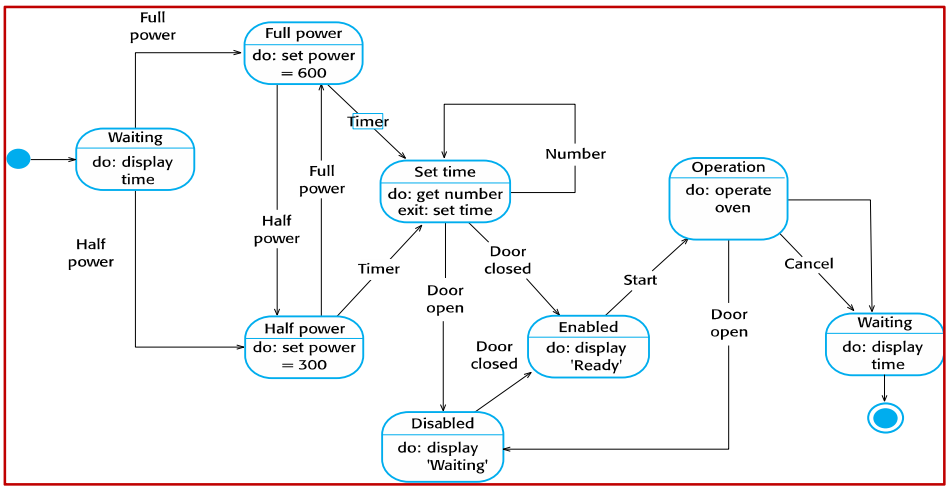
- ✧ These model the behaviour of the system in response to external and internal events.
- ✧ They show the system's responses to stimuli so are often used for modelling real-time systems.
- ✧ State machine models show system states as nodes and events as arcs between these nodes. When an event occurs,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

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

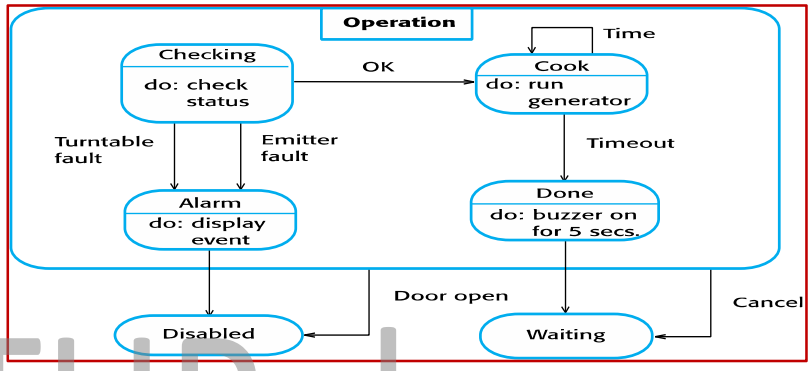
- ✧ States and stimuli for the microwave oven (b)

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

- ✧ State diagram of a microwave oven



Microwave oven operation



- ✧ The Operation state includes a number of sub-states.
- ✧ It shows that operation starts with a status check and that if any problems are discovered an alarm is indicated and operation is disabled.
- ✧ Cooking involves running the microwave generator for the specified time; on completion, a buzzer is sounded.
- ✧ If the door is opened during operation, the system moves to the disabled state.

Model-driven engineering

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

Usage of model-driven engineering

- ✧ Model-driven engineering is still at an early stage of development, and it is unclear whether or not it will have a significant effect on software engineering practice.
- ✧ Pros
  - Allows systems to be considered at higher levels of abstraction
  - Generating code automatically means that it is cheaper to adapt systems to new platforms.
- ✧ Cons
  - Models for abstraction and not necessarily right for implementation.
  - Savings from generating code may be outweighed by the costs of developing translators for new platforms.

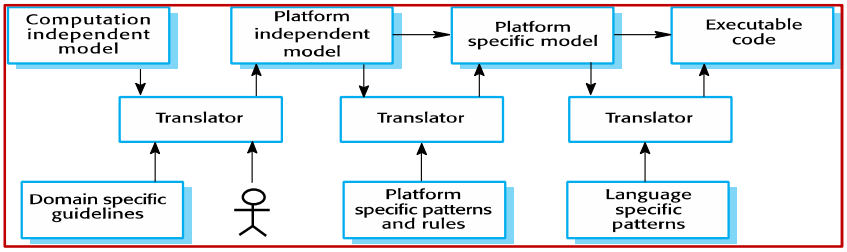
Model driven architecture

- ✧ Model-driven architecture (MDA) was the precursor of more general model-driven engineering
- ✧ MDA is a model-focused approach to software design and implementation that uses a subset of UML models to describe a system.

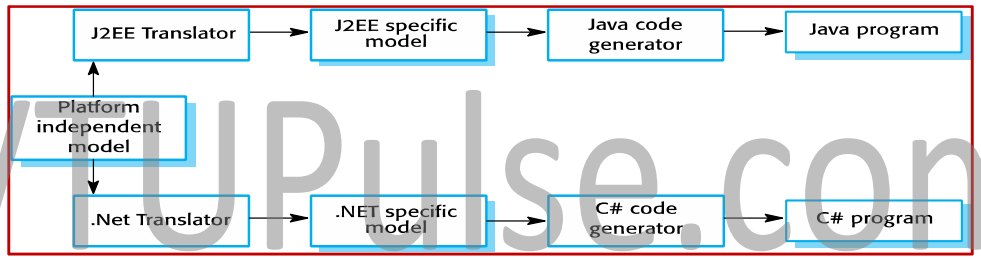
- ✧ Models at different levels of abstraction are created. From a high-level, platform independent model, it is possible, in principle, to generate a working program without manual intervention.

Types of model

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



- ✧ Figure: Multiple platform-specific models



- The translation of PIMs to PSMs is more mature and several commercial tools areavailable that provide translators from PIMs to common platforms such as Java andJ2EE.
- These rely on an extensive library of platform-specific rules and patterns toconvert the PIM to the PSM.
- There may be several PSMs for each PIM in the system.If a software system is intended to run on different platforms (e.g., J2EE and .NET),then it is only necessary to maintain the PIM.
- The PSMs for each platform are automaticallygenerated.
- MDA-support tools include platform-specific translators; it is oftenthe case that these will only offer partial support for the translation from PIMs toPSMs.
- 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.
- When MDA is introduced, special purpose translators may have to be created that take the characteristics of the local environment into account. In some cases (e.g., for user interface generation), completely automated PIM to PSM translation may be impossible.

Agile methods and MDA

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

Executable UML

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

Features of executable UML

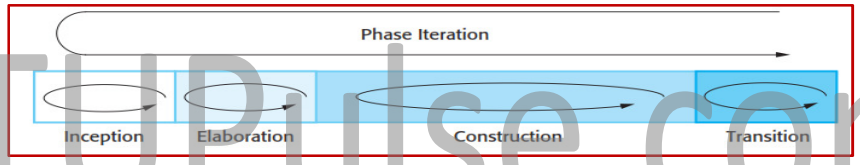
- ✧ To create an executable subset of UML, the number of model types has therefore been dramatically reduced to these 3 key types:
    - Domain models that identify the principal concerns in a system. They are defined using UML class diagrams and include objects, attributes and associations.
    - Class models in which classes are defined, along with their attributes and operations.
    - State models in which a state diagram is associated with each class and is used to describe the life cycle of the class.
- ✧ The dynamic behaviour of the system may be specified declaratively using the object constraint language (OCL), or may be expressed using UML's action language.

Design and Implementation:

The Rational Unified Process

- ✧ A modern generic process derived from the work on the UML and associated process.
- ✧ Brings together aspects of the 3 generic process models discussed previously.
- ✧ Normally described from 3 perspectives
    - A dynamic perspective that shows phases over time;
    - A static perspective that shows process activities;
    - A practice perspective that suggests good practice.

Phases in the Rational Unified Process



- ✧ Inception
    - Establish the business case for the system.
        - You should identify all external entities (people and systems) that will interact with the system and define these interactions. You then use this information to assess the contribution that the system makes to the business. If this contribution is minor, then the project may be cancelled after this phase.
- ✧ Elaboration
    - Develop an understanding of the problem domain and the system architecture.
        - The goals of the elaboration phase are to develop an understanding of the problem domain, establish an architectural framework for the system, develop the project plan, and identify key project risks. On completion of this phase you should have a requirements model for the system, which may be a set of UML use-cases, an architectural description, and a development plan for the software.
- ✧ Construction
    - System design, programming and testing.
        - 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.
- ✧ Transition
    - Deploy the system in its operating environment.
        - The final phase of the RUP is concerned with moving the system from the development community to the user community and making it work in a real environment. This is something that is ignored in most software process models but is, in fact, an expensive and sometimes problematic activity. On completion of this phase, you should have a documented software system that is working correctly in its operational environment

RUP iteration

- In-phase iteration
  - Each phase is iterative with results developed incrementally.
- Cross-phase iteration
  - As shown by the loop in the RUP model, the whole set of phases may be enacted incrementally.

Static workflows in the Rational Unified Process

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.

RUP good practice

- Develop software iteratively
  - Plan increments based on customer priorities and deliver highest priority increments first.
- Manage requirements
  - Explicitly document customer requirements and keep track of changes to these requirements.
- Use component-based architectures
  - Organize the system architecture as a set of reusable components.
- Visually model software
  - Use graphical UML models to present static and dynamic views of the software.
- Verify software quality
  - Ensure that the software meet's organizational quality standards.
- Control changes to software
  - Manage software changes using a change management system and configuration management tools.

Design principles

Component-based development

- Component-based software engineering (CBSE) is an approach to software development that relies on the reuse of entities called 'software components'.
- It emerged from the failure of object-oriented development to support effective reuse. Single object classes are too detailed and specific.
- Components are more abstract than object classes and can be considered to be stand-alone service providers. They can exist as stand-alone entities.

CBSE essentials

- ✧ Independent components specified by their interfaces.
- ✧ Component standards to facilitate component integration.
- ✧ Middleware that provides support for component inter-operability.
- ✧ A development process that is geared to reuse.

Design Principle

- ✧ Components are independent so do not interfere with each other;
- ✧ Component implementations are hidden;
- ✧ Communication is through well-defined interfaces;
- ✧ Component platforms are shared and reduce development costs

Component standards

- ✧ Standards need to be established so that components can communicate with each other and inter-operate.
- ✧ Unfortunately, several competing component standards were established:
  - Sun's Enterprise Java Beans
  - Microsoft's COM and .NET
  - CORBA's CCM
- ✧ In practice, these multiple standards have hindered the uptake of CBSE. It is impossible for components developed using different approaches to work together

Component definitions

- ✧ Councill and Heinmann:
  - A software component is a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard.
- ✧ Szyperski:
  - A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third-parties.

Component Characteristics

Component Characteristic	Description
Standardized	Component standardization means that a component used in a CBSE process has to conform to a standard component model. This model may define component interfaces, component metadata, documentation, composition, and deployment.
Independent	A component should be independent—it should be possible to compose and deploy it without having to use other specific components. In situations where the component needs externally provided services, these should be explicitly set out in a 'requires' interface specification.
Composable	For a component to be composable, all external interactions must take place through publicly defined interfaces. In addition, it must provide external access to information about itself, such as its methods and attributes.
Deployable	To be deployable, a component has to be self-contained. It must be able to operate as a stand-alone entity on a component platform that provides an implementation of the component model. This usually means that the component is binary and does not have to be compiled before it is deployed. If a component is implemented as a service, it does not have to be deployed by a user of a component. Rather, it is deployed by the service provider.
Documented	Components have to be fully documented so that potential users can decide whether or not the components meet their needs. The syntax and, ideally, the semantics of all component interfaces should be specified.

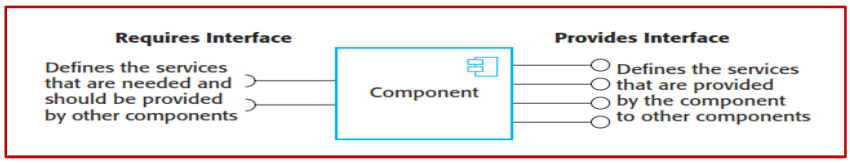
Component interfaces

- ✧ Provides interface
  - Defines the services that are provided by the component to other components.

- This interface, essentially, is the component API. It defines the methods that can be called by a user of the component.

✧ Requires interface

- Defines the services that specifies what services must be made available for the component to execute as specified.
- This does not compromise the independence or deployability of a component because the 'requires' interface does not define how these services should be provided.



Component models

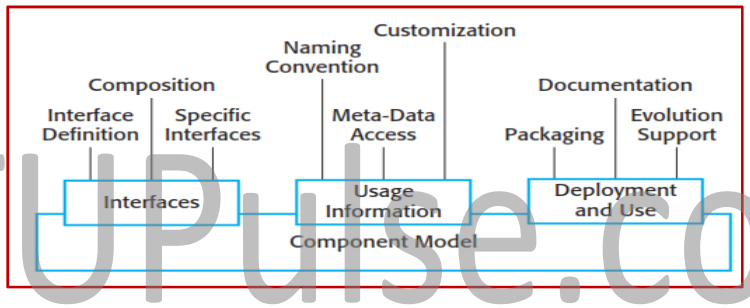
✧ A component model is a definition of standards for component implementation, documentation and deployment.

✧ Examples of component models

- EJB model (Enterprise Java Beans)
- COM+ model (.NET model)
- Corba Component Model

✧ The component model specifies how interfaces should be defined and the elements that should be included in an interface definition.

Basic elements of a component model



✧ Interfaces

- Components are defined by specifying their interfaces. The component model specifies how the interfaces should be defined and the elements, such as operation names, parameters and exceptions, which should be included in the interface definition.

✧ Usage

- In order for components to be distributed and accessed remotely, they need to have a unique name or handle associated with them. This has to be globally unique.

✧ Deployment

- The component model includes a specification of how components should be packaged for deployment as independent, executable entities.

CBSE processes

✧ CBSE processes are software processes that support component-based software engineering.

- They take into account the possibilities of reuse and the different process activities involved in developing and using reusable components.

✧ Development for reuse

- This process is concerned with developing components or services that will be reused in other applications. It usually involves generalizing existing components.

✧ Development with reuse

- This process is the process of developing new applications using existing components and services.

Supporting processes

✧ Component acquisition is the process of acquiring components for reuse or development into a reusable component.

- It may involve accessing locally- developed components or services or finding these components from an external source.

- ✧ Component management is concerned with managing a company's reusable components, ensuring that they are properly catalogued, stored and made available for reuse.
- ✧ Component certification is the process of checking a component and certifying that it meets its specification.

An object-oriented design process

- ✧ Structured object-oriented design processes involve developing a number of different system models.
- ✧ They require a lot of effort for development and maintenance of these models and, for small systems, this may not be cost-effective.
- ✧ However, for large systems developed by different groups design models are an important communication mechanism.

Process stages

- ✧ There are a variety of different object-oriented design processes that depend on the organization using the process.
- ✧ Common activities in these processes include:
  - Define the context and modes of use of the system;
  - Design the system architecture;
  - Identify the principal system objects;
  - Develop design models;
  - Specify object interfaces.
- ✧ Process illustrated here using a design for a wilderness weather station.

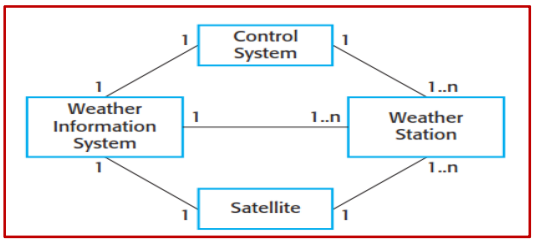
System context and interactions

- ✧ Understanding the relationships between the software that is being designed and its external environment 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.

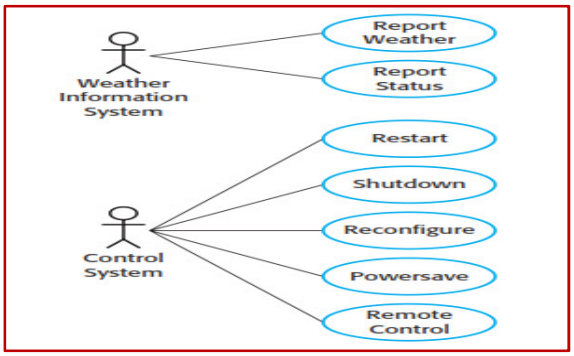
Context and interaction models

- ✧ 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.

System context for the weather station



Weatherstation use cases



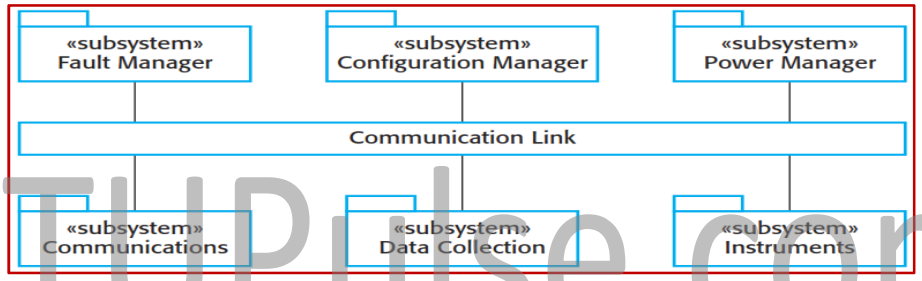
- ✧ Report weather—send weather data to the weather information system
- ✧ Report status—send status information to the weather information system
- ✧ Restart—if the weather station is shut down, restart the system
- ✧ Shutdown—shut down the weather station
- ✧ Reconfigure—reconfigure the weather station software
- ✧ Power save—put the weather station into power-saving mode
- ✧ Remote control—send control commands to any weather station subsystem

Use casedescription—Reportweather

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.

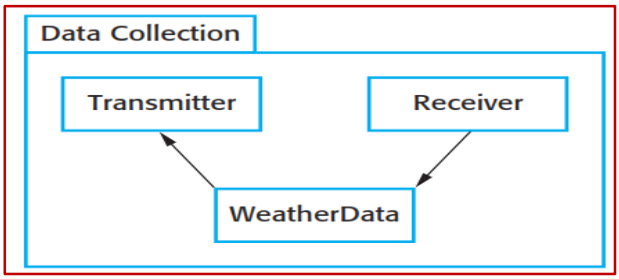
Architectural design

- Once interactions between the system and its environment have been understood, you use this information for designing the system architecture.
- 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.
- The weather station is composed of independent subsystems that communicate by broadcasting messages on a common infrastructure.



- The weather station is composed of independent subsystems that communicate by broadcasting messages on a common infrastructure.
- Each subsystem listens for messages on that infrastructure and picks up the messages that are intended for them.
- For example, when the communications subsystem receives a control command, such as shutdown, the command is picked up by each of the other subsystems, which then shut themselves down in the correct way. The key benefit of this architecture is that it is easy to support different configurations of subsystems because the sender of a message does not need to address the message to a particular subsystem.

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.

Object class identification

- Identifying object classes is often a difficult part of object oriented design.
- There is no 'magic formula' for object identification. It relies on the skill, experience and domain knowledge of system designers.
- Object identification is an iterative process. You are unlikely to get it right first time.

Approaches to identification

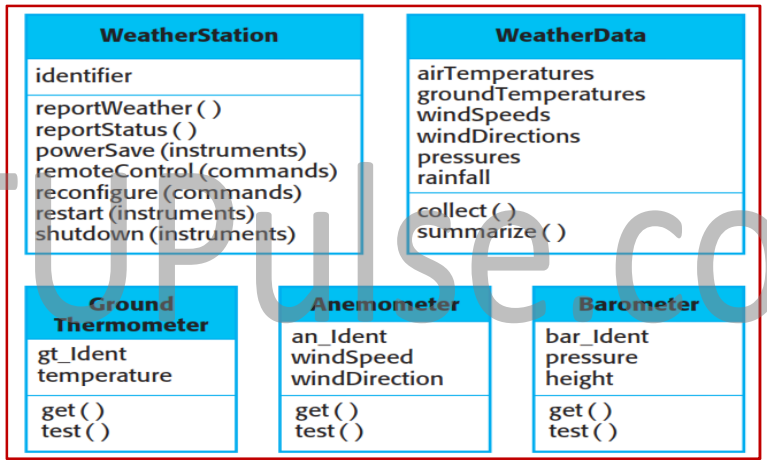
- ✧ Use a grammatical approach based on a natural language description of the system (used in Hood OOD method).
- ✧ Base the identification on tangible things in the application domain.
- ✧ Use a behavioural approach and identify objects based on what participates in what behaviour.
- ✧ Use a scenario-based analysis. The objects, attributes and methods in each scenario are identified.

Weather station description

- ✧ A weather station is a package of software controlled instruments which collects data, performs some data processing and transmits this data for further processing. The instruments include air and ground thermometers, an anemometer, a wind vane, a barometer and a rain gauge. Data is collected periodically.
- ✧ When a command is issued to transmit the weather data, the weather station processes and summarises the collected data. The summarised data is transmitted to the mapping computer when a request is received.

Weather station object classes

- ✧ Object class identification in the weather station system may be based on the tangible hardware and data in the system:
  - ▪ Ground thermometer, Anemometer, Barometer
    - • Application domain objects that are 'hardware' objects related to the instruments in the system.
  - ▪ Weather station
    - • The basic interface of the weather station to its environment. It therefore reflects the interactions identified in the use-case model.
  - ▪ Weather data
    - • Encapsulates the summarized data from the instruments.



Design models

- ✧ Design models show the objects and object classes and relationships between these entities.
- ✧ Static models describe the static structure of the system in terms of object classes and relationships.
- ✧ Dynamic models describe the dynamic interactions between objects.

Examples of design models

- ✧ Subsystem models that show logical groupings of objects into coherent subsystems.
- ✧ Sequence models that show the sequence of object interactions.
- ✧ State machine models that show how individual objects change their state in response to events.
- ✧ Other models include use-case models, aggregation models, generalisation models, etc.

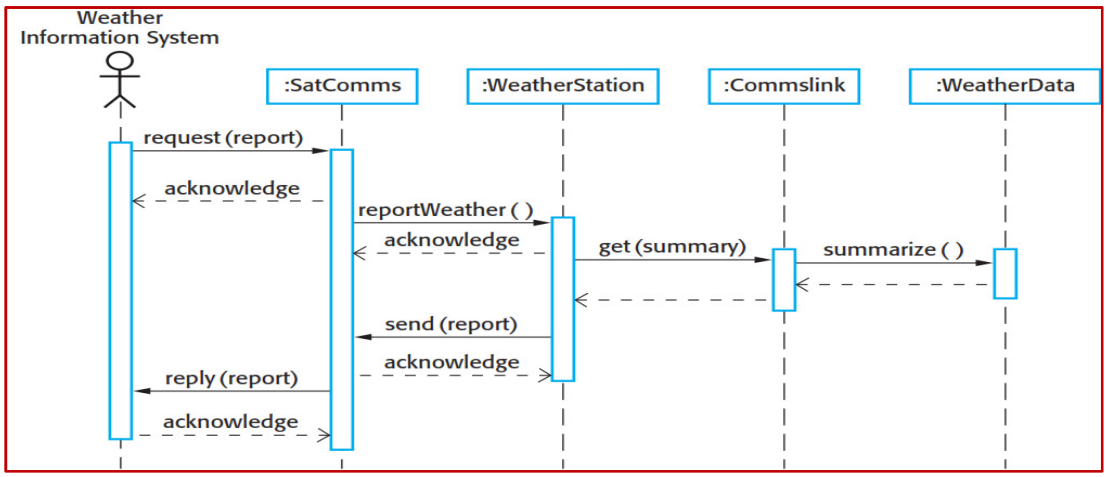
Subsystem models

- ✧ Shows how the design is organised into logically related groups of objects.
- ✧ In the UML, these are shown using packages - an encapsulation construct. This is a logical model. The actual organisation of objects in the system may be different.

Sequence models

- ✧ Sequence models show the sequence of object interactions that take place
  - ▪ Objects are arranged horizontally across the top;
  - ▪ Time is represented vertically so models are read top to bottom;
  - ▪ Interactions are represented by labelled arrows, Different styles of arrow represent different types of interaction;

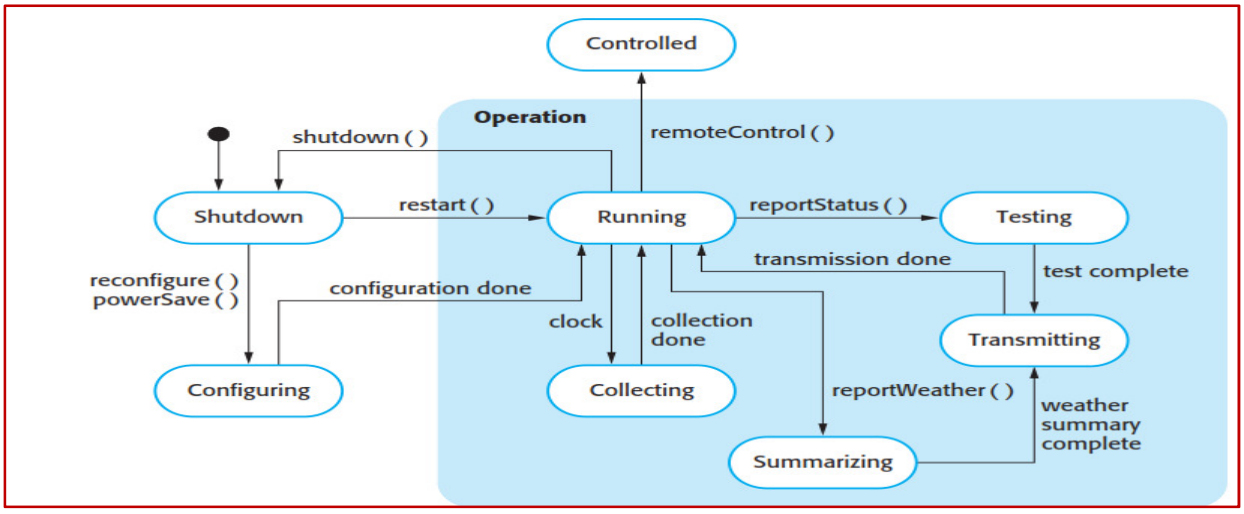
- A thin rectangle in an object lifeline represents the time when the object is the controlling object in the system.



- ✧ 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.
- ✧ 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.
- ✧ 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.
- ✧ Commslink calls the summarize method in the object WeatherData and waits for a reply.
- ✧ The weather data summary is computed and returned to WeatherStation via the Commslink object.
- ✧ WeatherStation then calls the SatComms object to transmit the summarized data to the weather information system, through the satellite communications system.

State diagrams

- ✧ State diagrams are used to show how objects respond to different service requests and the state transitions triggered by these requests.
- ✧ State diagrams are useful high-level models of a system or an object's run-time behavior.
- ✧ You don't usually need a state diagram for all of the objects in the system. Many of the objects in a system are relatively simple and a state model adds unnecessary detail to the design.



- ✧ 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 normal 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.
- ✧ In the Running state, the system expects further messages. If a shutdown() message is received, the object returns to the shutdown state.

- 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.
- If a reportStatus() message is received, the system moves to the Testing state, then the Transmitting state, before returning to the Running state.
- 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.
- If a remoteControl() message is received, the system moves to a controlled state in which it responds to a different set of messages from the remote control room. These are not shown on this diagram.

Interface specification

- Object interfaces have to be specified so that the objects and other components can be designed in parallel.
- Designers should avoid designing the interface representation but should hide this in the object itself.
- Objects may have several interfaces which are viewpoints on the methods provided.
- The UML uses class diagrams for interface specification but Java may also be used.

Design patterns

- A design pattern is a way of reusing abstract knowledge about a problem and its solution.
- A pattern is a description of the problem and the essence of its solution.
- It should be sufficiently abstract to be reused in different settings.
- Pattern descriptions usually make use of object-oriented characteristics such as inheritance and polymorphism.

Pattern elements

- Name
  - A meaningful pattern identifier.
- Problem description.
- Solution description.
  - Not a concrete design but a template for a design solution that can be instantiated in different ways.
- Consequences
  - The results and trade-offs of applying the pattern.

The Observer pattern

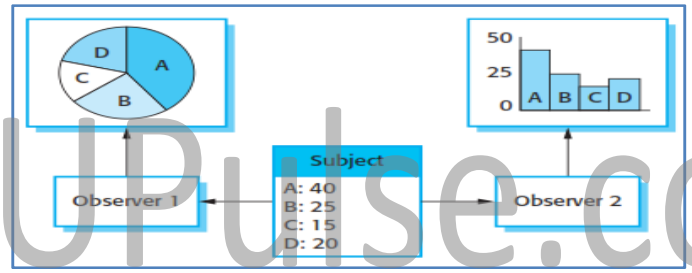
- Name
  - Observer.
- Description
  - Separates the display of object state from the object itself.
- Problem description
  - Used when multiple displays of state is needed.
- Solution description
  - See slide with UML description.
- Consequences
  - Optimisations to enhance display performance are impractical

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.

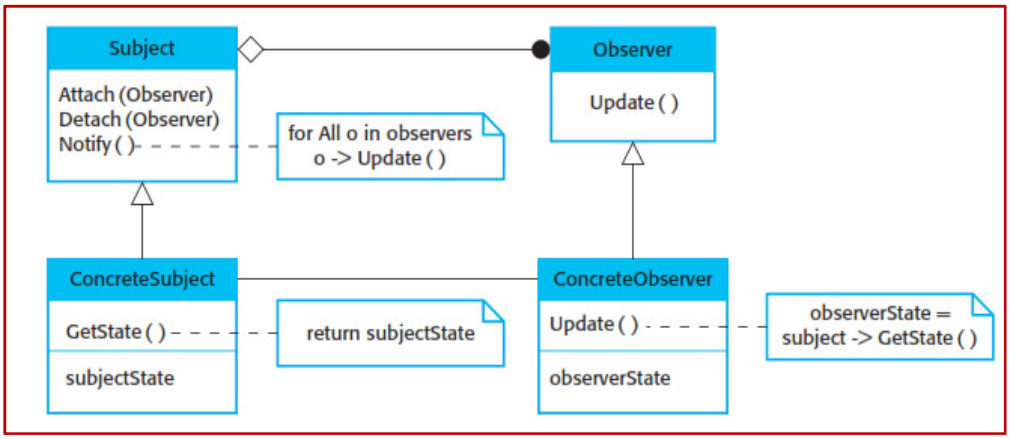
Multiple displays using the Observer pattern



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

- 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.

A UML model of the Observer pattern



Design problems

- To use patterns in your design, you need to recognize that any design problem you are facing may have an associated pattern that can be applied.
  - Tell several objects that the state of some other object has changed (Observer pattern).
  - Tidy up the interfaces to a number of related objects that have often been developed incrementally (Façade pattern).
  - Provide a standard way of accessing the elements in a collection, irrespective of how that collection is implemented (Iterator pattern).
  - Allow for the possibility of extending the functionality of an existing class at run-time (Decorator pattern).

Implementation issues

- Focus here is not on programming, although this is obviously important, but on other implementation issues that are often not covered in programming texts:
  - 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.
  - Configuration management during the development process, you have to keep track of the many different versions of each software component in a configuration management system.
  - 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).

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.
- Costs and schedule pressure mean that this approach became increasingly unviable, especially for commercial and Internet-based systems.
- An approach to development based around the reuse of existing software emerged and is now generally used for business and scientific software.

Reuse levels

- The abstraction level
  - At this level, you don't reuse software directly but use knowledge of successful abstractions in the design of your software.
- The object level
  - At this level, you directly reuse objects from a library rather than writing the code yourself.
- The component level
  - Components are collections of objects and object classes that you reuse in application systems.
- The system level
  - At this level, you reuse entire application systems.

Reuse costs

- The costs of the time spent in looking for software to reuse and assessing whether or not it meets your needs.
- 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.

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.
- 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.

- ✧ 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.

Host-target development

- ✧ Most 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.
- ✧ Development platform usually has different installed software than execution platform; these platforms may have different architectures.

Development platform tools

- ✧ An integrated compiler and syntax-directed editing system that allows you 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.

Integrated development environments (IDEs)

- ✧ 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.
- ✧ 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.

Component/system deployment factors

- ✧ If a component is designed for 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.
- ✧ 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.
- ✧ 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 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 (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.
- ✧ 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.
- ✧ The best-known open source product is, of course, the Linux operating system which is widely used as a server system and, increasingly, as a desktop environment.
- ✧ Other important open source products are Java, the Apache web server and the mySQL database management system.

Open source issues

- ✧ 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?

Open source business

- ✧ More and more product companies are using an open source approach to development.
- ✧ Their business model is not reliant on selling a software product but on selling support for that product.

- ✧ They 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.

Open source licensing

- ✧ 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.
    - Some open source developers believe that if an open source component is used to develop a new system, then that system should also be open source.
    - Others are willing to allow their code to be used without this restriction. The developed systems may be proprietary and sold as closed source systems.

License models

- ✧ The GNU General Public License (GPL). This is a so-called 'reciprocal' license that means that if you use open source software that is licensed under the GPL license, then you must make that software open source.
- ✧ The GNU Lesser General Public License (LGPL) 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.
- ✧ The Berkley Standard Distribution (BSD) License. This is a non-reciprocal license, which means you are not obliged to re-publish any changes or modifications made to open source code. You can include the code in proprietary systems that are sold.

License management

- ✧ 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.
- ✧ Have auditing systems in place.
- ✧ Participate in the open source community.