

---

# **Chapter 5 – System Modeling**

# Topics covered

---

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

# System modeling

---

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

# Existing and planned system models

---

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

# System perspectives

---

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

# UML diagram types

---

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

# Use of graphical models

---

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

---

# **Context models**

# Context models

---

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

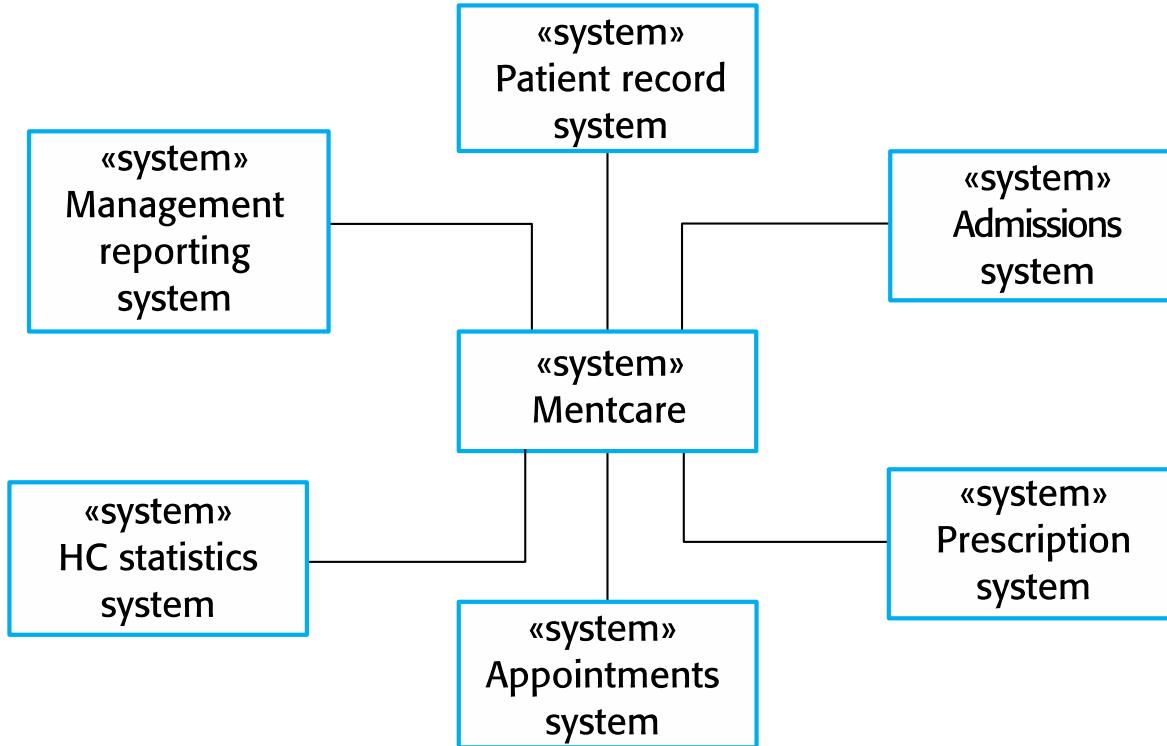
# System boundaries

---

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

# The context of the Mentcare system

---

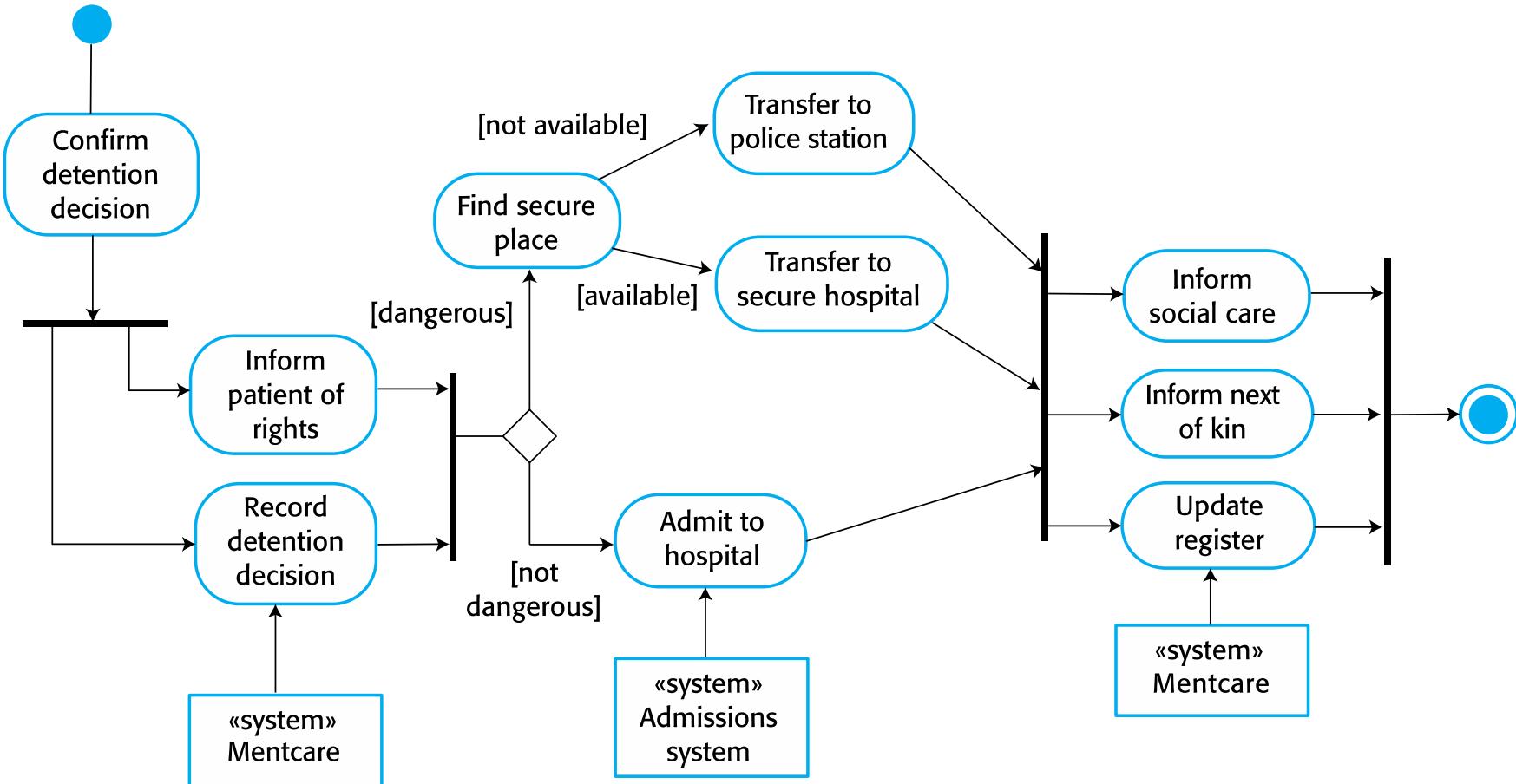


# Process perspective

---

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

# Process model of involuntary detention



# Interaction models

# Interaction models

---

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

# 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.
- ✧ Represented diagrammatically to provide an overview of the use case and in a more detailed textual form.

# Transfer-data use case

---

✧ A use case in the Mentcare system



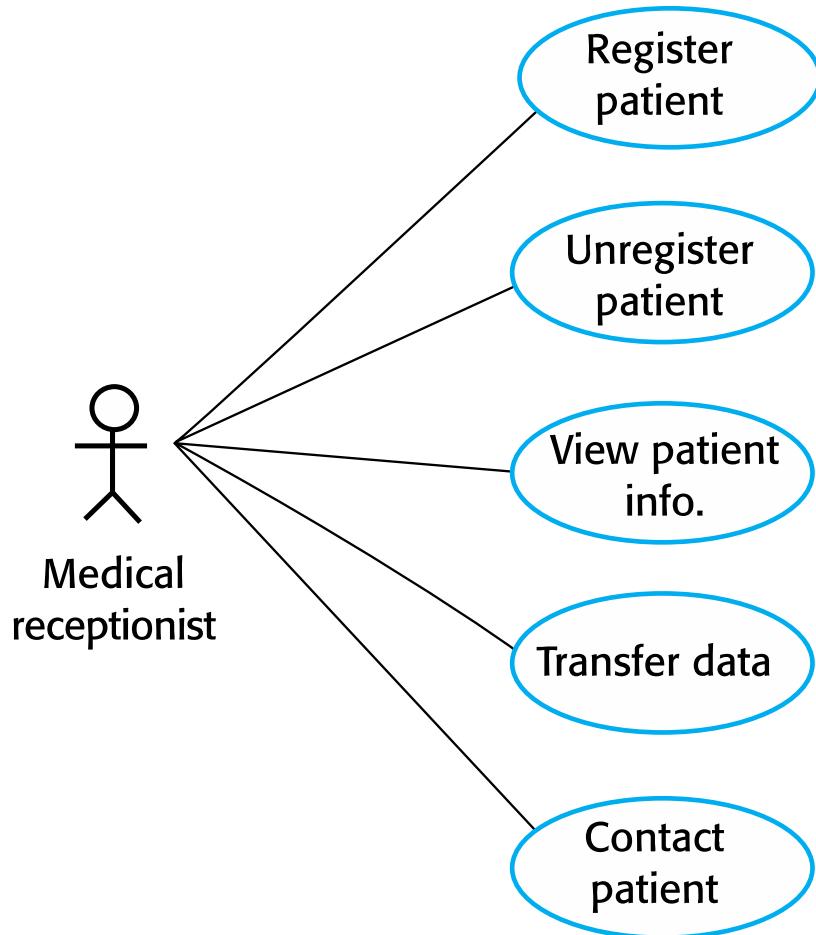
# Tabular description of the ‘Transfer data’ use-case

---

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

# Use cases in the Mentcare system involving the role 'Medical Receptionist'

---

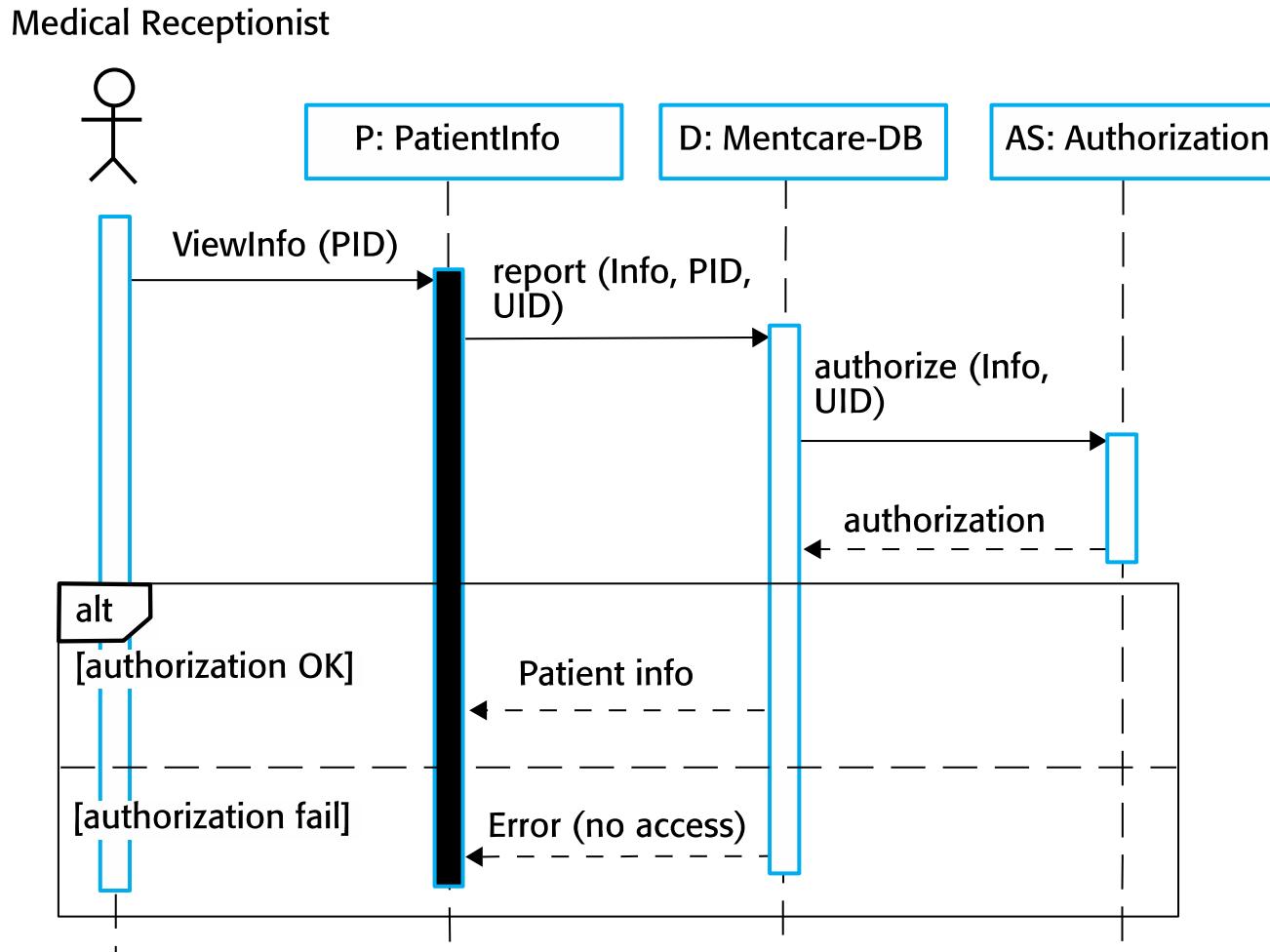


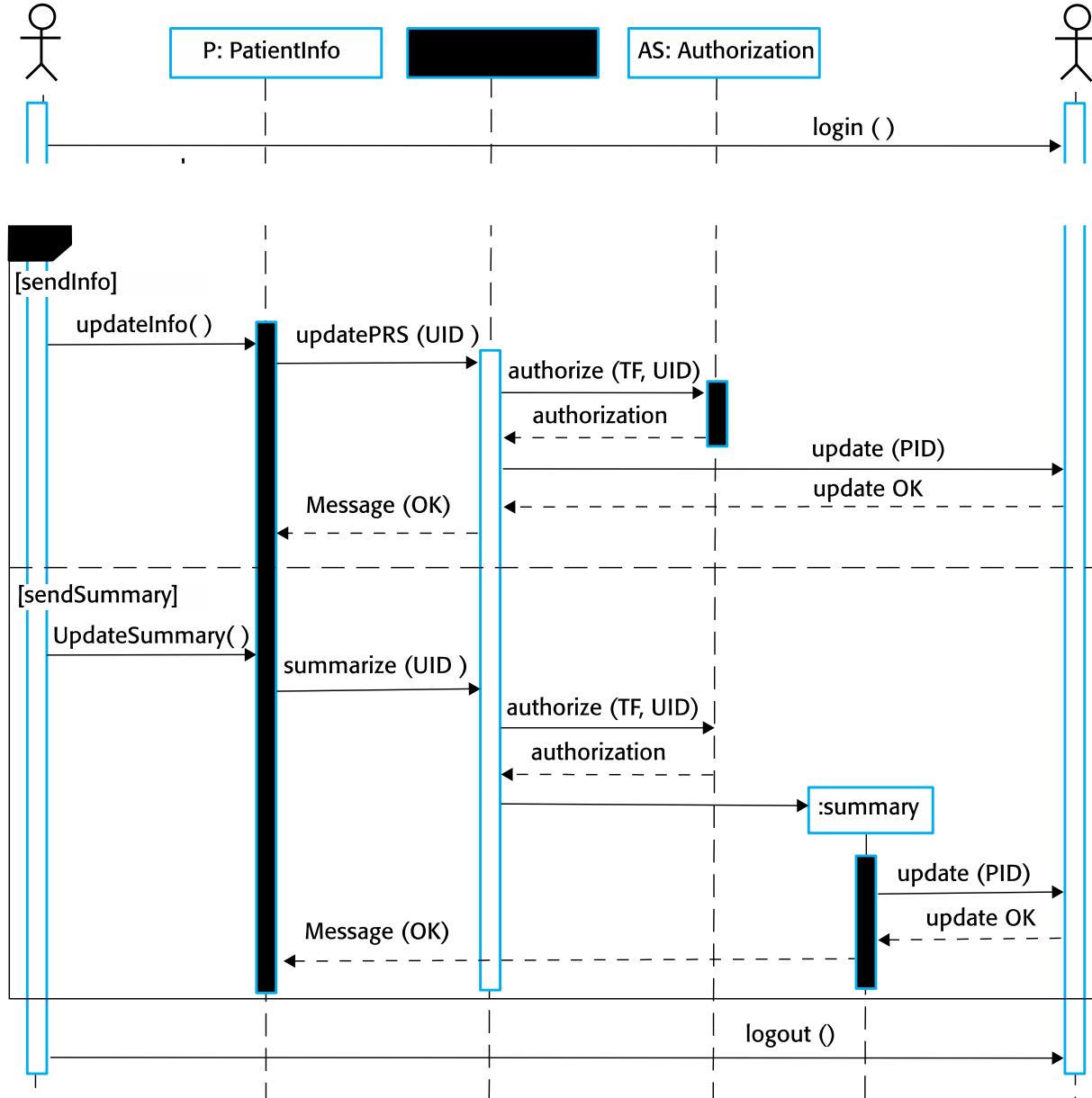
# Sequence diagrams

---

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

# Sequence diagram for View patient information





## Sequence diagram for Transfer Data

---

# **Structural models**

# Structural models

---

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

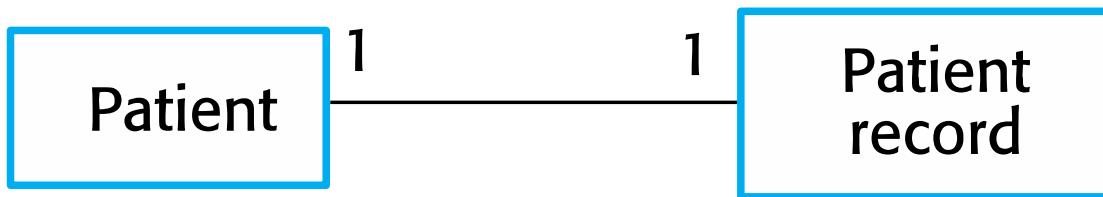
# Class diagrams

---

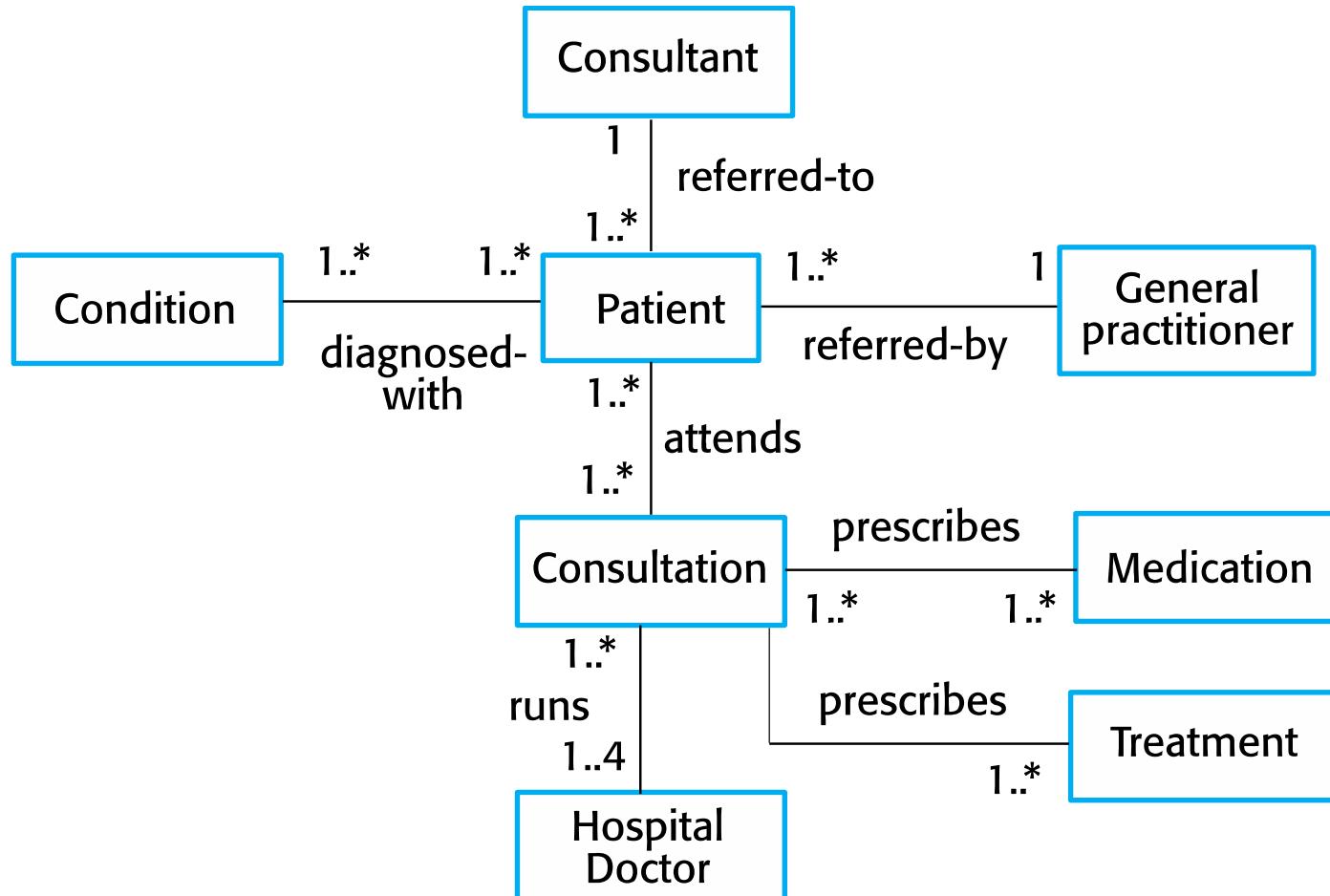
- ✧ Class diagrams are used when developing an object-oriented system model to show the classes in a system and the associations between these classes.
- ✧ An object class can be thought of as a general definition of one kind of system object.
- ✧ An association is a link between classes that indicates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

# UML classes and association

---

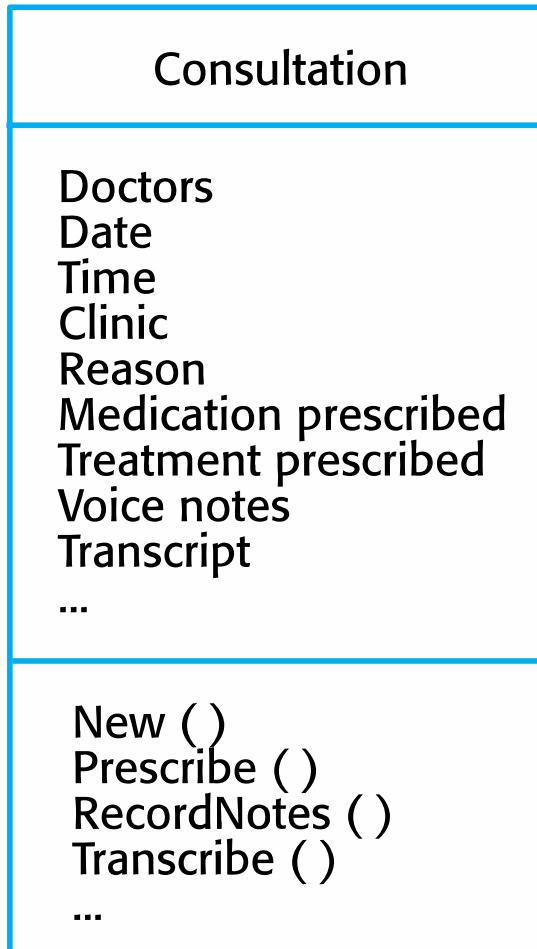


# Classes and associations in the MHC-PMS



# The Consultation class

---



# Generalization

---

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

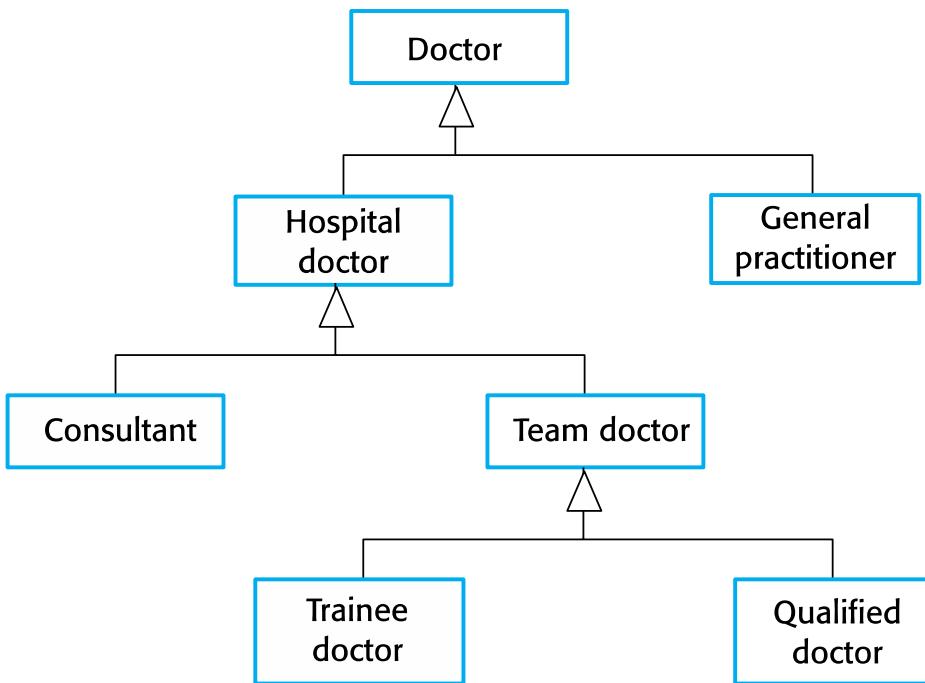
# Generalization

---

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

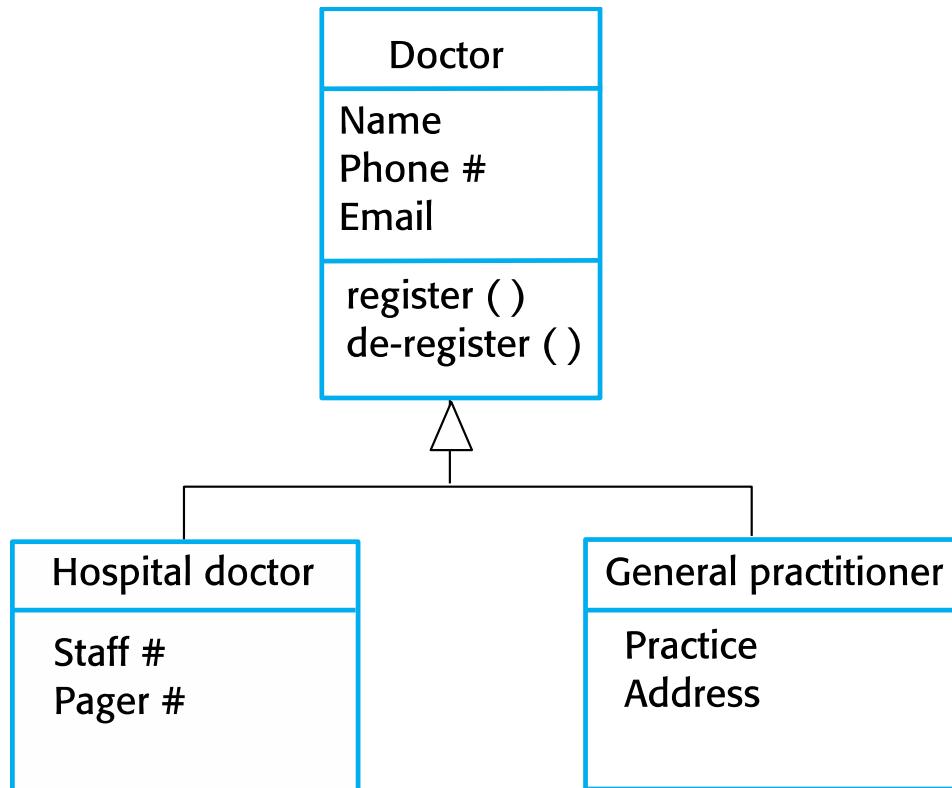
# A generalization hierarchy

---



# A generalization hierarchy with added detail

---



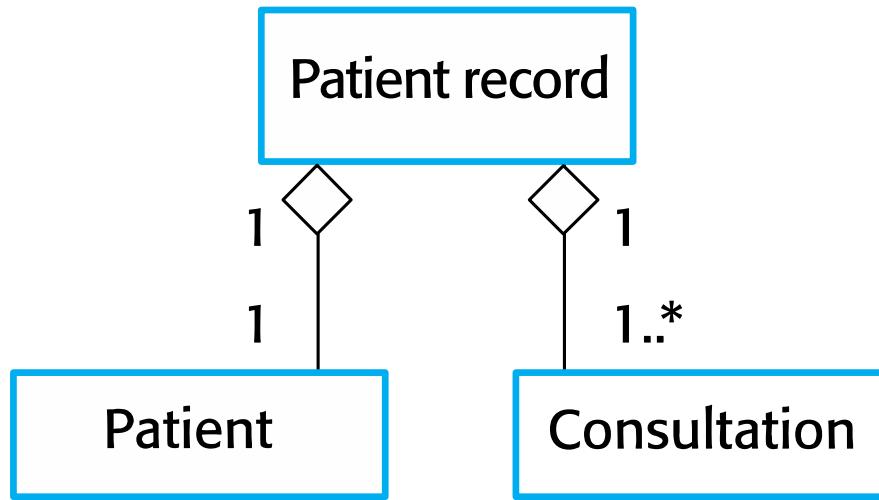
# Object class aggregation models

---

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

# The aggregation association

---



---

# **Behavioral models**

# Behavioral models

---

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

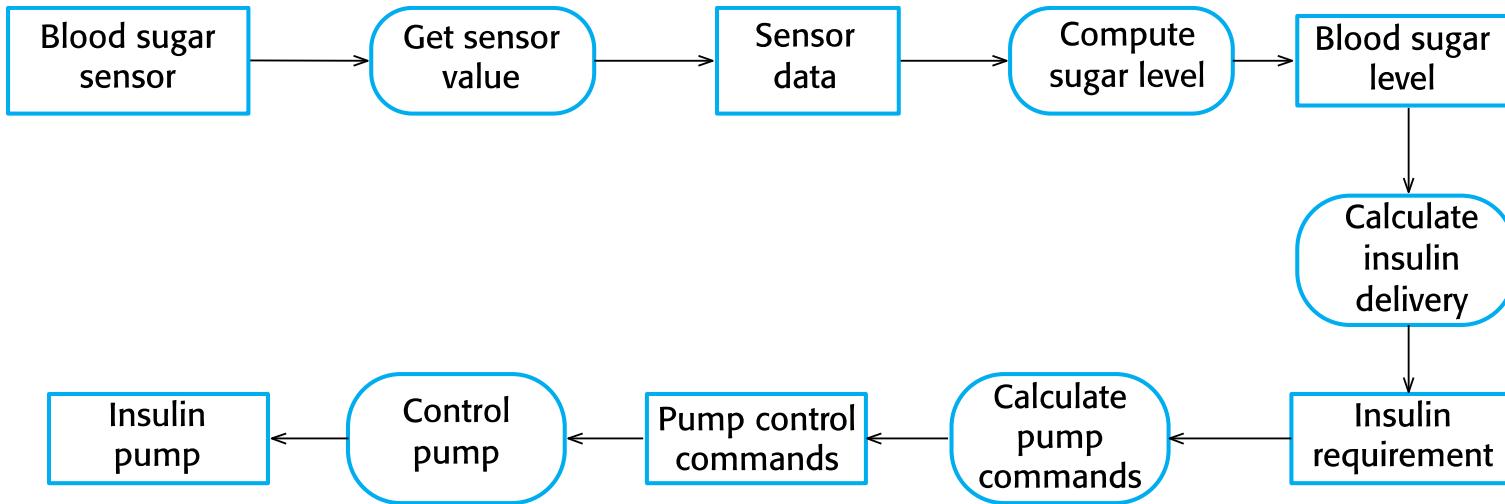
# Data-driven modeling

---

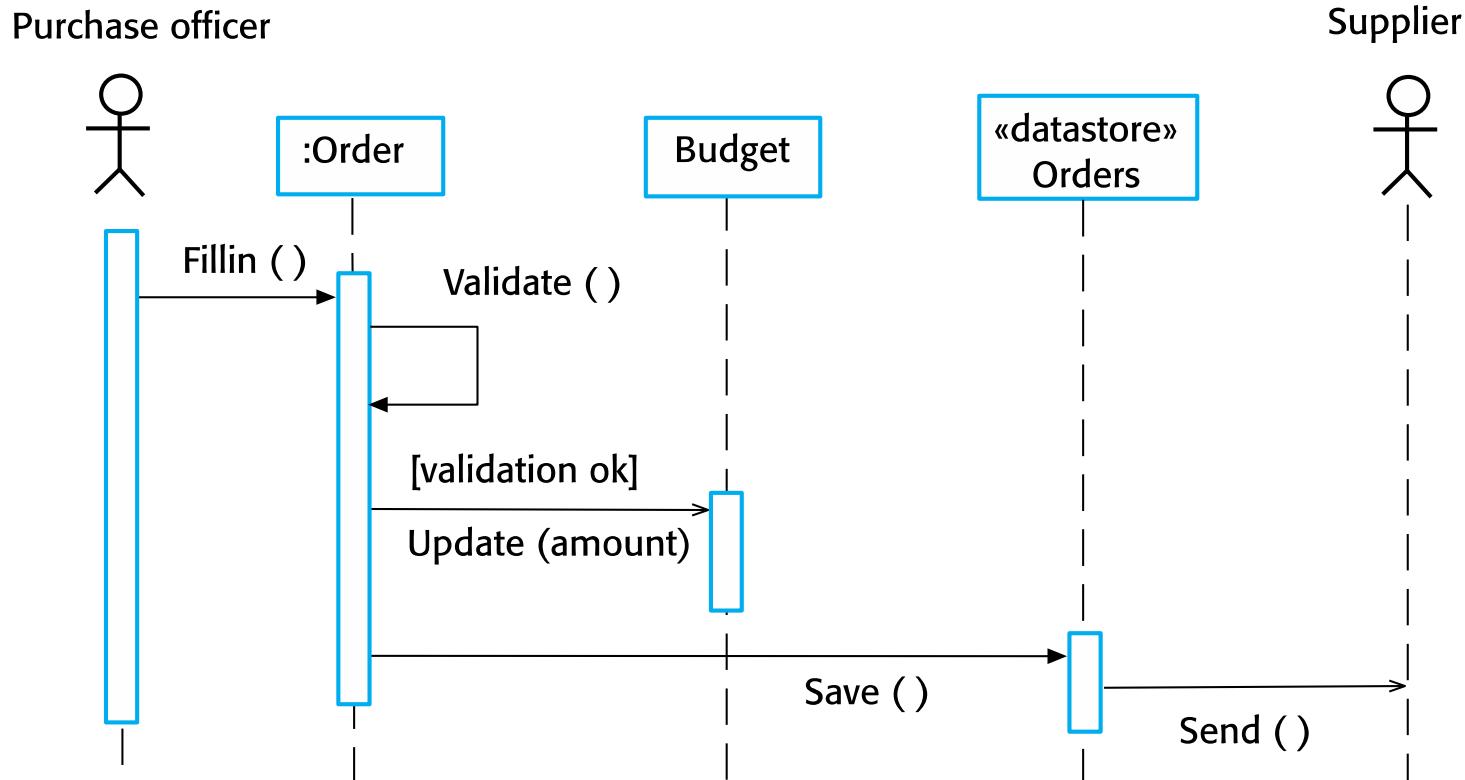
- ✧ Many business systems are data-processing systems that are primarily driven by data. They are controlled by the data input to the system, with relatively little external event processing.
- ✧ Data-driven models show the sequence of actions involved in processing input data and generating an associated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

# An activity model of the insulin pump's operation

---



# Order processing



# Event-driven modeling

---

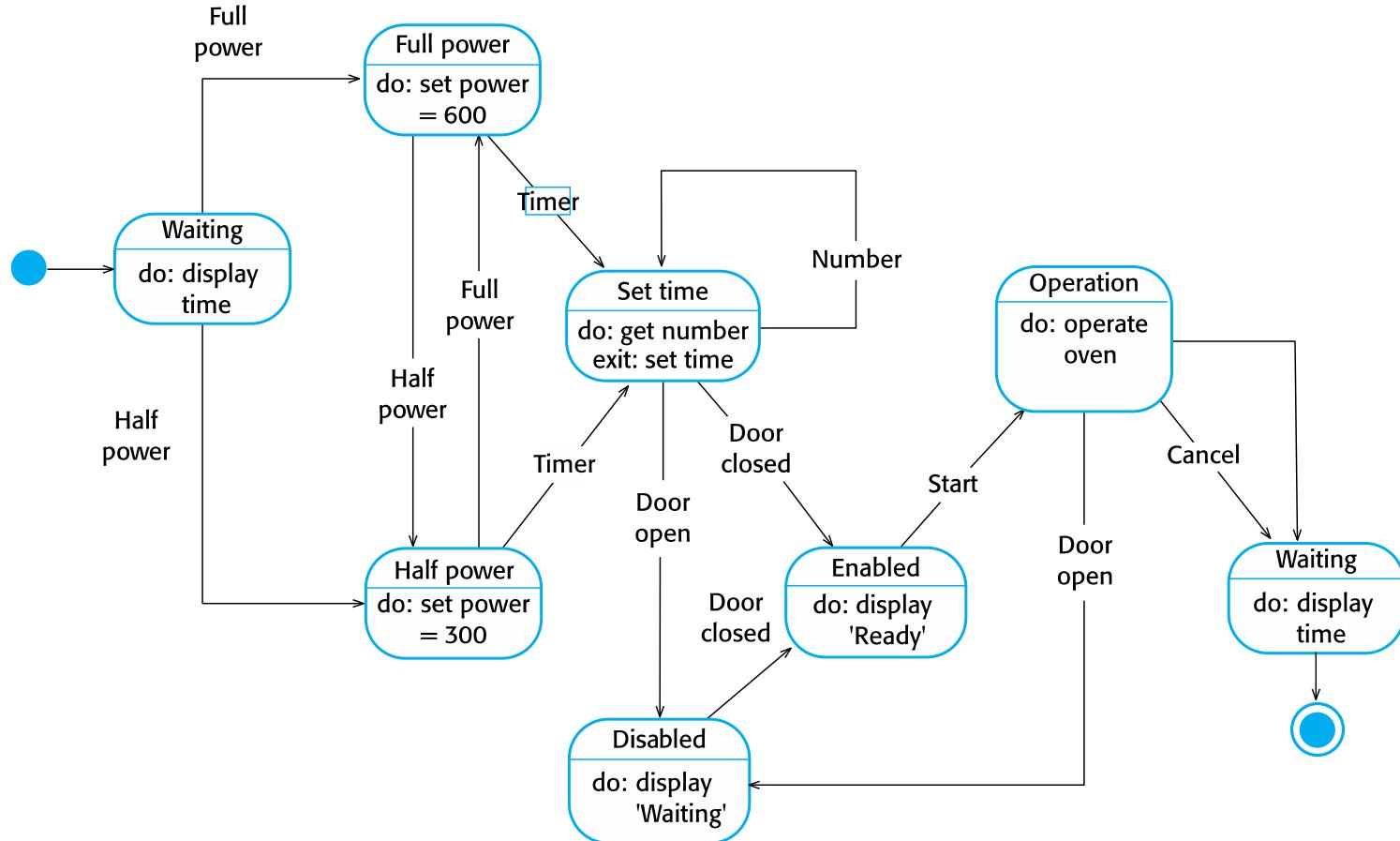
- ✧ Real-time systems are often event-driven, with minimal data processing. For example, a landline phone switching system responds to events such as ‘receiver off hook’ by generating a dial tone.
- ✧ Event-driven modeling shows how a system responds to external and internal events.
- ✧ It is based on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

# State machine models

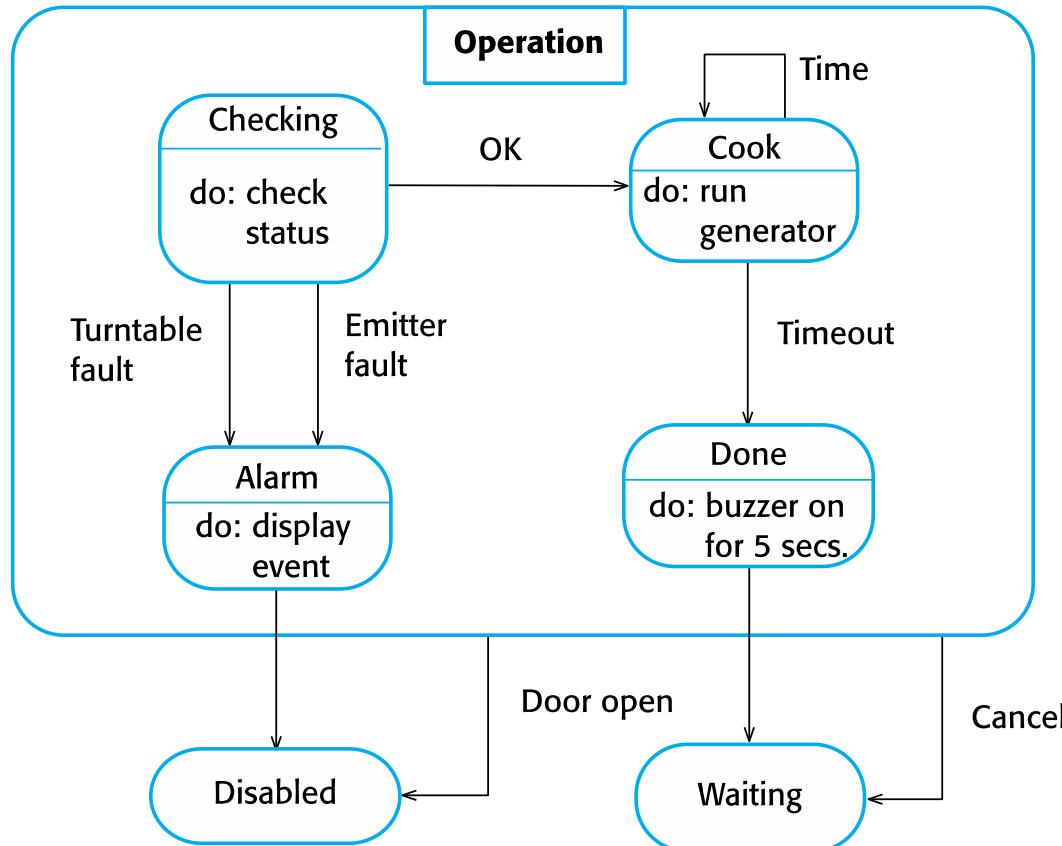
---

- ✧ These model the 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.
- ✧ Statecharts are an integral part of the UML and are used to represent state machine models.

# State diagram of a microwave oven



# Microwave oven operation



# 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.

# **Model-driven engineering**

# Model-driven engineering

---

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

# Usage of model-driven engineering

---

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

# Model driven architecture

---

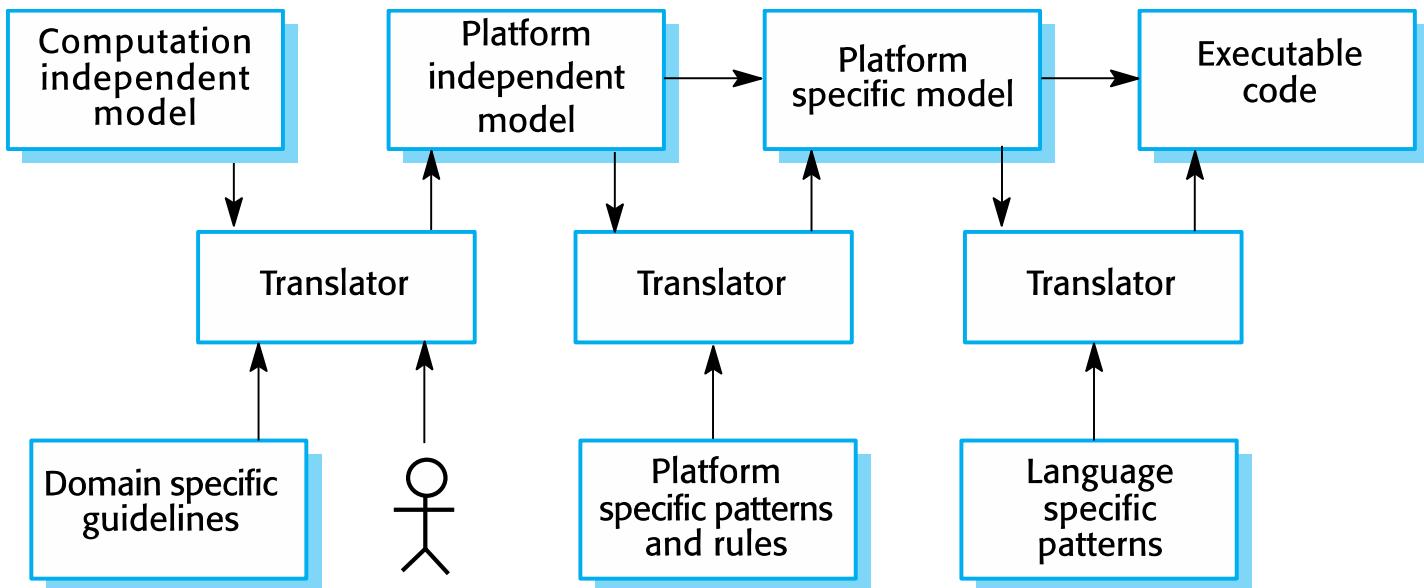
- ✧ Model-driven architecture (MDA) was the precursor of more general model-driven engineering
- ✧ MDA is a model-focused approach to software design and implementation that uses a subset of UML models to describe a system.
- ✧ Models at different levels of abstraction are created. From a high-level,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

# Types of model

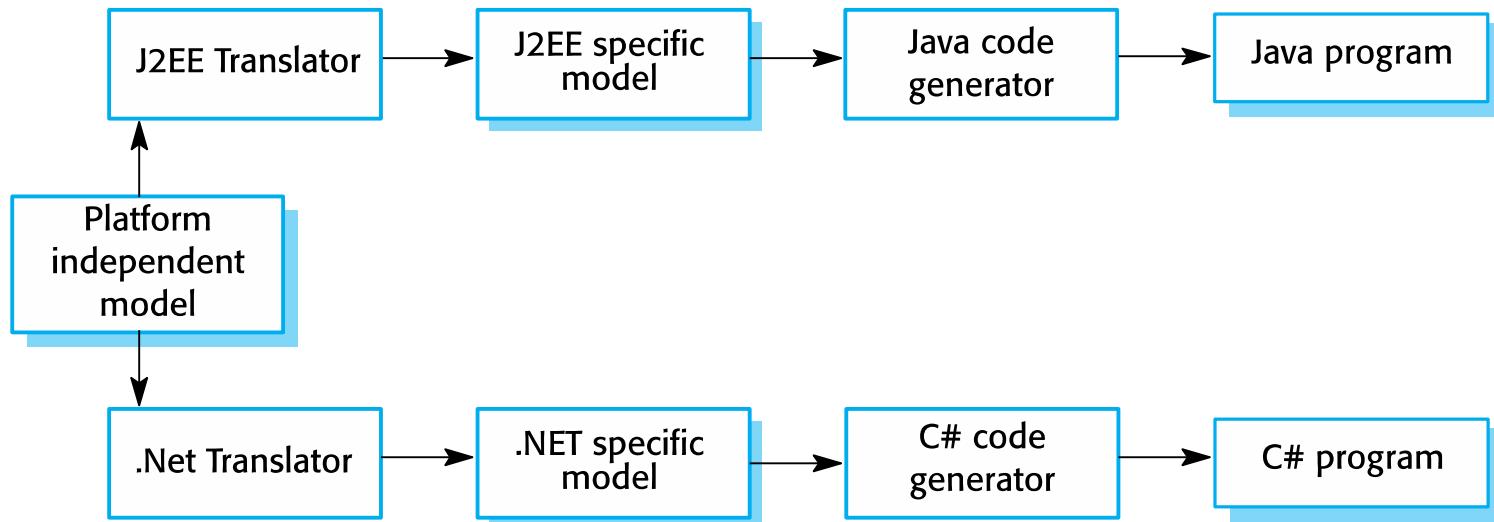
---

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

# MDA transformations



# Multiple platform-specific models



# Agile methods and MDA

---

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

# Adoption of MDA

---

- ✧ A range of factors has limited the adoption of MDE/MDA
- ✧ Specialized tool support is required to convert models from one level to another
- ✧ There is limited tool availability and organizations may require tool adaptation and customisation to their environment
- ✧ For the long-lifetime systems developed using MDA, companies are reluctant to develop their own tools or rely on small companies that may go out of business

# Adoption of MDA

---

- ✧ Models are a good way of facilitating discussions about a software design. However the abstractions that are useful for discussions may not be the right abstractions for implementation.
- ✧ For most complex systems, implementation is not the major problem – requirements engineering, security and dependability, integration with legacy systems and testing are all more significant.

# Adoption of MDA

---

- ✧ The arguments for platform-independence are only valid for large, long-lifetime systems. For software products and information systems, the savings from the use of MDA are likely to be outweighed by the costs of its introduction and tooling.
- ✧ The widespread adoption of agile methods over the same period that MDA was evolving has diverted attention away from model-driven approaches.

# Key points

---

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

# Key points

---

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

# **Chapter 6 – Architectural Design**

# Topics covered

---

- ✧ Architectural design decisions
- ✧ Architectural views
- ✧ Architectural patterns
- ✧ Application architectures

# Architectural design

---

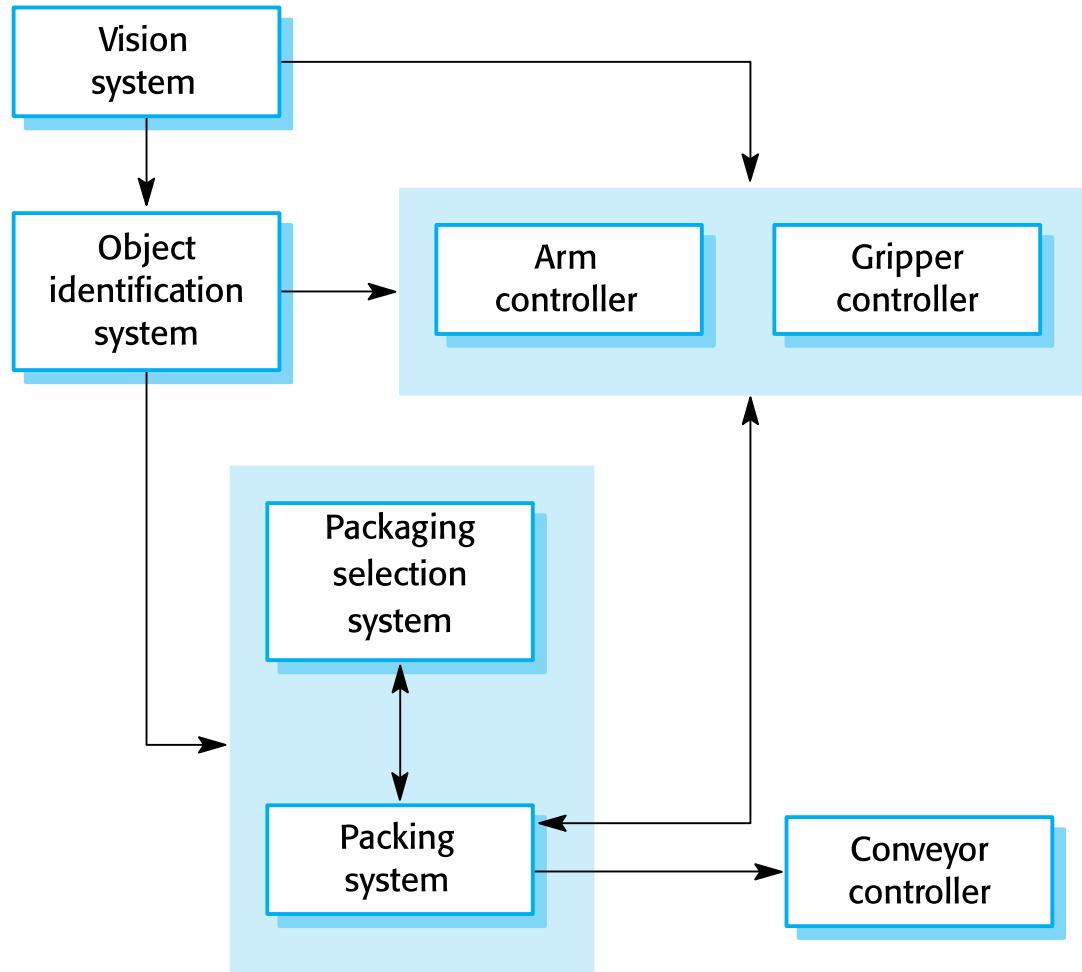
- ✧ Architectural design is concerned with understanding how a software system should be organized and designing the overall structure of that system.
- ✧ Architectural design is the critical link between design and requirements engineering, as it identifies the main structural components in a system and the relationships between them.
- ✧ The output of the architectural design process is an architectural model that describes how the system is organized as a set of communicating components.

# Agility and architecture

---

- ✧ It is generally accepted that an early stage of agile processes is to design an overall systems architecture.
- ✧ Refactoring the system architecture is usually expensive because it affects so many components in the system

# The architecture of a packing robot control system



# Architectural abstraction

---

- ✧ Architecture in the small is concerned with the architecture of individual programs. At this level, we are concerned with the way that an individual program is decomposed into components.
- ✧ Architecture in the large is concerned with the architecture of complex enterprise systems that include other systems, programs, and program components. These enterprise systems are distributed over different computers, which may be owned and managed by different companies.

# Advantages of explicit architecture

---

## ✧ Stakeholder communication

- Architecture may be used as a focus of discussion by system stakeholders.

## ✧ System analysis

- Means that analysis of whether the system can meet its non-functional requirements is possible.

## ✧ Large-scale reuse

- The architecture may be reusable across a range of systems
- Product-line architectures may be developed.

# Architectural representations

---

- ✧ Simple, informal block diagrams showing entities and relationships are the most frequently used method for documenting software architectures.
- ✧ But these have been criticised because they lack semantics, do not show the types of relationships between entities nor the visible properties of entities in the architecture.
- ✧ Depends on the use of architectural models. The requirements for model semantics depends on how the models are used.

# Box and line diagrams

---

- ✧ Very abstract - they do not show the nature of component relationships nor the externally visible properties of the sub-systems.
- ✧ However, useful for communication with stakeholders and for project planning.

# Use of architectural models

---

- ✧ As a way of facilitating discussion about the system design
  - A high-level architectural view of a system is useful for communication with system stakeholders and project planning because it is not cluttered with detail. Stakeholders can relate to it and understand an abstract view of the system. They can then discuss the system as a whole without being confused by detail.
- ✧ As a way of documenting an architecture that has been designed
  - The aim here is to produce a complete system model that shows the different components in a system, their interfaces and their connections.

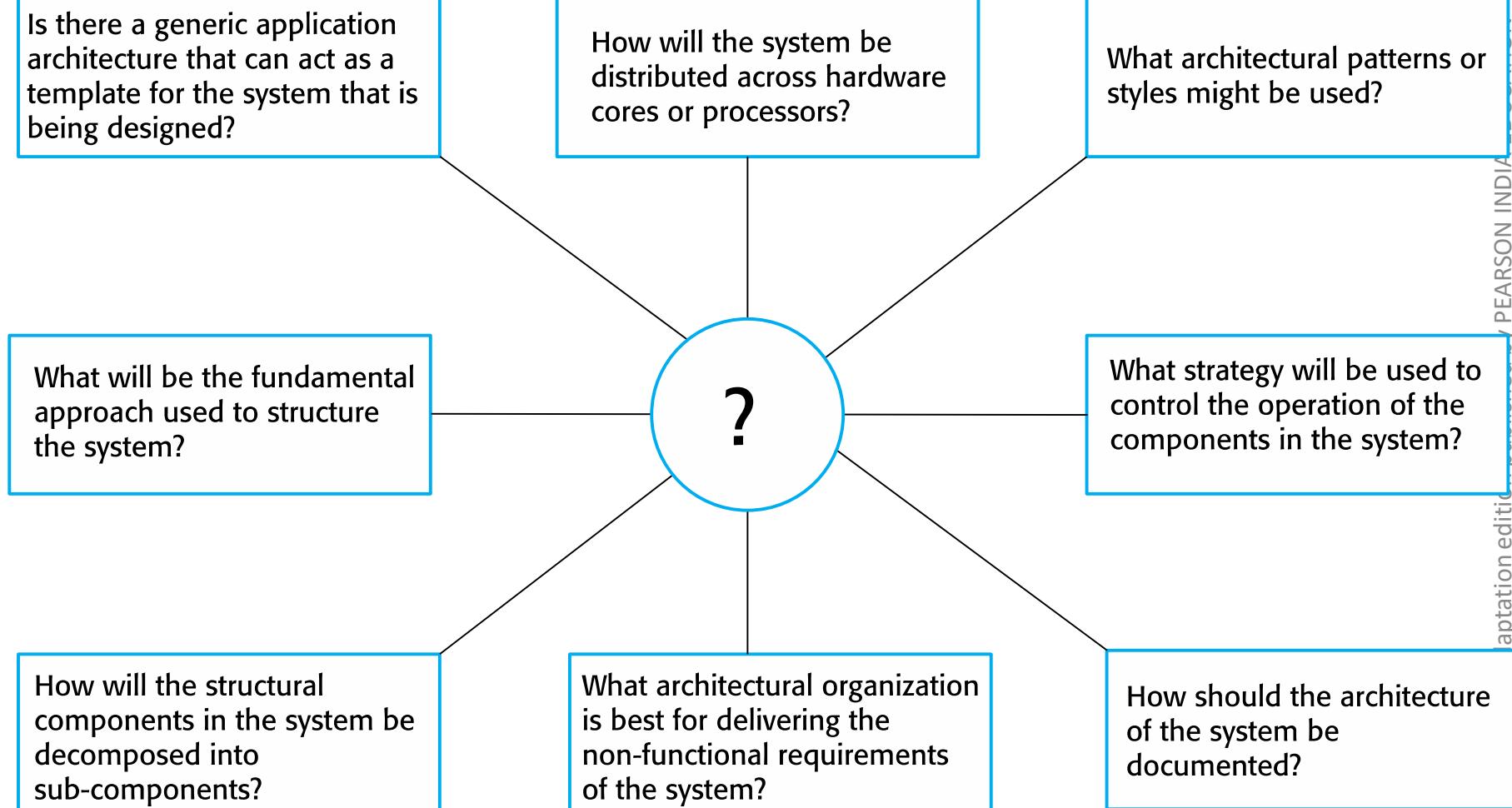
# Architectural design decisions

# Architectural design decisions

---

- ✧ Architectural design is a creative process so the process differs depending on the type of system being developed.
- ✧ However, a number of common decisions span all design processes and these decisions affect the non-functional characteristics of the system.

# Architectural design decisions



# Architecture reuse

---

- ✧ Systems in the same domain often have similar architectures that reflect domain concepts.
- ✧ Application product lines are built around a core architecture with variants that satisfy particular customer requirements.
- ✧ The architecture of a system may be designed around one of more architectural patterns or ‘styles’.
  - These capture the essence of an architecture and can be instantiated in different ways.

# Architecture and system characteristics

---

- ✧ Performance
  - Localise critical operations and minimise communications. Use large rather than fine-grain components.
- ✧ Security
  - Use a layered architecture with critical assets in the inner layers.
- ✧ Safety
  - Localise safety-critical features in a small number of subsystems.
- ✧ Availability
  - Include redundant components and mechanisms for fault tolerance.
- ✧ Maintainability
  - Use fine-grain, replaceable components.

---

# **Architectural views**

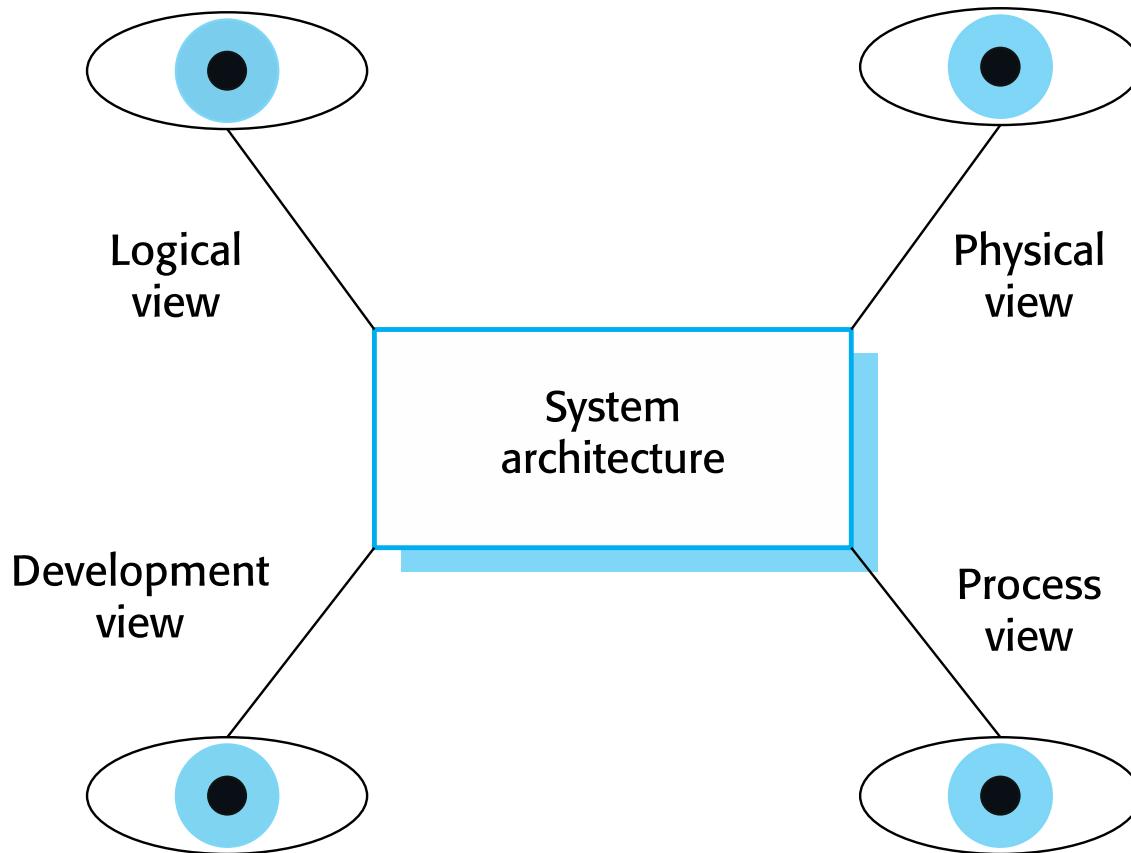
# Architectural views

---

- ✧ What views or perspectives are useful when designing and documenting a system's architecture?
- ✧ What notations should be used for describing architectural models?
- ✧ Each architectural model only shows one view or perspective of the system.
  - It might show how a system is decomposed into modules, how the run-time processes interact or the different ways in which system components are distributed across a network. For both design and documentation, you usually need to present multiple views of the software architecture.

# Architectural views

---



## 4 + 1 view model of software architecture

---

- ✧ A logical view, which shows the key abstractions in the system as objects or object classes.
- ✧ A process view, which shows how, at run-time, the system is composed of interacting processes.
- ✧ A development view, which shows how the software is decomposed for development.
- ✧ A physical view, which shows the system hardware and how software components are distributed across the processors in the system.
- ✧ Related using use cases or scenarios (+1)

# Representing architectural views

---

- ✧ Some people argue that the Unified Modeling Language (UML) is an appropriate notation for describing and documenting system architectures
- ✧ I disagree with this as I do not think that the UML includes abstractions appropriate for high-level system description.
- ✧ Architectural description languages (ADLs) have been developed but are not widely used

# Architectural patterns

# Architectural patterns

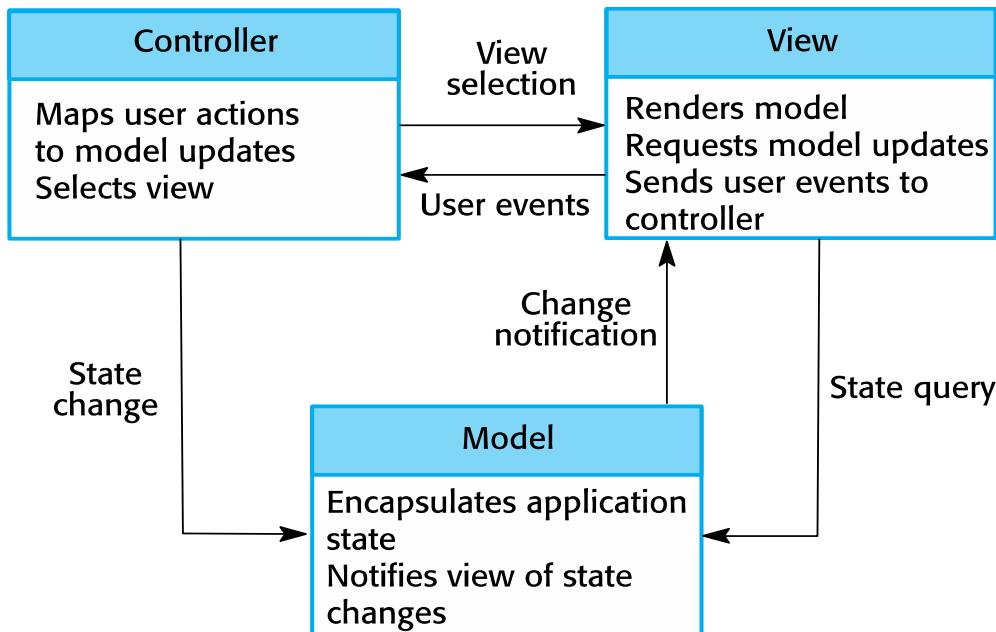
---

- ✧ Patterns are a means of representing, sharing and reusing knowledge.
- ✧ An architectural pattern is a stylized description of good design practice, which has been tried and tested in different environments.
- ✧ Patterns should include information about when they are and when they are not useful.
- ✧ Patterns may be represented using tabular and graphical descriptions.

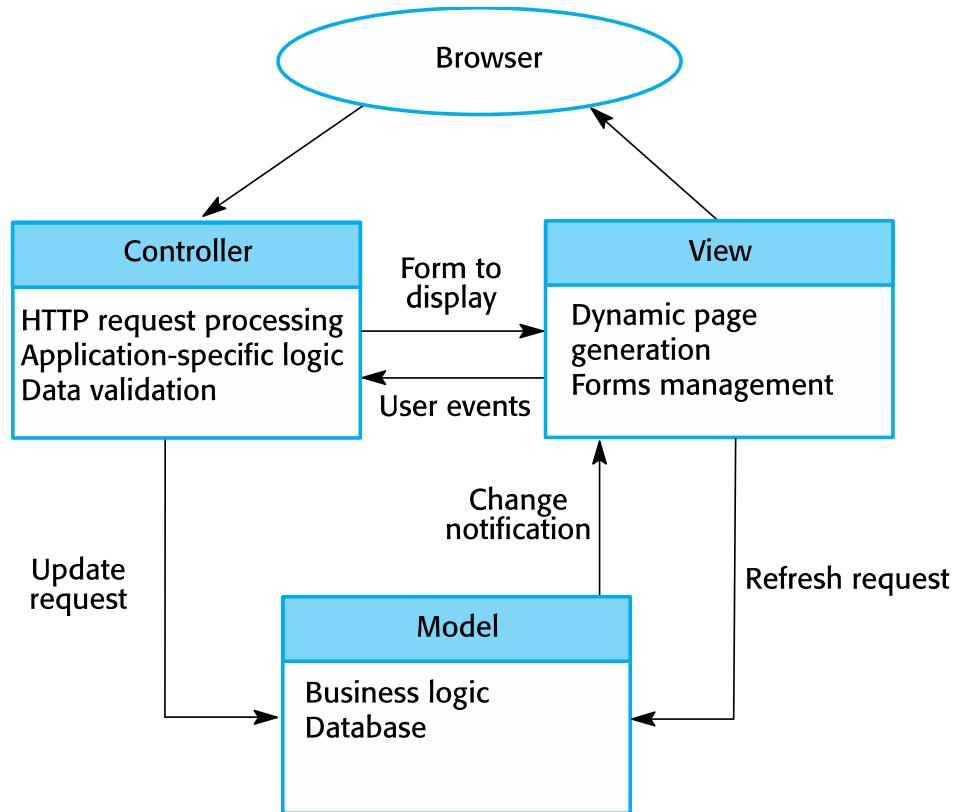
# The Model-View-Controller (MVC) pattern

Name	MVC (Model-View-Controller)
Description	Separates presentation and interaction from the system data. The system is structured into three logical components that interact with each other. The Model component manages the system data and associated operations on that data. The View component defines and manages how the data is presented to the user. The Controller component manages user interaction (e.g., key presses, mouse clicks, etc.) and passes these interactions to the View and the Model. See Figure 6.3.
Example	Figure 6.4 shows the architecture of a web-based application system organized using the MVC pattern.
When used	Used when there are multiple ways to view and interact with data. Also used when the future requirements for interaction and presentation of data are unknown.
Advantages	Allows the data to change independently of its representation and vice versa. Supports presentation of the same data in different ways with changes made in one representation shown in all of them.
Disadvantages	Can involve additional code and code complexity when the data model and interactions are simple.

# The organization of the Model-View-Controller



# Web application architecture using the MVC pattern



# Layered architecture

---

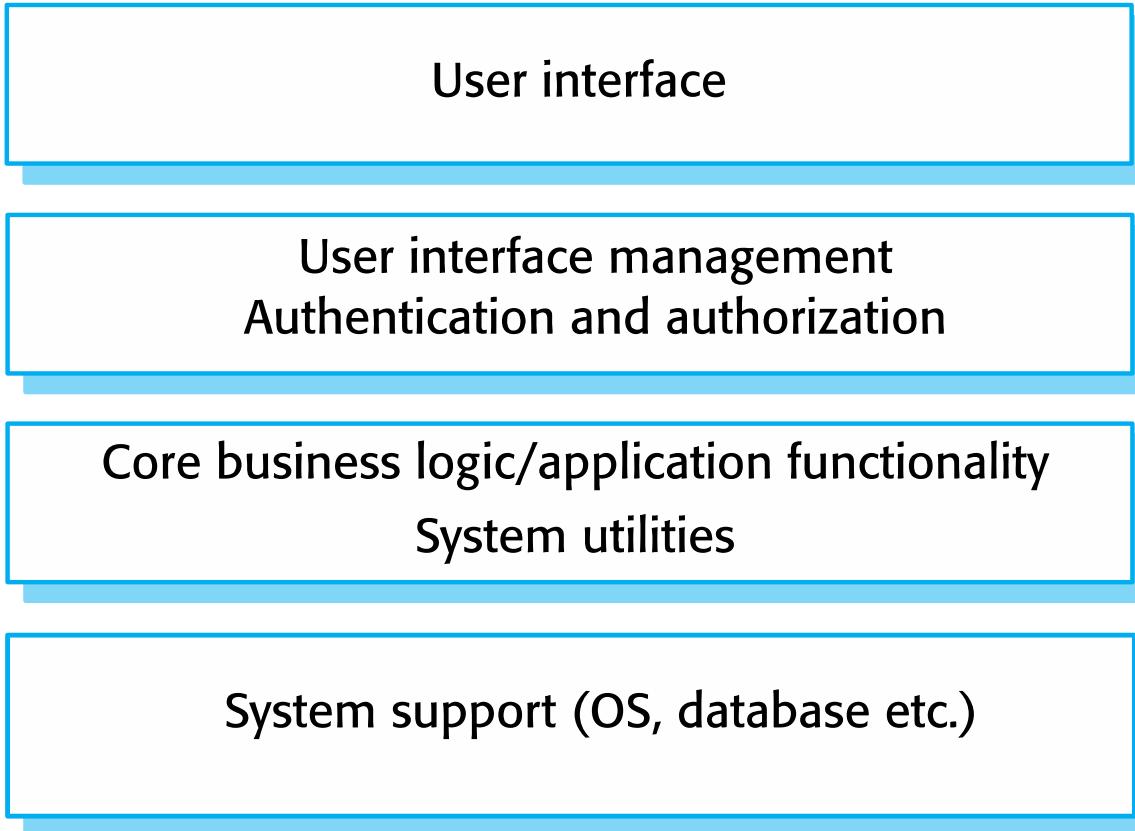
- ✧ Used to model the interfacing of sub-systems.
- ✧ Organises the system into a set of layers (or abstract machines) each of which provide a set of services.
- ✧ Supports the incremental development of sub-systems in different layers. When a layer interface changes, only the adjacent layer is affected.
- ✧ However, often artificial to structure systems in this way.

# The Layered architecture pattern

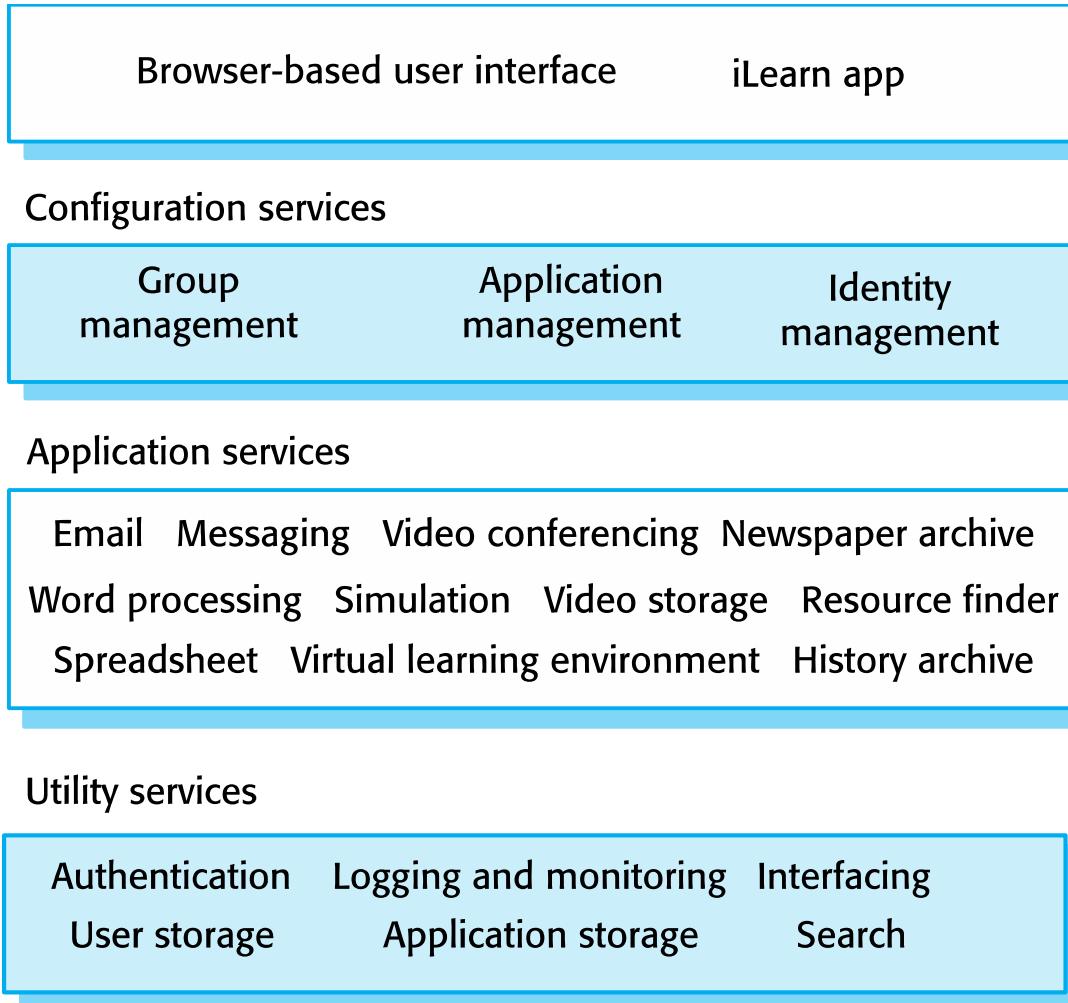
Name	Layered architecture
Description	Organizes the system into layers with related functionality associated with each layer. A layer provides services to the layer above it so the lowest-level layers represent core services that are likely to be used throughout the system. See Figure 6.6.
Example	A layered model of a system for sharing copyright documents held in different libraries, as shown in Figure 6.7.
When used	Used when building new facilities on top of existing systems; when the development is spread across several teams with each team responsibility for a layer of functionality; when there is a requirement for multi-level security.
Advantages	Allows replacement of entire layers so long as the interface is maintained. Redundant facilities (e.g., authentication) can be provided in each layer to increase the dependability of the system.
Disadvantages	In practice, providing a clean separation between layers is often difficult and a high-level layer may have to interact directly with lower-level layers rather than through the layer immediately below it. Performance can be a problem because of multiple levels of interpretation of a service request as it is processed at each layer.

# A generic layered architecture

---



# The architecture of the iLearn system



# Repository architecture

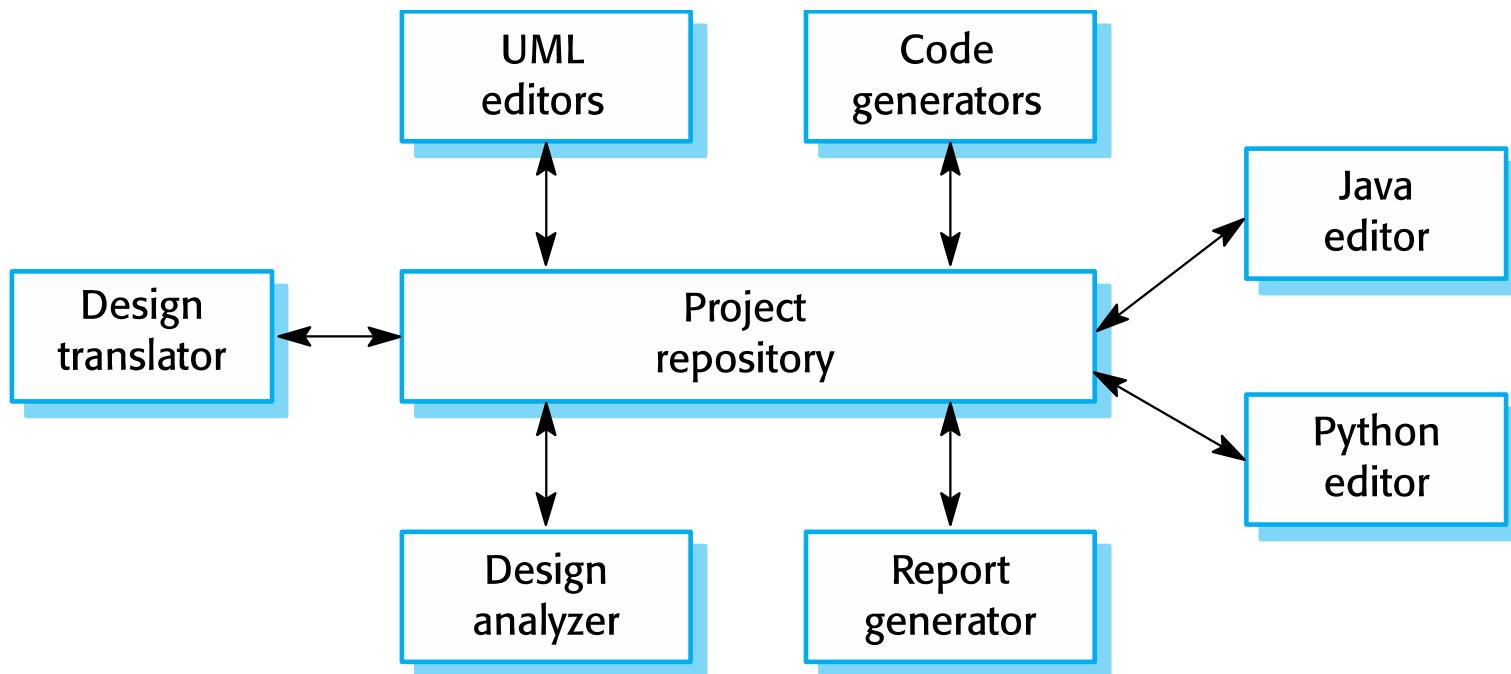
---

- ✧ Sub-systems must exchange data. This may be done in two ways:
  - Shared data is held in a central database or repository and may be accessed by all sub-systems;
  - Each sub-system maintains its own database and passes data explicitly to other sub-systems.
- ✧ When large amounts of data are to be shared, the repository model of sharing is most commonly used as this is an efficient data sharing mechanism.

# The Repository pattern

Name	Repository
<b>Description</b>	All data in a system is managed in a central repository that is accessible to all system components. Components do not interact directly, only through the repository.
<b>Example</b>	Figure 6.9 is an example of an IDE where the components use a repository of system design information. Each software tool generates information which is then available for use by other tools.
<b>When used</b>	You should use this pattern when you have a system in which large volumes of information are generated that has to be stored for a long time. You may also use it in data-driven systems where the inclusion of data in the repository triggers an action or tool.
<b>Advantages</b>	Components can be independent—they do not need to know of the existence of other components. Changes made by one component can be propagated to all components. All data can be managed consistently (e.g., backups done at the same time) as it is all in one place.
<b>Disadvantages</b>	The repository is a single point of failure so problems in the repository affect the whole system. May be inefficiencies in organizing all communication through the repository. Distributing the repository across several computers may be difficult.

# A repository architecture for an IDE



# Client-server architecture

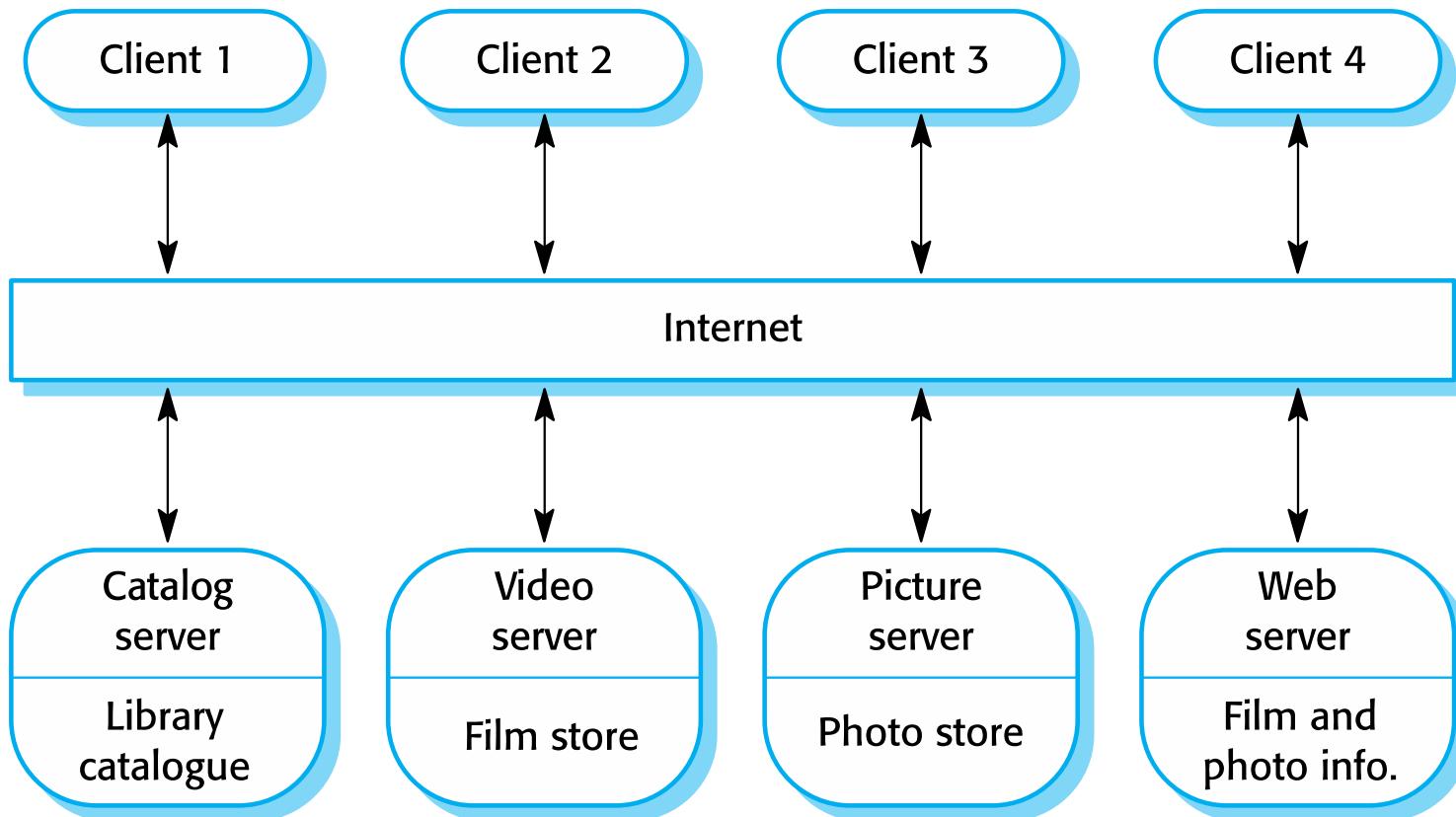
---

- ✧ Distributed system model which shows how data and processing is distributed across a range of components.
  - Can be implemented on a single computer.
- ✧ Set of stand-alone servers which provide specific services such as printing, data management, etc.
- ✧ Set of clients which call on these services.
- ✧ Network which allows clients to access servers.

# The Client–server pattern

Name	Client-server
<b>Description</b>	In a client–server architecture, the functionality of the system is organized into services, with each service delivered from a separate server. Clients are users of these services and access servers to make use of them.
<b>Example</b>	Figure 6.11 is an example of a film and video/DVD library organized as a client–server system.
<b>When used</b>	Used when data in a shared database has to be accessed from a range of locations. Because servers can be replicated, may also be used when the load on a system is variable.
<b>Advantages</b>	The principal advantage of this model is that servers can be distributed across a network. General functionality (e.g., a printing service) can be available to all clients and does not need to be implemented by all services.
<b>Disadvantages</b>	Each service is a single point of failure so susceptible to denial of service attacks or server failure. Performance may be unpredictable because it depends on the network as well as the system. May be management problems if servers are owned by different organizations.

# A client–server architecture for a film library



# Pipe and filter architecture

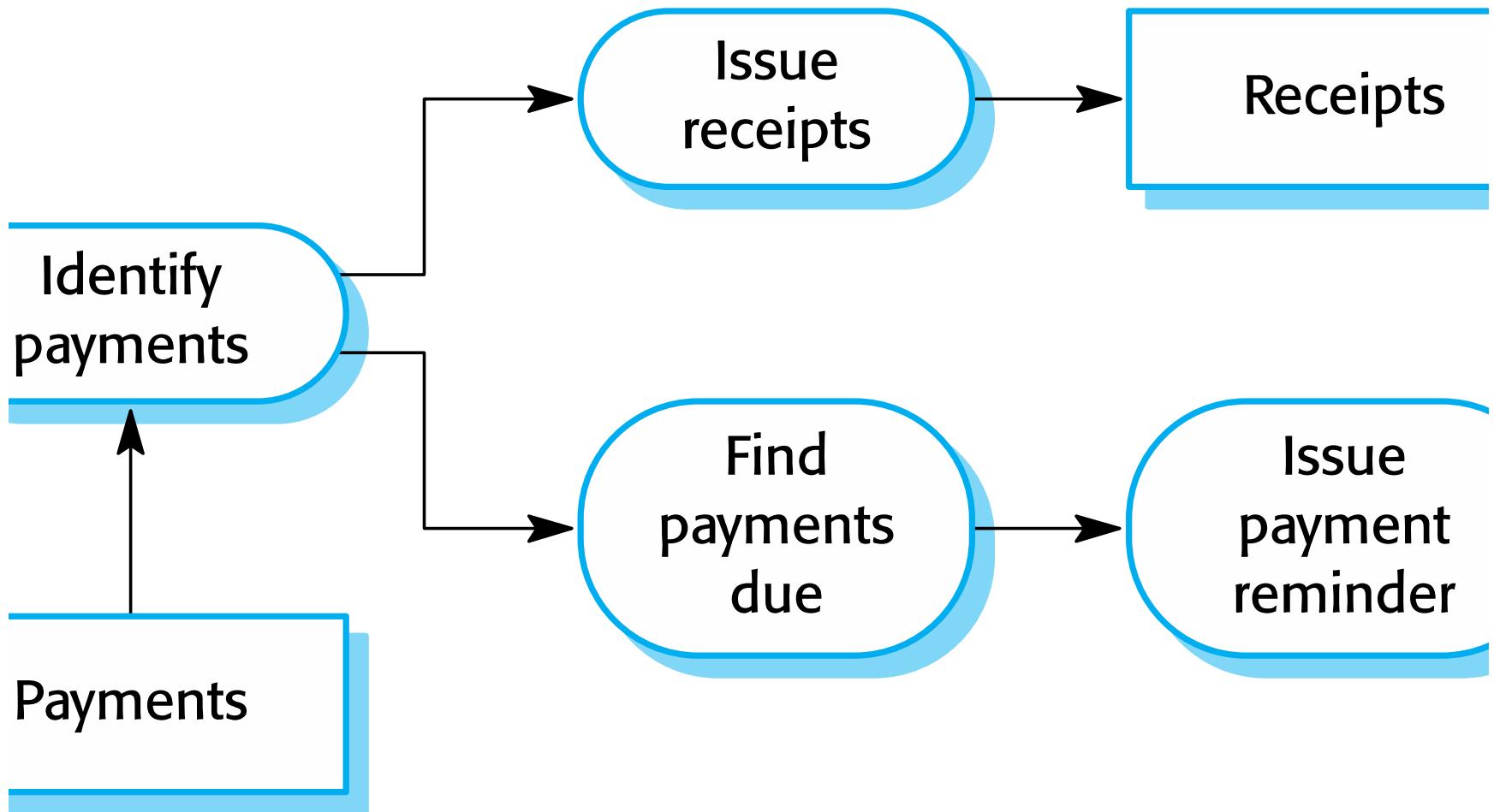
---

- ✧ Functional transformations process their inputs to produce outputs.
- ✧ May be referred to as a pipe and filter model (as in UNIX shell).
- ✧ Variants of this approach are very common. When transformations are sequential, this is a batch sequential model which is extensively used in data processing systems.
- ✧ Not really suitable for interactive systems.

# The pipe and filter pattern

Name	Pipe and filter
<b>Description</b>	The processing of the data in a system is organized so that each processing component (filter) is discrete and carries out one type of data transformation. The data flows (as in a pipe) from one component to another for processing.
<b>Example</b>	Figure 6.13 is an example of a pipe and filter system used for processing invoices.
<b>When used</b>	Commonly used in data processing applications (both batch- and transaction-based) where inputs are processed in separate stages to generate related outputs.
<b>Advantages</b>	Easy to understand and supports transformation reuse. Workflow style matches the structure of many business processes. Evolution by adding transformations is straightforward. Can be implemented as either a sequential or concurrent system.
<b>Disadvantages</b>	The format for data transfer has to be agreed upon between communicating transformations. Each transformation must parse its input and unparse its output to the agreed form. This increases system overhead and may mean that it is impossible to reuse functional transformations that use incompatible data structures.

# An example of the pipe and filter architecture used in a payments system



# Application architectures

# Application architectures

---

- ✧ Application systems are designed to meet an organisational need.
- ✧ As businesses have much in common, their application systems also tend to have a common architecture that reflects the application requirements.
- ✧ A generic application architecture is an architecture for a type of software system that may be configured and adapted to create a system that meets specific requirements.

# Use of application architectures

---

- ✧ As a starting point for architectural design.
- ✧ As a design checklist.
- ✧ As a way of organising the work of the development team.
- ✧ As a means of assessing components for reuse.
- ✧ As a vocabulary for talking about application types.

# Examples of application types

---

- ✧ Data processing applications
  - Data driven applications that process data in batches without explicit user intervention during the processing.
- ✧ Transaction processing applications
  - Data-centred applications that process user requests and update information in a system database.
- ✧ Event processing systems
  - Applications where system actions depend on interpreting events from the system's environment.
- ✧ Language processing systems
  - Applications where the users' intentions are specified in a formal language that is processed and interpreted by the system.

# Application type examples

---

- ✧ Two very widely used generic application architectures are transaction processing systems and language processing systems.
- ✧ Transaction processing systems
  - E-commerce systems;
  - Reservation systems.
- ✧ Language processing systems
  - Compilers;
  - Command interpreters.

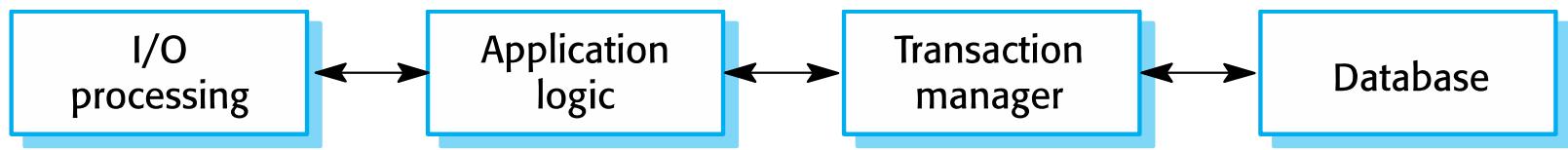
# Transaction processing systems

---

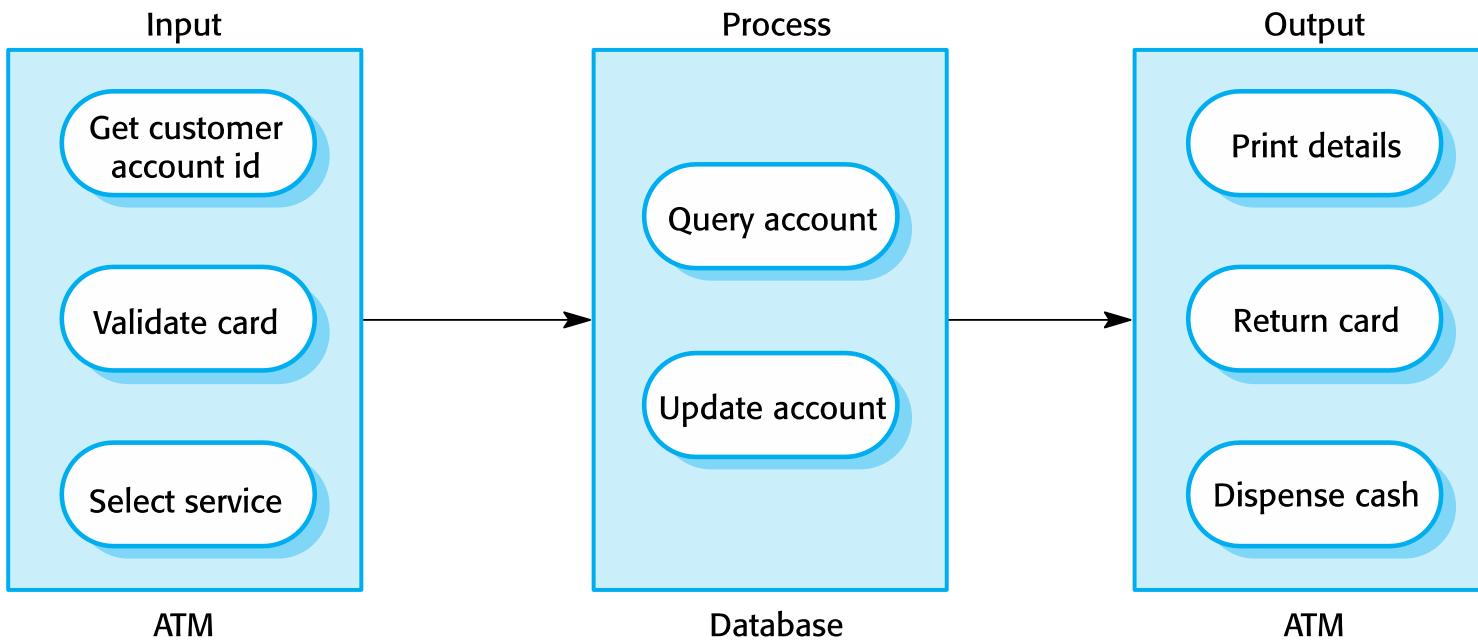
- ✧ Process user requests for information from a database or requests to update the database.
- ✧ From a user perspective a transaction is:
  - Any coherent sequence of operations that satisfies a goal;
  - For example - find the times of flights from London to Paris.
- ✧ Users make asynchronous requests for service which are then processed by a transaction manager.

# The structure of transaction processing applications

---



# The software architecture of an ATM system



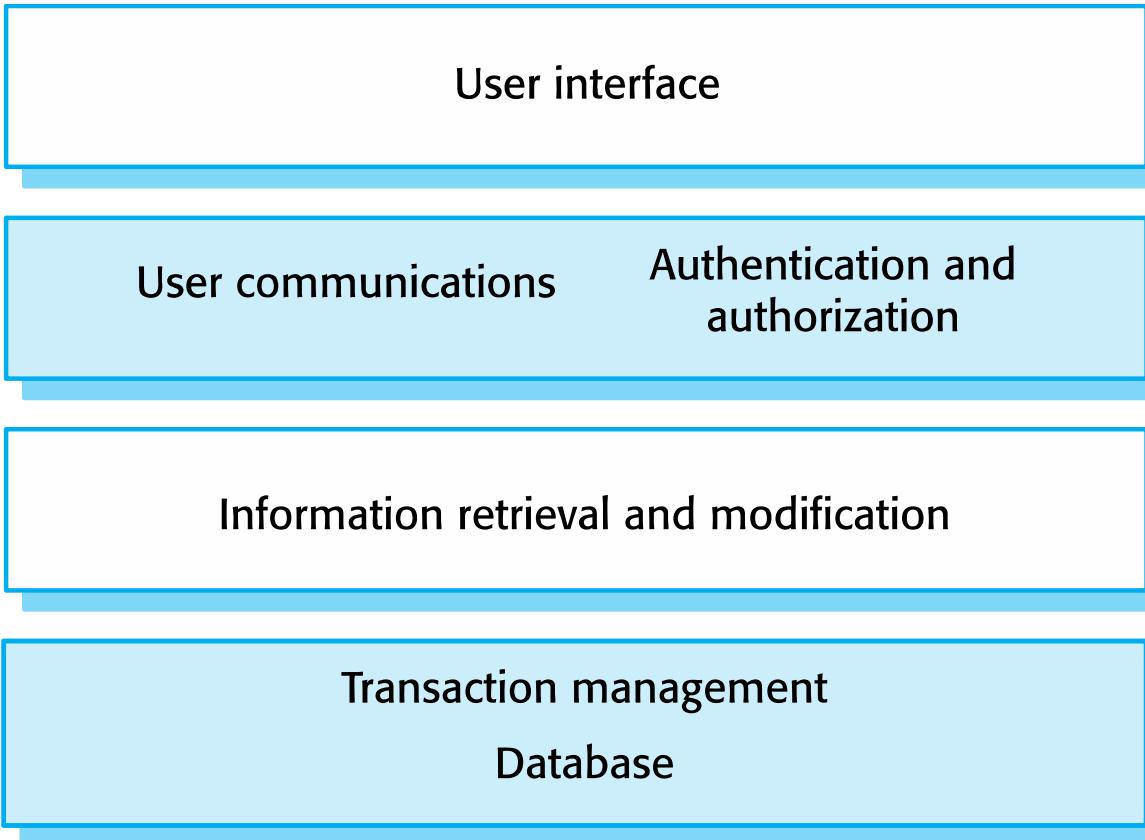
# Information systems architecture

---

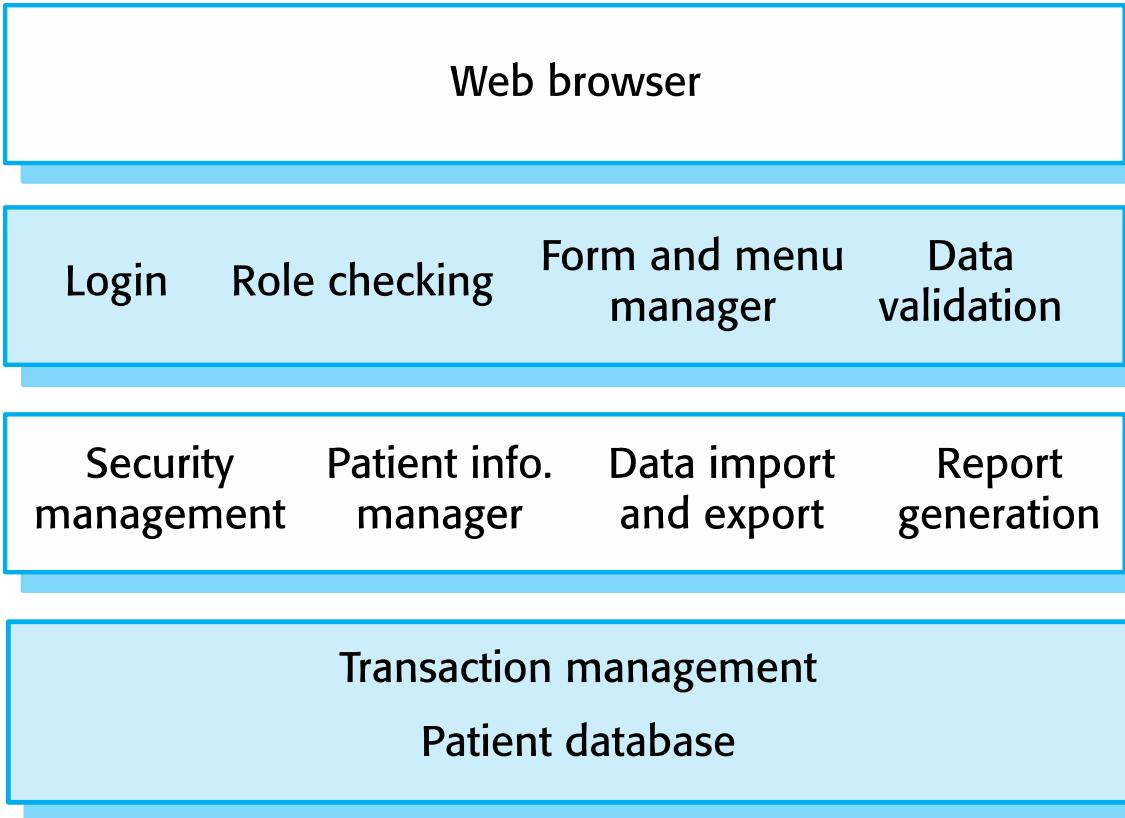
- ✧ Information systems have a generic architecture that can be organised as a layered architecture.
- ✧ These are transaction-based systems as interaction with these systems generally involves database transactions.
- ✧ Layers include:
  - The user interface
  - User communications
  - Information retrieval
  - System database

# Layered information system architecture

---



# The architecture of the Mentcare system



# Web-based information systems

---

- ✧ Information and resource management systems are now usually web-based systems where the user interfaces are implemented using a web browser.
- ✧ For example, e-commerce systems are Internet-based resource management systems that accept electronic orders for goods or services and then arrange delivery of these goods or services to the customer.
- ✧ In an e-commerce system, the application-specific layer includes additional functionality supporting a ‘shopping cart’ in which users can place a number of items in separate transactions, then pay for them all together in a single transaction.

# Server implementation

---

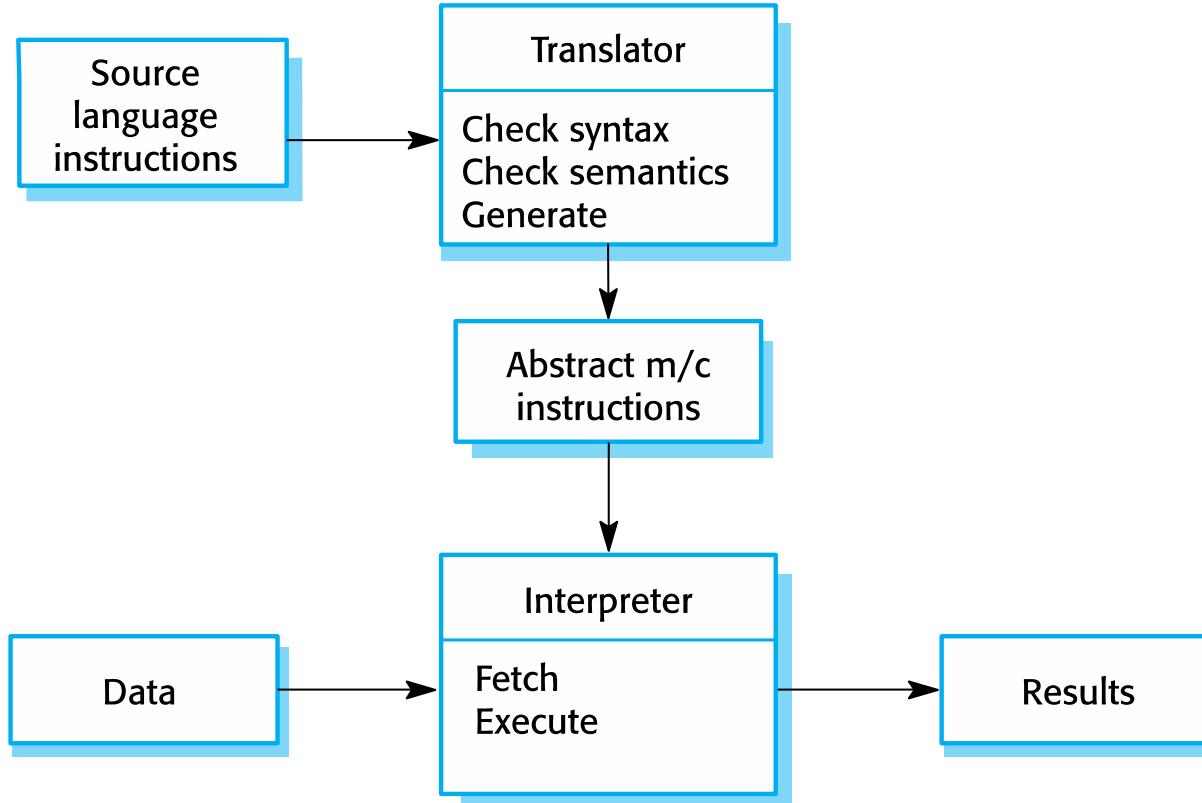
- ✧ These systems are often implemented as multi-tier client server/architectures (discussed in Chapter 17)
  - The web server is responsible for all user communications, with the user interface implemented using a web browser;
  - The application server is responsible for implementing application-specific logic as well as information storage and retrieval requests;
  - The database server moves information to and from the database and handles transaction management.

# Language processing systems

---

- ✧ Accept a natural or artificial language as input and generate some other representation of that language.
- ✧ May include an interpreter to act on the instructions in the language that is being processed.
- ✧ Used in situations where the easiest way to solve a problem is to describe an algorithm or describe the system data
  - Meta-case tools process tool descriptions, method rules, etc and generate tools.

# The architecture of a language processing system



# Compiler components

---

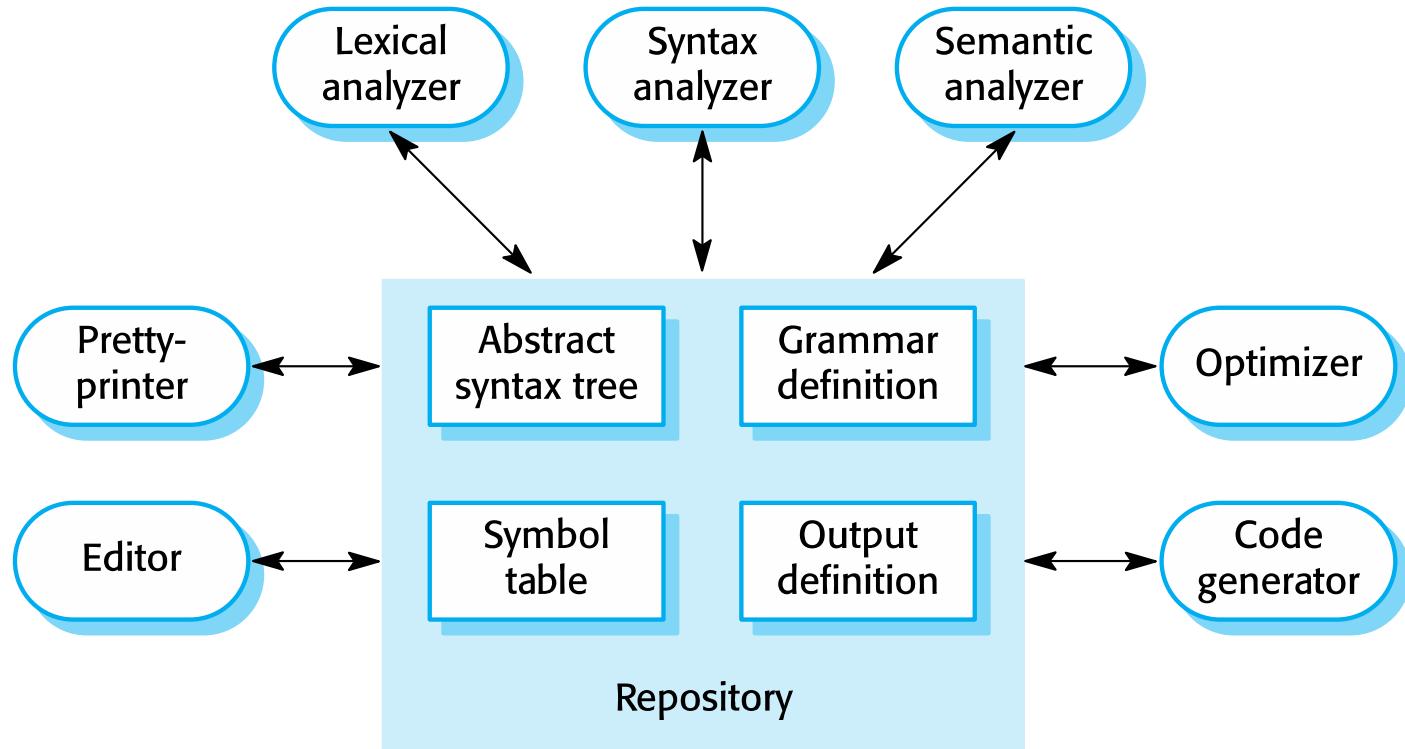
- ✧ A lexical analyzer, which takes input language tokens and converts them to an internal form.
- ✧ A symbol table, which holds information about the names of entities (variables, class names, object names, etc.) used in the text that is being translated.
- ✧ A syntax analyzer, which checks the syntax of the language being translated.
- ✧ A syntax tree, which is an internal structure representing the program being compiled.

# Compiler components

---

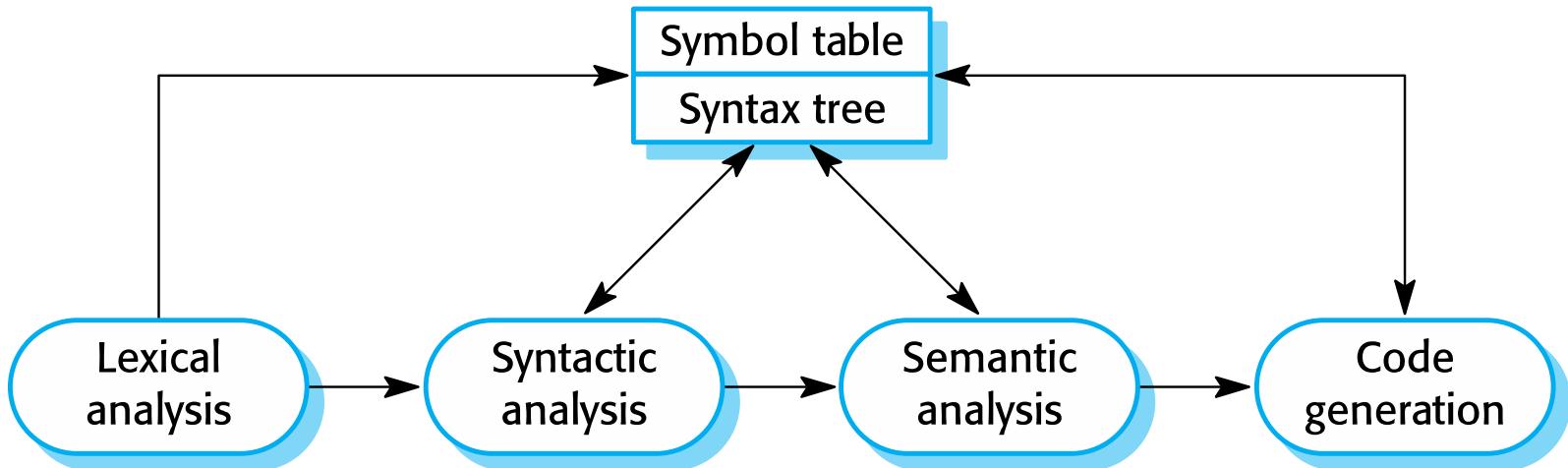
- ✧ A semantic analyzer that uses information from the syntax tree and the symbol table to check the semantic correctness of the input language text.
- ✧ A code generator that ‘walks’ the syntax tree and generates abstract machine code.

# A repository architecture for a language processing system



# A pipe and filter compiler architecture

---



# Key points

---

- ✧ A software architecture is a description of how a software system is organized.
- ✧ Architectural design decisions include decisions on the type of application, the distribution of the system, the architectural styles to be used.
- ✧ Architectures may be documented from several different perspectives or views such as a conceptual view, a logical view, a process view, and a development view.
- ✧ Architectural patterns are a means of reusing knowledge about generic system architectures. They describe the architecture, explain when it may be used and describe its advantages and disadvantages.

# Key points

---

- ✧ Models of application systems architectures help us understand and compare applications, validate application system designs and assess large-scale components for reuse.
- ✧ Transaction processing systems are interactive systems that allow information in a database to be remotely accessed and modified by a number of users.
- ✧ Language processing systems are used to translate texts from one language into another and to carry out the instructions specified in the input language. They include a translator and an abstract machine that executes the generated language.

# **Chapter 7 – Design and Implementation**

# Topics covered

---

- ✧ Object-oriented design using the UML
- ✧ Design patterns
- ✧ Implementation issues
- ✧ Open source development

# Design and implementation

---

- ✧ 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 inter-leaved.
  - 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.

# Build or buy

---

- ✧ 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.
- ✧ When you develop an application in this way, the design process becomes concerned with how to use the configuration features of that system to deliver the system requirements.

# **Object-oriented design using the UML**

# 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.

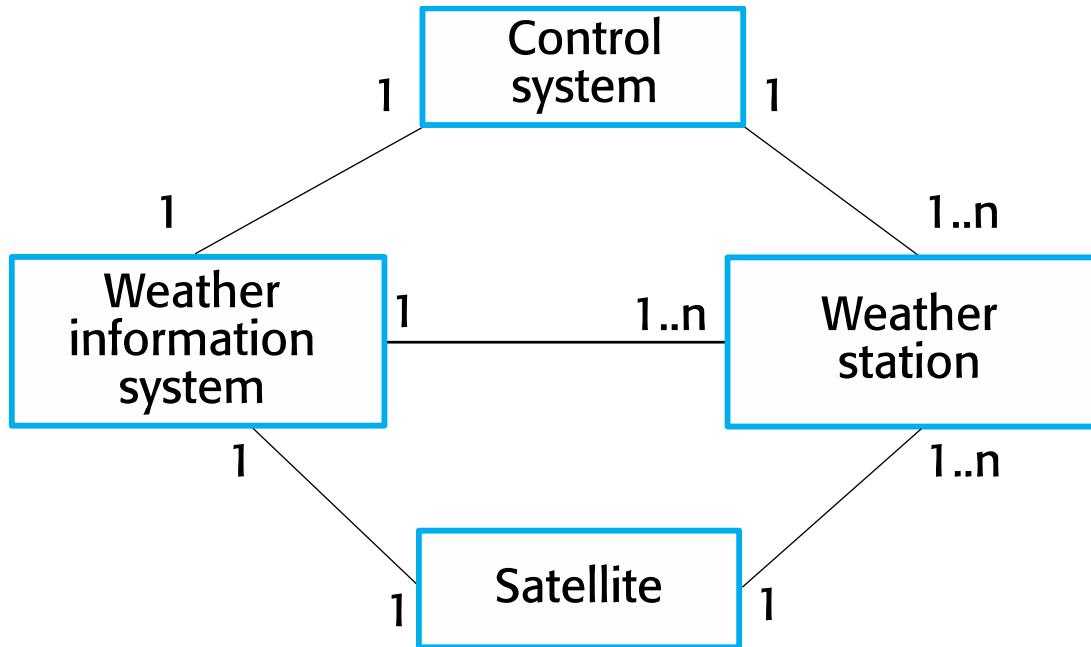
# Context and interaction models

---

- ✧ 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.

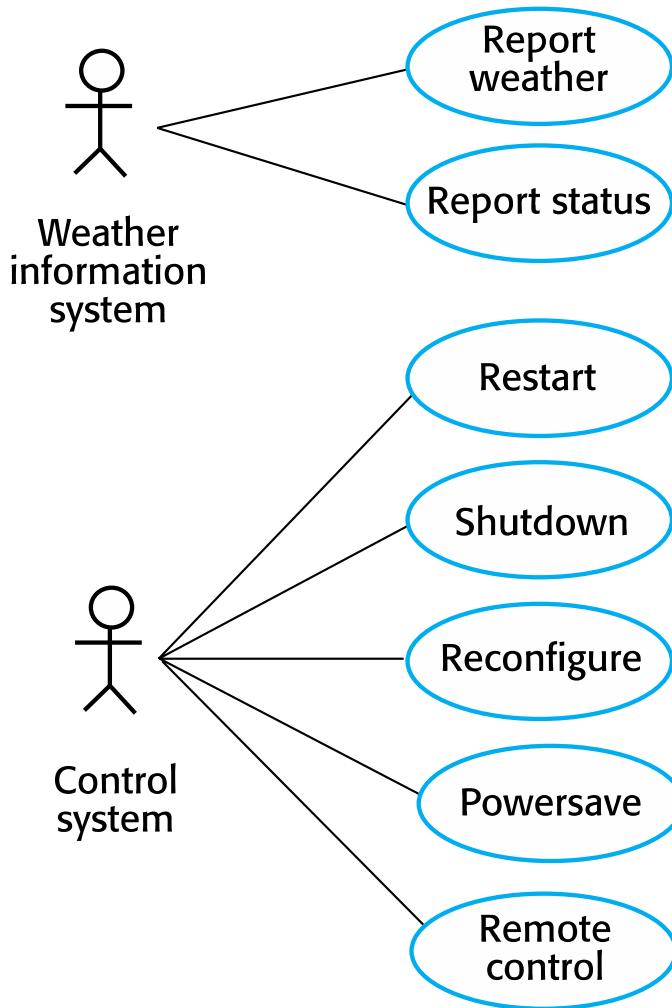
# System context for the weather station

---



# Weather station use cases

---



# Use case description—Report weather

System	Weather station
Use case	Report weather
Actors	Weather information system, Weather station
Description	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 is 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.

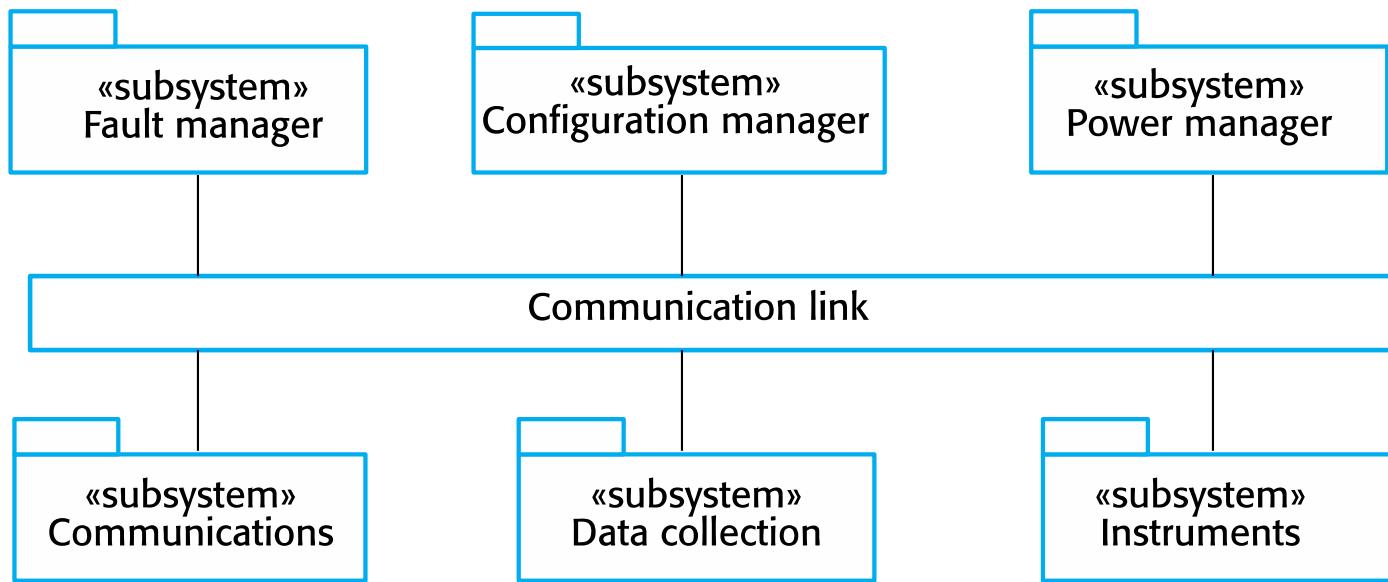
# Architectural design

---

- ✧ 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.

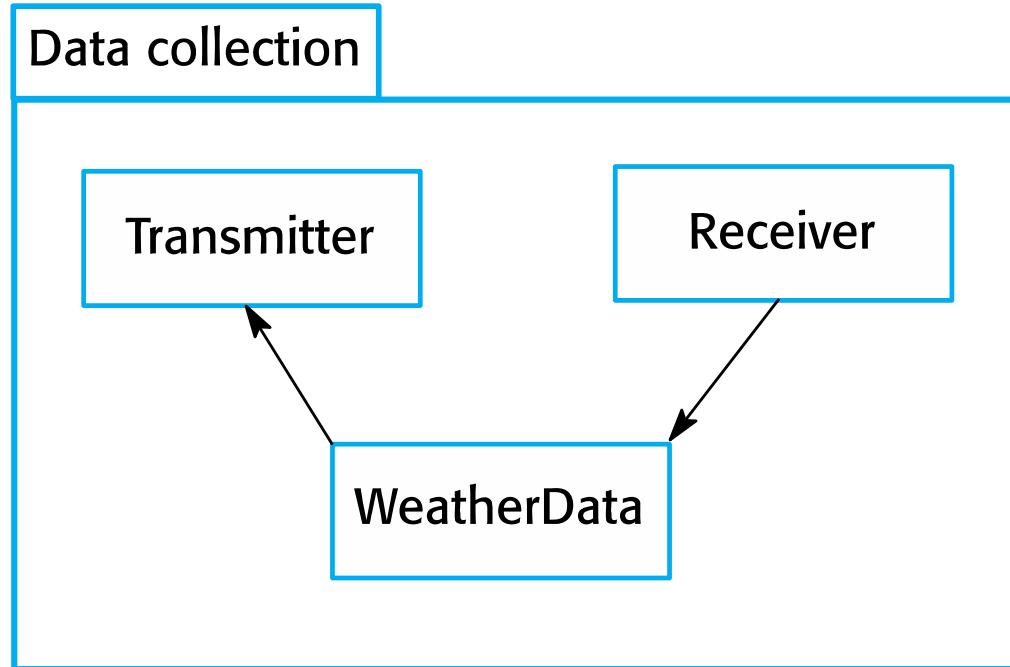
# High-level architecture of the weather station

---



# Architecture of data collection 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.
- ✧ 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 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.

# Weather station object classes

```
reportWeather ()  
reportStatus ()  
powerSave (instruments)  
remoteControl (commands)  
reconfigure (commands)  
restart (instruments)  
shutdown (instruments)
```

```
groundTemperatures  
windSpeeds  
windDirections  
pressures  
rainfall
```

```
collect ()  
summarize ()
```

## Ground thermometer

```
gt_Ident  
temperature
```

## Anemometer

```
an_Ident  
windSpeed  
windDirection
```

## Barometer

```
bar_Ident  
pressure  
height
```

# Design models

---

- ✧ Design models show the objects and object classes and relationships between these entities.
- ✧ There are two kinds of design model:
  - Structural 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.

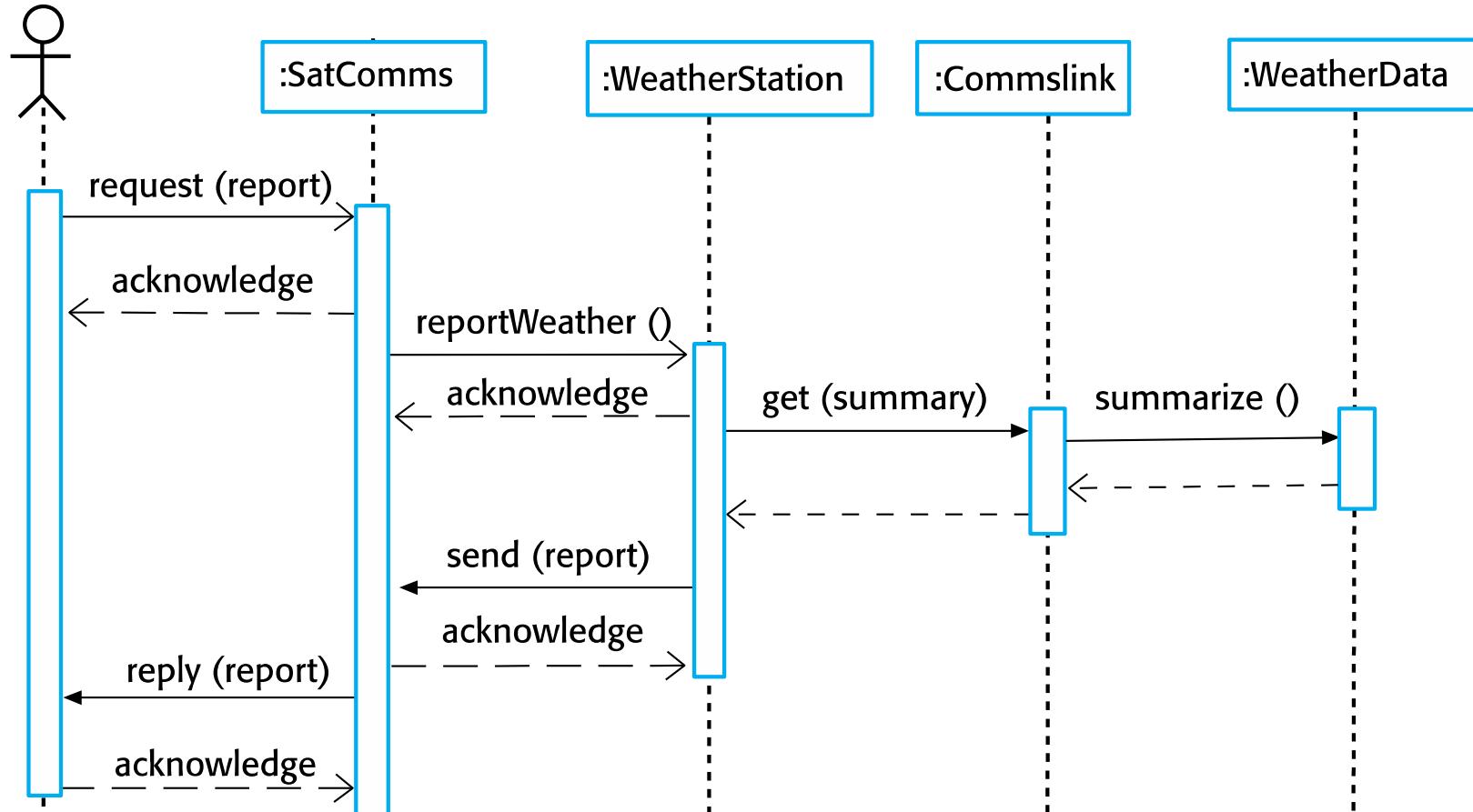
# Sequence models

---

- ✧ 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.

# Sequence diagram describing data collection

information system

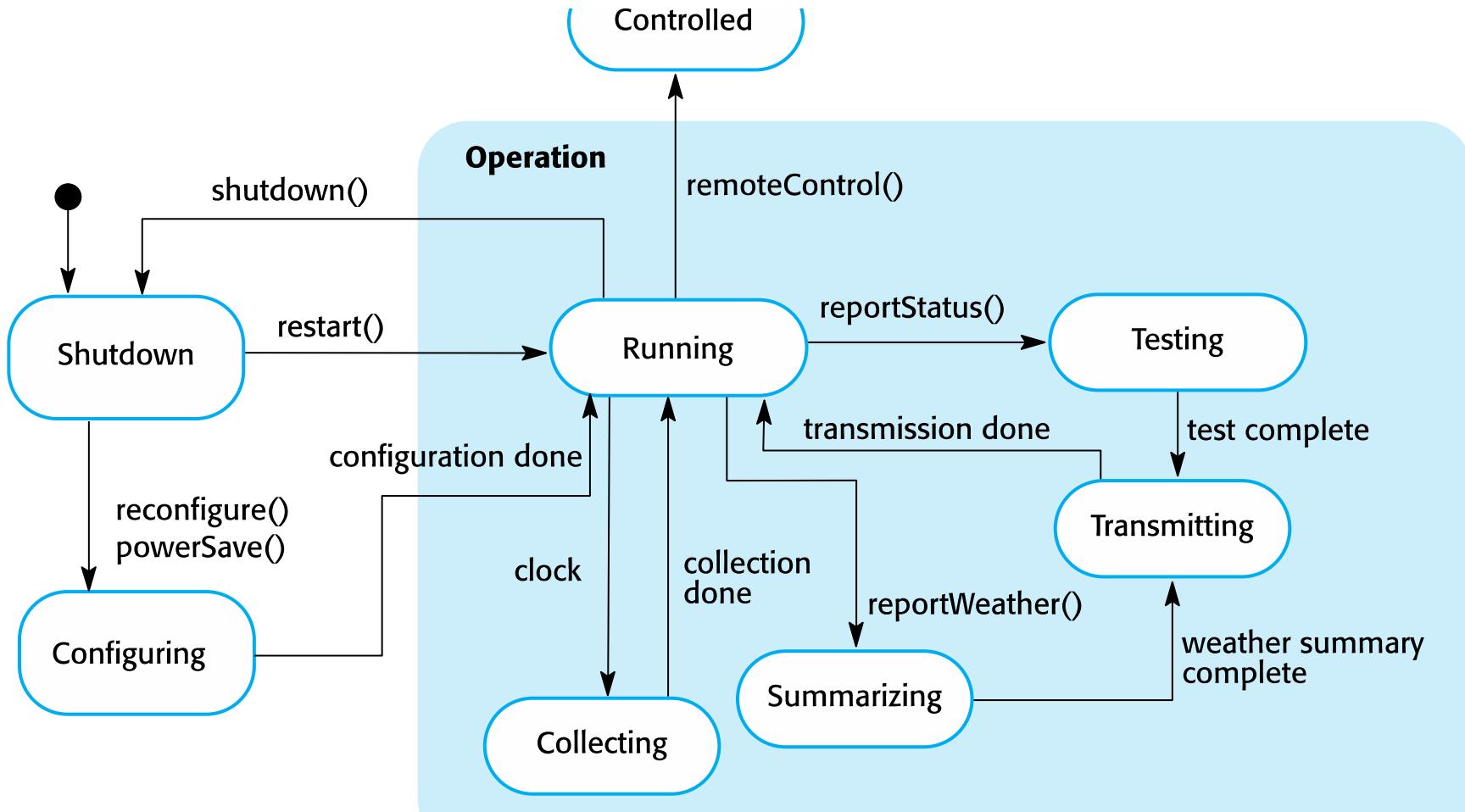


# State diagrams

---

- ✧ 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.

# Weather station state 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.

# Weather station interfaces

---

**«interface»**  
**Reporting**

weatherReport (WS-Ident): Wreport  
statusReport (WS-Ident): Sreport

**«interface»**  
**Remote Control**

startInstrument(instrument): iStatus  
stopInstrument (instrument): iStatus  
collectData (instrument): iStatus  
provideData (instrument ): string

---

# **Design patterns**

# 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.

# Patterns

---

- ✧ *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.*

# 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 are needed.
- ✧ Solution description
  - See slide with UML description.
- ✧ Consequences
  - Optimisations to enhance display performance are impractical.

# The Observer pattern (1)

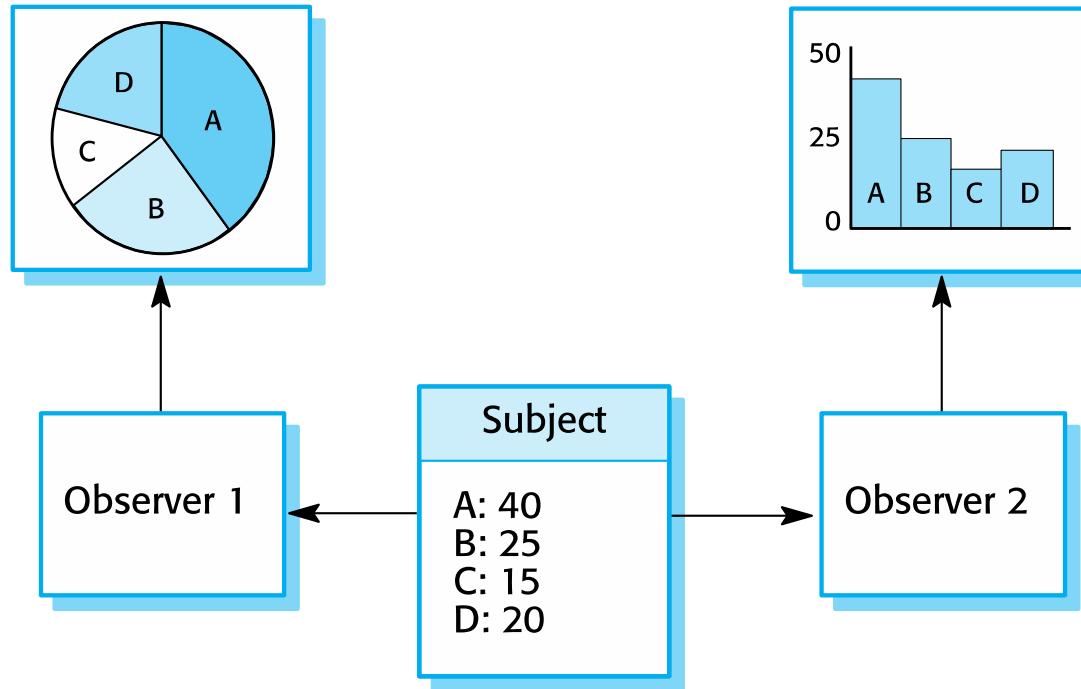
Pattern name	Observer
Description	<p>Separates the display of the state of an object from the object itself and allows alternative displays to be provided. When the object state changes, all displays are automatically notified and updated to reflect the change.</p>
Problem description	<p>In many situations, 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.</p> <p>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.</p>

# The Observer pattern (2)

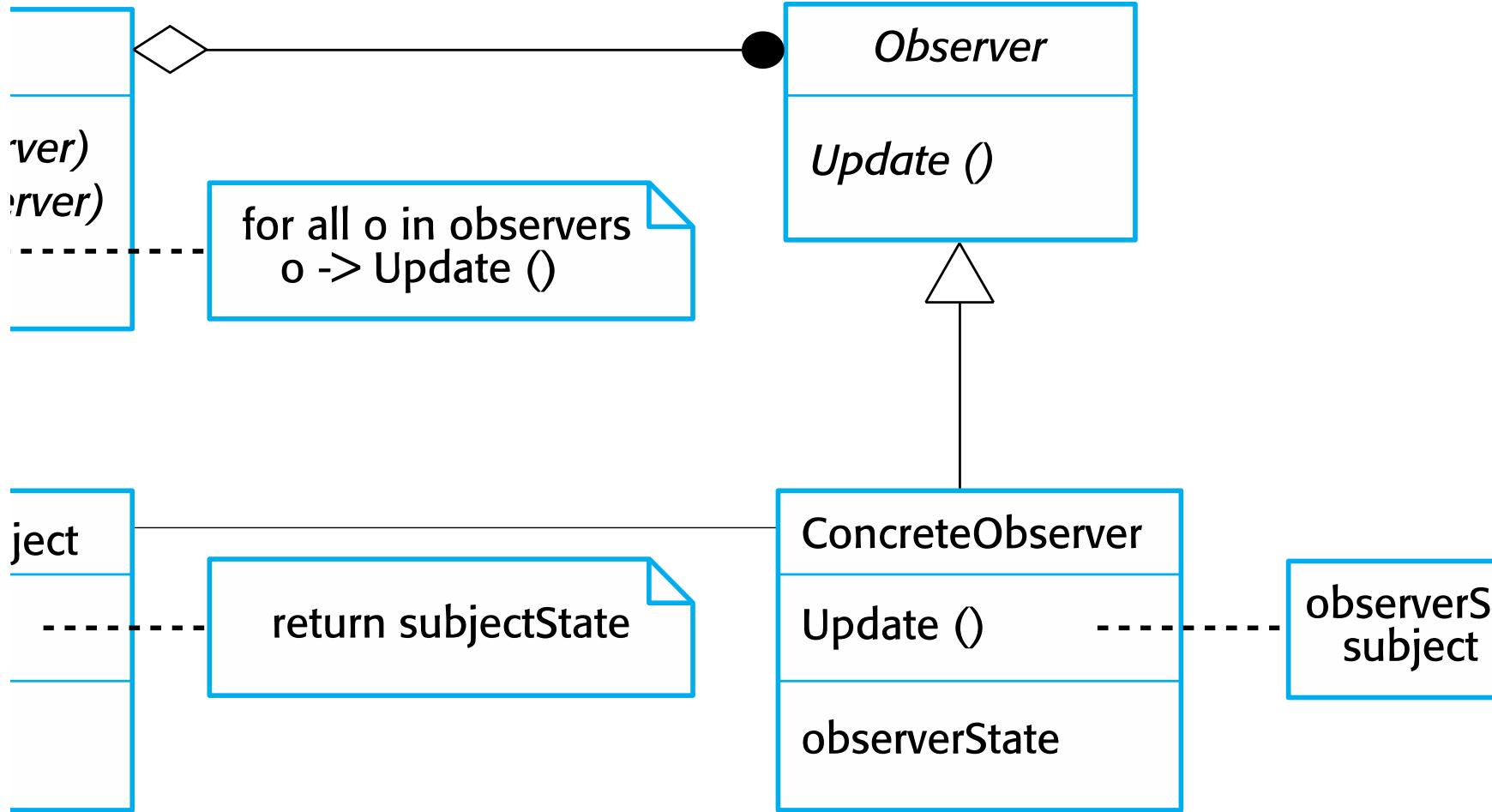
Pattern name	Observer
Solution description	<p>This involves two abstract objects, Subject and Observer, and two concrete objects, ConcreteSubject and ConcreteObject, which inherit the attributes of the related abstract objects. The abstract objects include general operations that are applicable in all 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.</p> <p>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.</p>
Consequences	<p>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.</p>

# Multiple displays using the Observer pattern

---



# 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

# 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

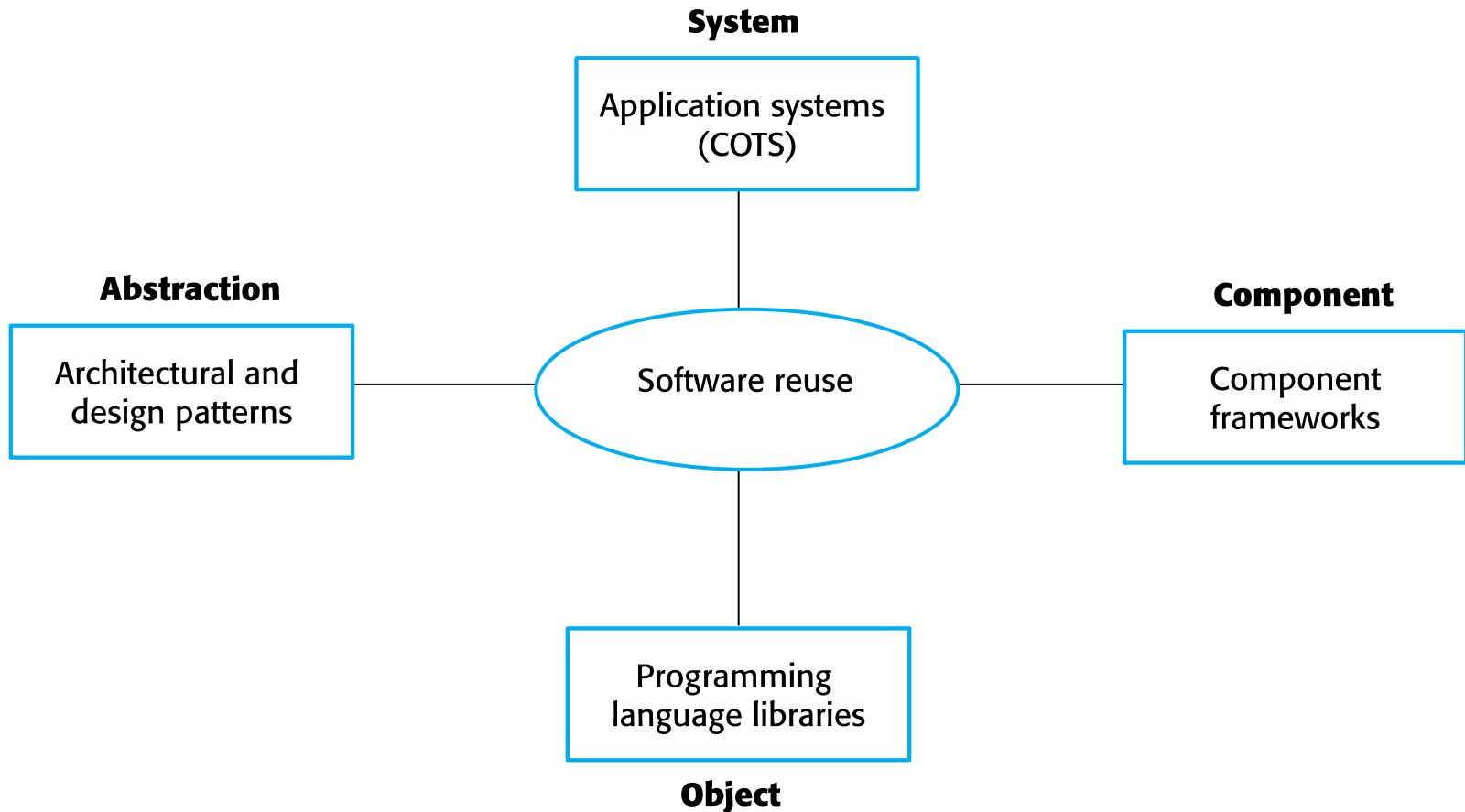
- Components are collections of objects and object classes that you reuse in application systems.

## ✧ The system level

- At this level, you reuse entire application systems.

# Software reuse

---



## 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

---

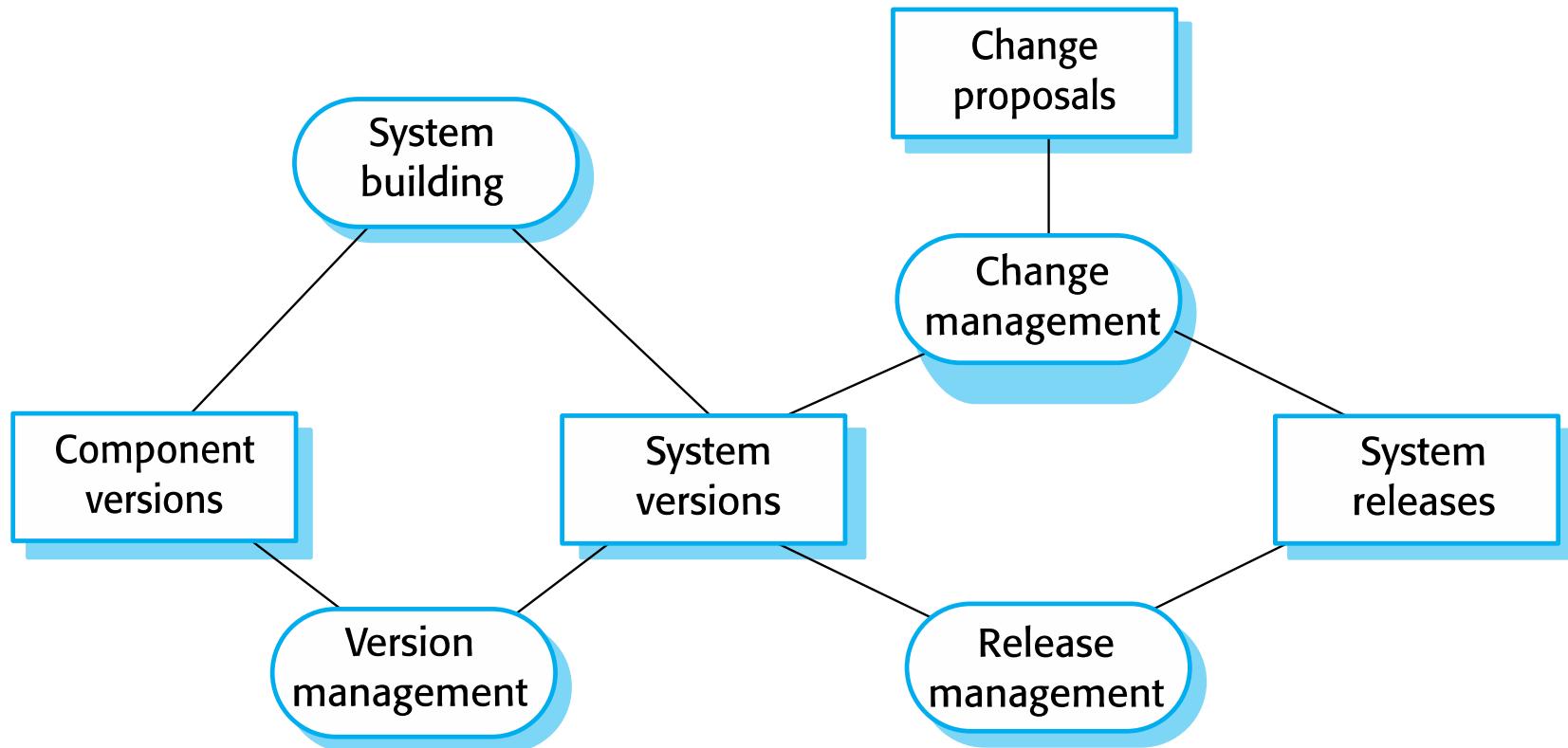
- ✧ Configuration management is the name given to the general process of managing a changing software system.
- ✧ The aim of configuration management is to support the system integration process so that all developers can access the project code and documents in a controlled way, find out what changes have been made, and compile and link components to create a system.
- ✧ See also Chapter 25.

# Configuration management activities

---

- ✧ Version management, where support is provided to keep track of the different versions of software components. Version management systems include facilities to coordinate development by several programmers.
- ✧ 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.

# Configuration management tool interaction

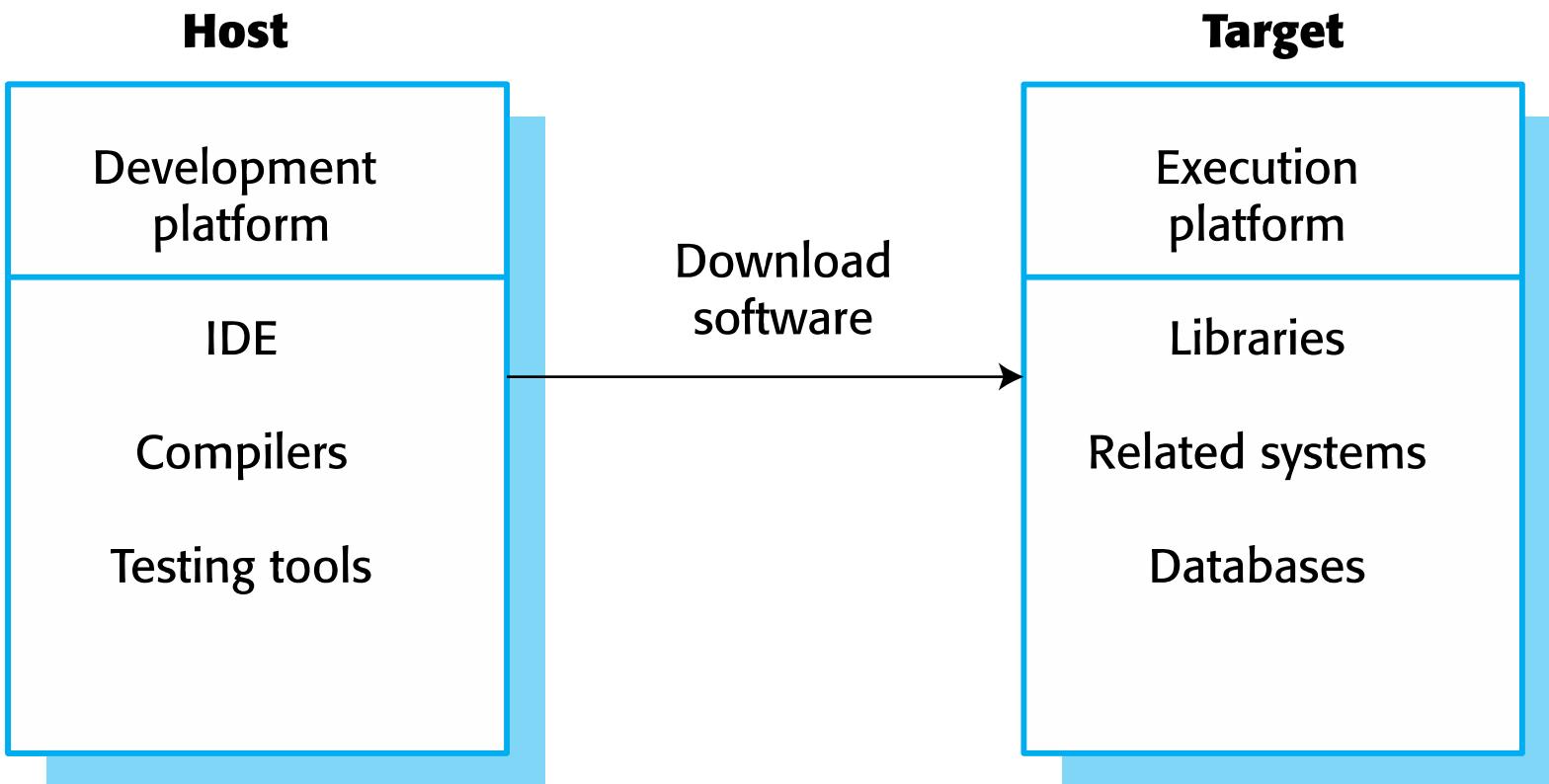


# Host-target development

---

- ✧ 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.

# Host-target development



# 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 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.
- ✧ 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

---

- ✧ 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](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.
- ✧ 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.

# Open source systems

---

- ✧ 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

---

- ✧ 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.
  - 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.

# Key points

---

- ✧ Software design and implementation are inter-leaved activities. The level of detail in the design depends on the type of system and whether you are using a plan-driven or agile approach.
- ✧ The process of object-oriented design includes activities to design the system architecture, identify objects in the system, describe the design using different object models and document the component interfaces.
- ✧ A range of different models may be produced during an object-oriented design process. These include static models (class models, generalization models, association models) and dynamic models (sequence models, state machine models).
- ✧ Component interfaces must be defined precisely so that other objects can use them. A UML interface stereotype may be used to define interfaces.

# Key points

---

- ✧ When developing software, you should always consider the possibility of reusing existing software, either as components, services or complete systems.
- ✧ Configuration management is the process of managing changes to an evolving software system. It is essential when a team of people are cooperating to develop software.
- ✧ Most software development is host-target development. You use an IDE on a host machine to develop the software, which is transferred to a target machine for execution.
- ✧ Open source development involves making the source code of a system publicly available. This means that many people can propose changes and improvements to the software.

# **Chapter 7 – Design and Implementation**

# Topics covered

---

- ✧ Object-oriented design using the UML
- ✧ Design patterns
- ✧ Implementation issues
- ✧ Open source development

# Design and implementation

---

- ✧ 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 inter-leaved.
  - 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.

# Build or buy

---

- ✧ 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.
- ✧ When you develop an application in this way, the design process becomes concerned with how to use the configuration features of that system to deliver the system requirements.

# **Object-oriented design using the UML**

# 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.

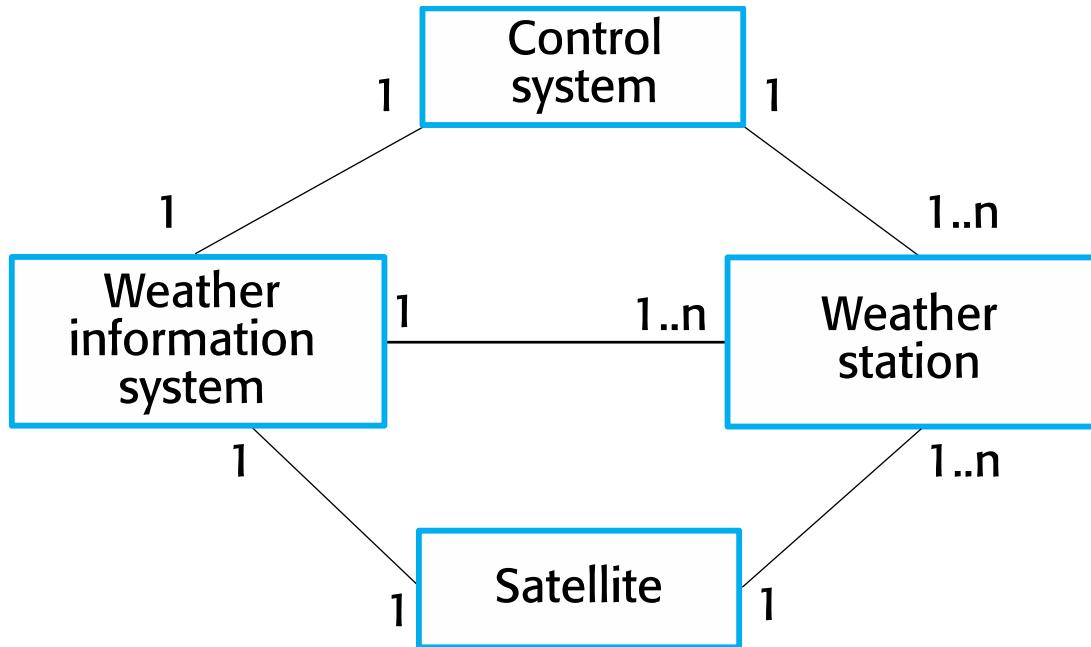
# Context and interaction models

---

- ✧ 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.

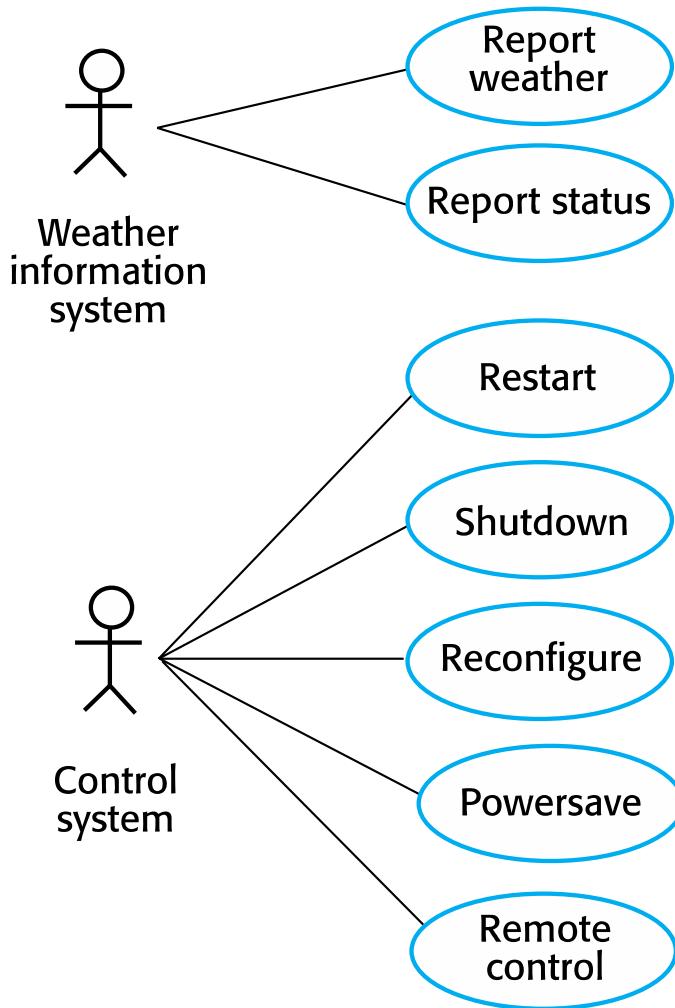
# System context for the weather station

---



# Weather station use cases

---



# Use case description—Report weather

System	Weather station
Use case	Report weather
Actors	Weather information system, Weather station
Description	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 is 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.

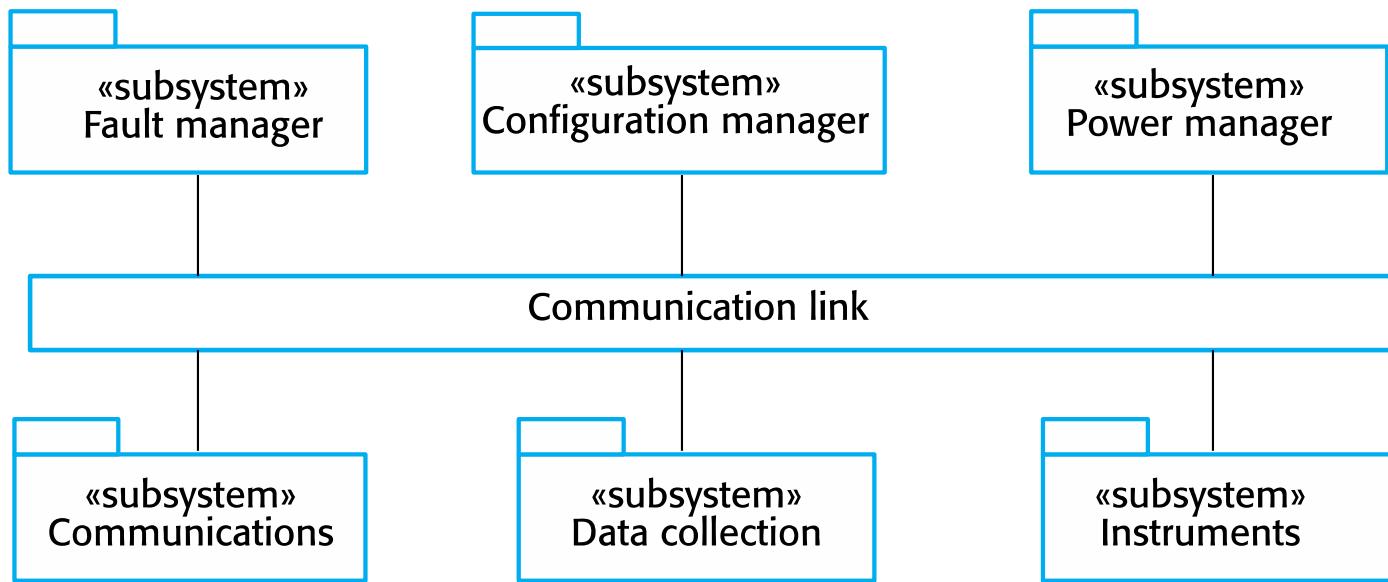
# Architectural design

---

- ✧ 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.

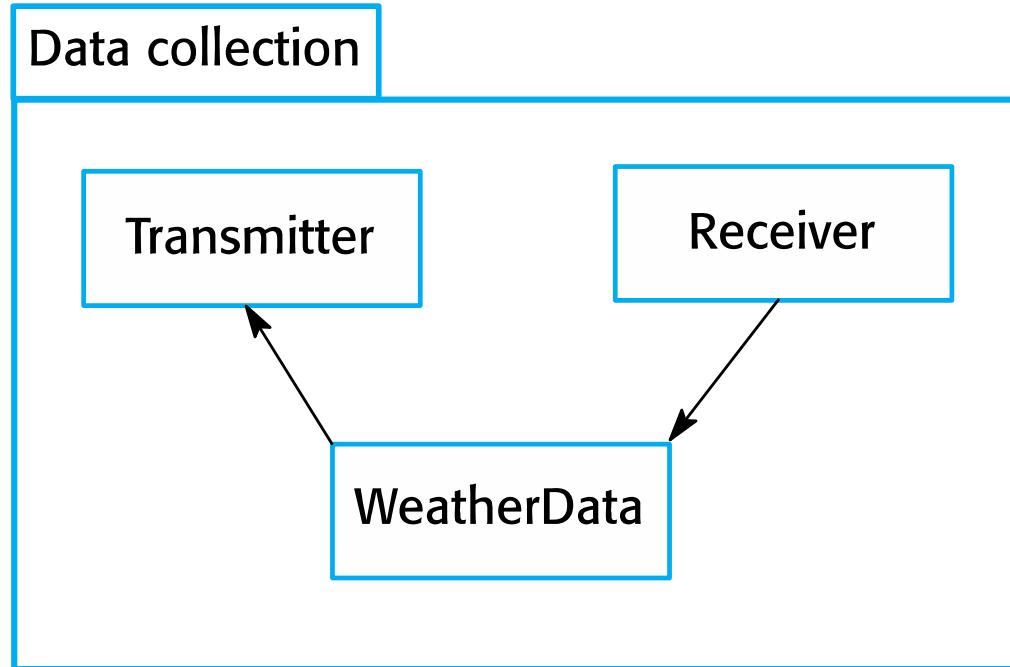
# High-level architecture of the weather station

---



# Architecture of data collection 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.
- ✧ 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 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.

# Weather station object classes

```
reportWeather ()  
reportStatus ()  
powerSave (instruments)  
remoteControl (commands)  
reconfigure (commands)  
restart (instruments)  
shutdown (instruments)
```

```
groundTemperatures  
windSpeeds  
windDirections  
pressures  
rainfall
```

```
collect ()  
summarize ()
```

## Ground thermometer

```
gt_Ident  
temperature
```

## Anemometer

```
an_Ident  
windSpeed  
windDirection
```

## Barometer

```
bar_Ident  
pressure  
height
```

# Design models

---

- ✧ Design models show the objects and object classes and relationships between these entities.
- ✧ There are two kinds of design model:
  - Structural 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.

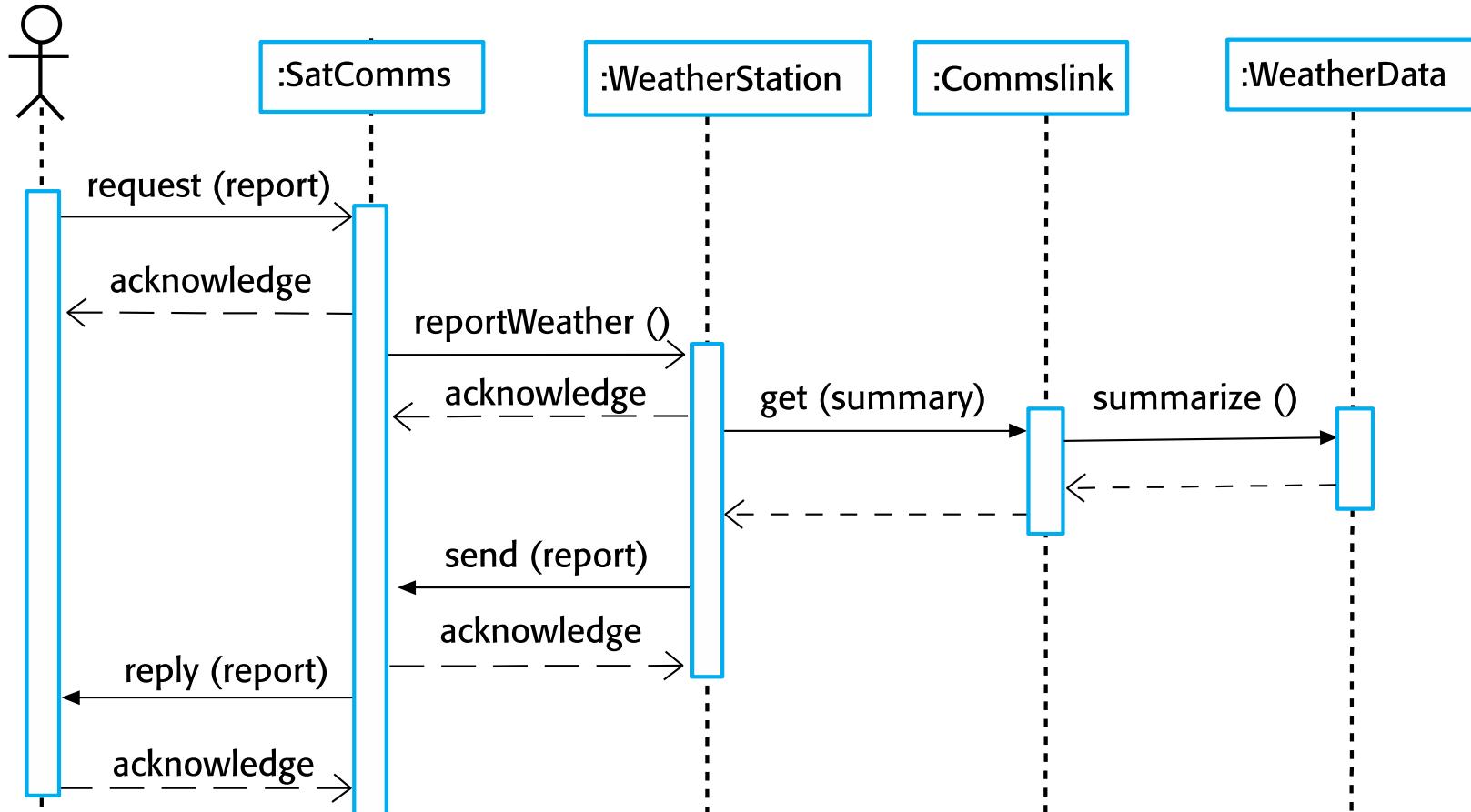
# Sequence models

---

- ✧ 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.

# Sequence diagram describing data collection

information system

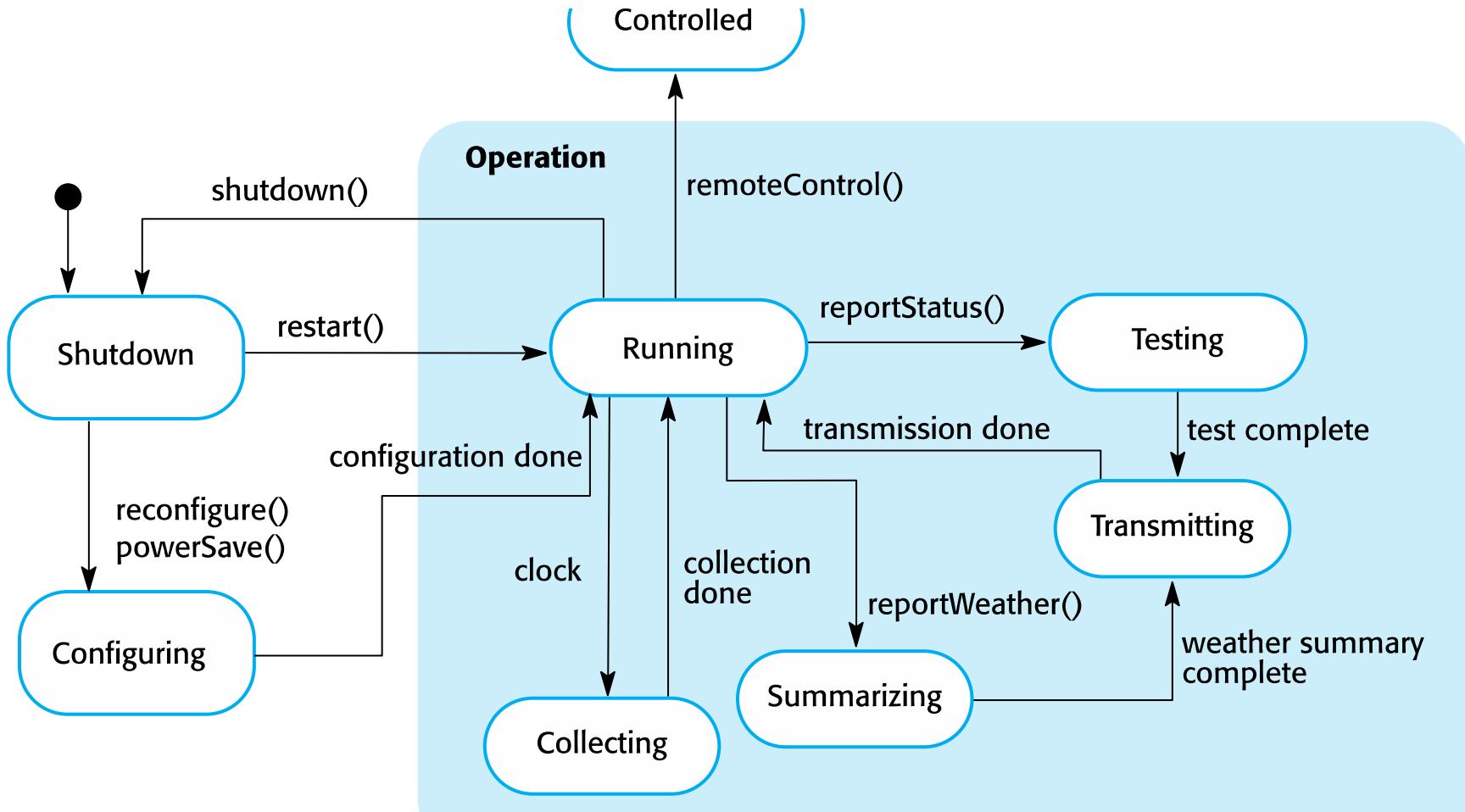


# State diagrams

---

- ✧ 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.

# Weather station state 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.

# Weather station interfaces

---

**«interface»**  
**Reporting**

weatherReport (WS-Ident): Wreport  
statusReport (WS-Ident): Sreport

**«interface»**  
**Remote Control**

startInstrument(instrument): iStatus  
stopInstrument (instrument): iStatus  
collectData (instrument): iStatus  
provideData (instrument ): string

---

# **Design patterns**

# 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.

# Patterns

---

- ✧ *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.*

# 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 are needed.
- ✧ Solution description
  - See slide with UML description.
- ✧ Consequences
  - Optimisations to enhance display performance are impractical.

# The Observer pattern (1)

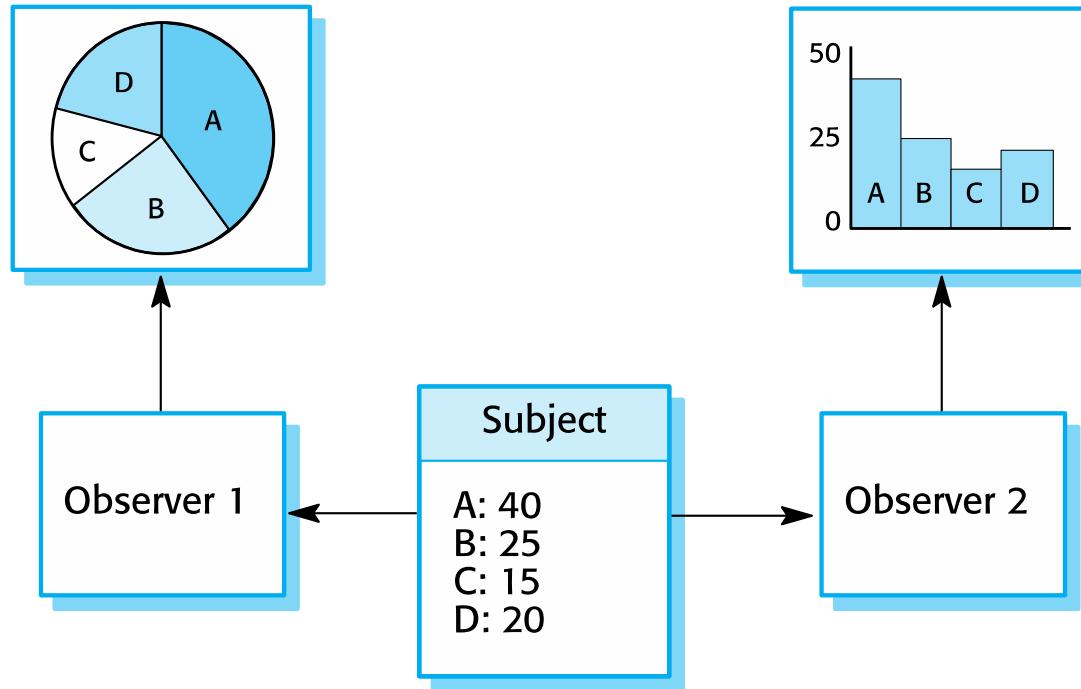
Pattern name	Observer
Description	<p>Separates the display of the state of an object from the object itself and allows alternative displays to be provided. When the object state changes, all displays are automatically notified and updated to reflect the change.</p>
Problem description	<p>In many situations, 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.</p> <p>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.</p>

# The Observer pattern (2)

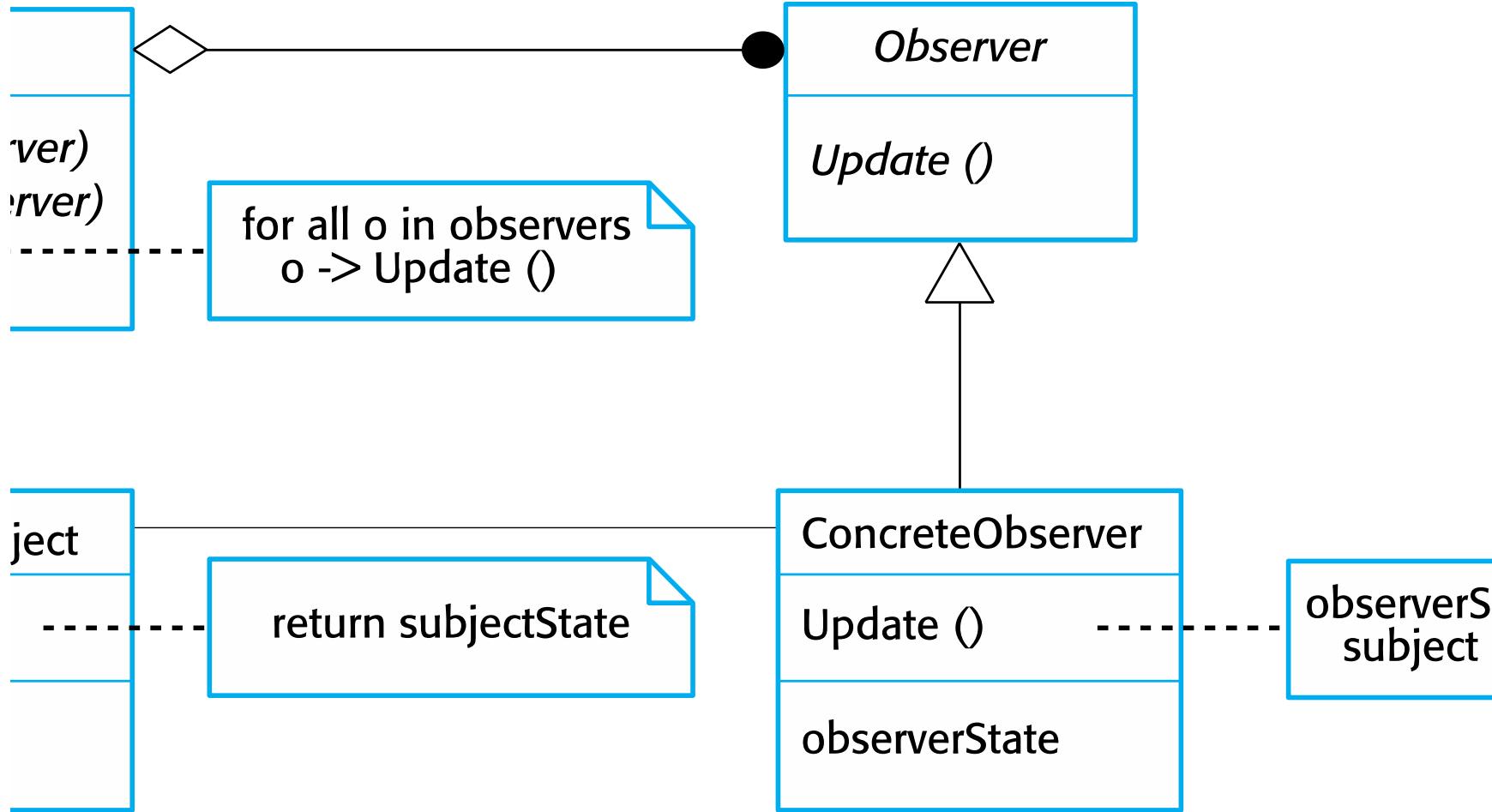
Pattern name	Observer
Solution description	<p>This involves two abstract objects, Subject and Observer, and two concrete objects, ConcreteSubject and ConcreteObject, which inherit the attributes of the related abstract objects. The abstract objects include general operations that are applicable in all 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.</p> <p>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.</p>
Consequences	<p>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.</p>

# Multiple displays using the Observer pattern

---



# 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

# 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

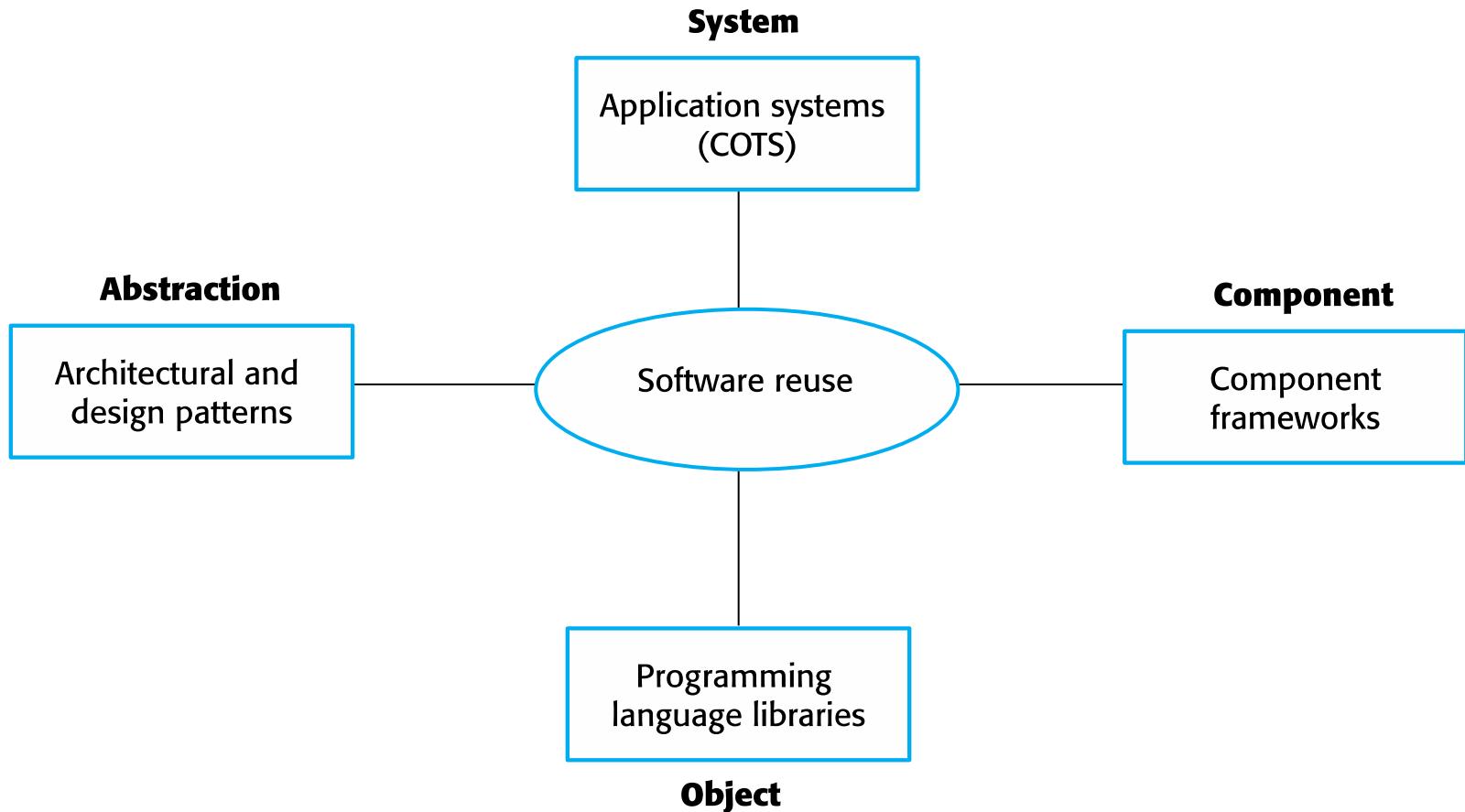
- Components are collections of objects and object classes that you reuse in application systems.

## ✧ The system level

- At this level, you reuse entire application systems.

# Software reuse

---



## 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

---

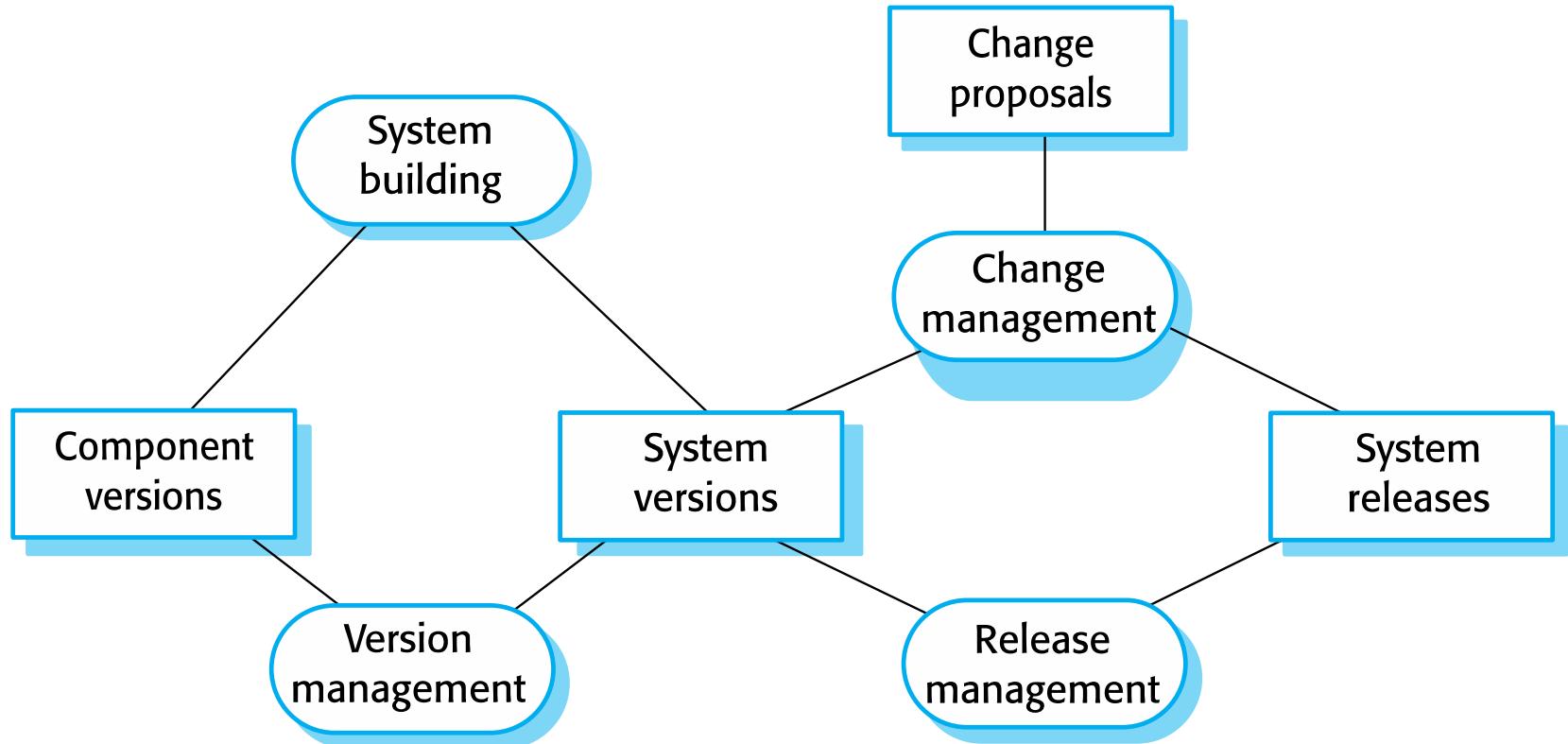
- ✧ Configuration management is the name given to the general process of managing a changing software system.
- ✧ The aim of configuration management is to support the system integration process so that all developers can access the project code and documents in a controlled way, find out what changes have been made, and compile and link components to create a system.
- ✧ See also Chapter 25.

# Configuration management activities

---

- ✧ Version management, where support is provided to keep track of the different versions of software components. Version management systems include facilities to coordinate development by several programmers.
- ✧ 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.

# Configuration management tool interaction

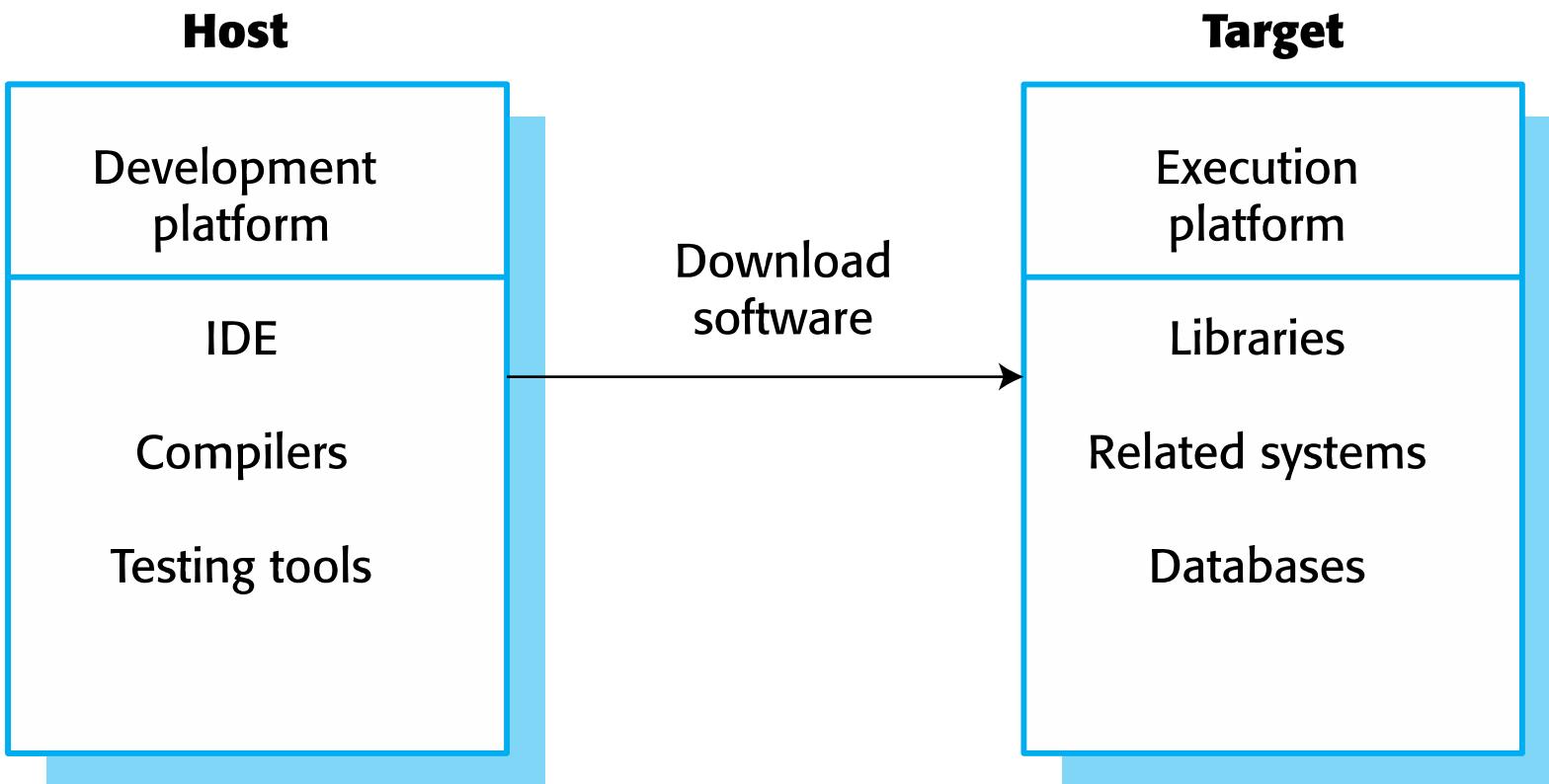


# Host-target development

---

- ✧ 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.

# Host-target development



# 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 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.
- ✧ 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

---

- ✧ 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](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.
- ✧ 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.

# Open source systems

---

- ✧ 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

---

- ✧ 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.
  - 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.

# Key points

---

- ✧ Software design and implementation are inter-leaved activities. The level of detail in the design depends on the type of system and whether you are using a plan-driven or agile approach.
- ✧ The process of object-oriented design includes activities to design the system architecture, identify objects in the system, describe the design using different object models and document the component interfaces.
- ✧ A range of different models may be produced during an object-oriented design process. These include static models (class models, generalization models, association models) and dynamic models (sequence models, state machine models).
- ✧ Component interfaces must be defined precisely so that other objects can use them. A UML interface stereotype may be used to define interfaces.

# Key points

---

- ✧ When developing software, you should always consider the possibility of reusing existing software, either as components, services or complete systems.
- ✧ Configuration management is the process of managing changes to an evolving software system. It is essential when a team of people are cooperating to develop software.
- ✧ Most software development is host-target development. You use an IDE on a host machine to develop the software, which is transferred to a target machine for execution.
- ✧ Open source development involves making the source code of a system publicly available. This means that many people can propose changes and improvements to the software.

# Chapter 1

---

## ■ The Nature of Software

*Slide Set to accompany*

*Software Engineering: A Practitioner's Approach, 8/e*  
by Roger S. Pressman and Bruce R. Maxim

Slides copyright © 1996, 2001, 2005, 2009, 2014 by Roger S. Pressman

***For non-profit educational use only***

May be reproduced ONLY for student use at the university level when used in conjunction with *Software Engineering: A Practitioner's Approach, 8/e*. Any other reproduction or use is prohibited without the express written permission of the author.

All copyright information MUST appear if these slides are posted on a website for student use.

# What is Software?

---

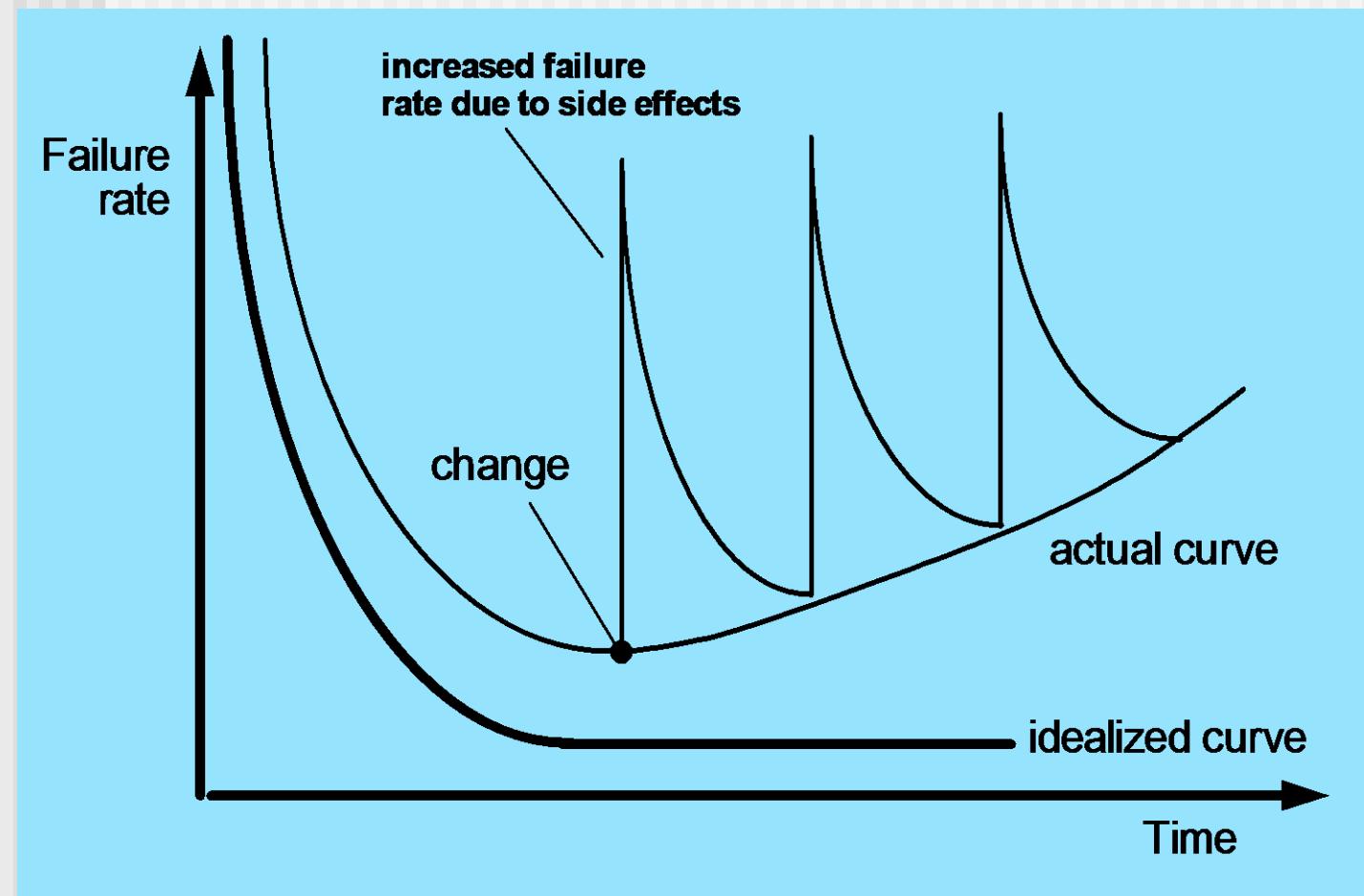
*Software is: (1) **instructions** (computer programs) that when executed provide desired features, function, and performance; (2) **data structures** that enable the programs to adequately manipulate information and (3) **documentation** that describes the operation and use of the programs.*

# What is Software?

---

- ***Software is developed or engineered, it is not manufactured in the classical sense.***
- ***Software doesn't "wear out."***
- ***Although the industry is moving toward component-based construction, most software continues to be custom-built.***

# Wear vs. Deterioration



# Software Applications

---

- System software
- Application software
- Engineering/Scientific software
- Embedded software
- Product-line software
- Web/Mobile applications)
- AI software (robotics, neural nets, game playing)

# Legacy Software

---

## *Why must it change?*

- software must be **adapted** to meet the needs of new computing environments or technology.
- software must be **enhanced** to implement new business requirements.
- software must be **extended to make it interoperable** with other more modern systems or databases.
- software must be **re-architected** to make it viable within a network environment.

# WebApps

---

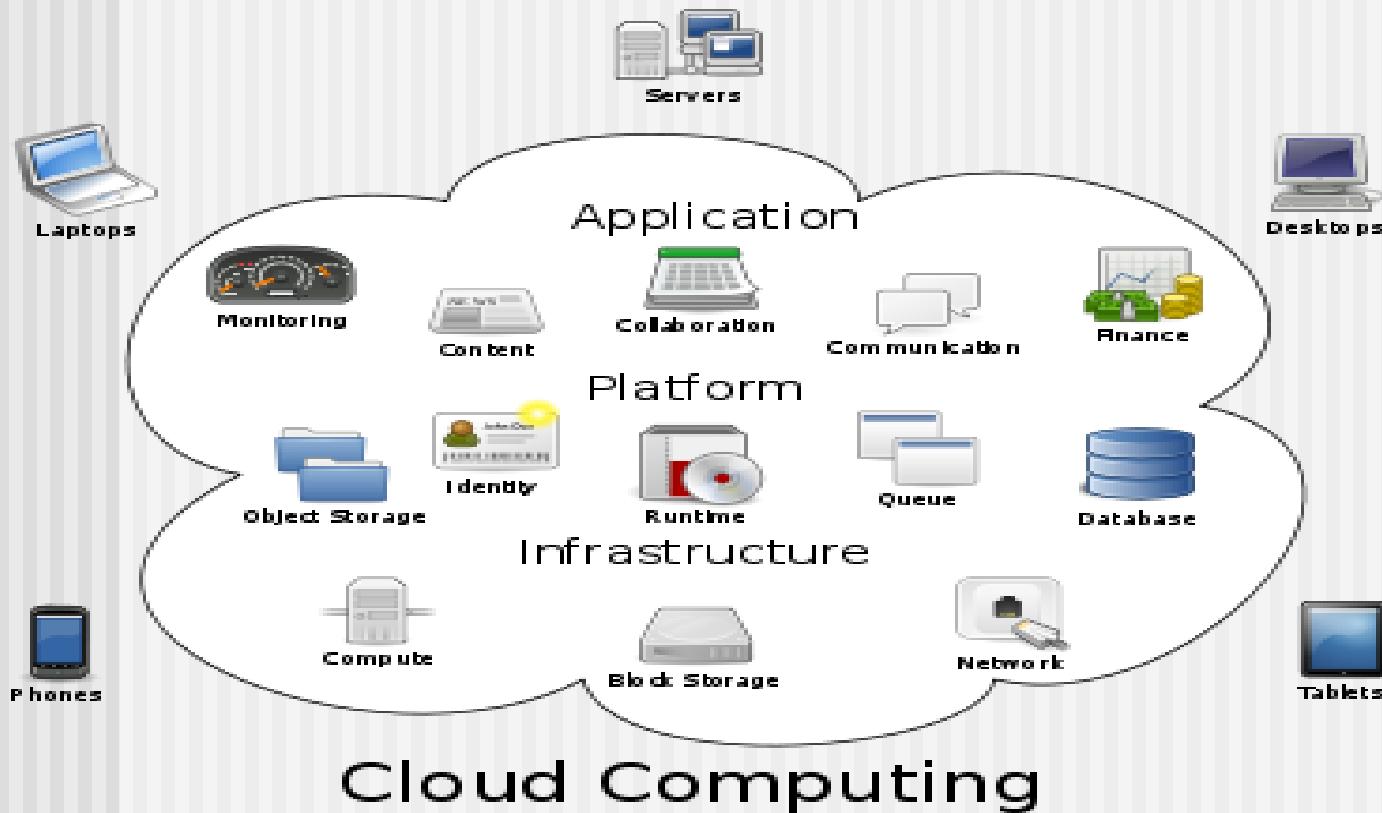
- Modern WebApps are much more than hypertext files with a few pictures
- WebApps are augmented with tools like XML and Java to allow Web engineers including interactive computing capability
- WebApps may standalone capability to end users or may be integrated with corporate databases and business applications
- Semantic web technologies (Web 3.0) have evolved into sophisticated corporate and consumer applications that encompass semantic databases that require web linking, flexible data representation, and application programmer interfaces (API's) for access
- The aesthetic nature of the content remains an important determinant of the quality of a WebApp.

# Mobile Apps

---

- Reside on mobile platforms such as cell phones or tablets
- Contain user interfaces that take both device characteristics and location attributes
- Often provide access to a combination of web-based resources and local device processing and storage capabilities
- Provide persistent storage capabilities within the platform
- A *mobile web application* allows a mobile device to access to web-based content using a browser designed to accommodate the strengths and weaknesses of the mobile platform
- A *mobile app* can gain direct access to the hardware found on the device to provide local processing and storage capabilities
- As time passes these differences will become blurred

# Cloud Computing



# Cloud Computing

---

- *Cloud computing* provides distributed data storage and processing resources to networked computing devices
- Computing resources reside outside the cloud and have access to a variety of resources inside the cloud
- Cloud computing requires developing an architecture containing both frontend and backend services
- Frontend services include the client devices and application software to allow access
- Backend services include servers, data storage, and server-resident applications
- Cloud architectures can be segmented to restrict access to private data

# Product Line Software

---

- *Product line software* is a set of software-intensive systems that share a common set of features and satisfy the needs of a particular market
- These software products are developed using the same application and data architectures using a common core of reusable software components
- A software product line shares a set of assets that include *requirements, architecture, design patterns, reusable components, test cases*, and other work products
- A software product line allows in the development of many products that are engineered by capitalizing on the commonality among all products within the product line

# Characteristics of WebApps - II

---

- **Data driven.** The primary function of many WebApps is to use hypermedia to present text, graphics, audio, and video content to the end-user.
- **Content sensitive.** The quality and aesthetic nature of content remains an important determinant of the quality of a WebApp.
- **Continuous evolution.** Unlike conventional application software that evolves over a series of planned, chronologically-spaced releases, Web applications evolve continuously.
- **Immediacy.** Although *immediacy*—the compelling need to get software to market quickly—is a characteristic of many application domains, WebApps often exhibit a time to market that can be a matter of a few days or weeks.
- **Security.** Because WebApps are available via network access, it is difficult, if not impossible, to limit the population of end-users who may access the application.
- **Aesthetics.** An undeniable part of the appeal of a WebApp is its look and feel.

# Chapter 1- Introduction

# Topics covered

---

- ✧ Professional software development
  - What is meant by software engineering.
- ✧ Software engineering ethics
  - A brief introduction to ethical issues that affect software engineering.
- ✧ Case studies
  - An introduction to three examples that are used in later chapters in the book.

# Software engineering

---

- ✧ The economies of ALL developed nations are dependent on software.
- ✧ More and more systems are software controlled
- ✧ Software engineering is concerned with theories, methods and tools for professional software development.
- ✧ Expenditure on software represents a significant fraction of GNP in all developed countries.

## Software costs

---

- ✧ Software costs often dominate computer system costs.  
The costs of software on a PC are often greater than the hardware cost.
- ✧ Software costs more to maintain than it does to develop.  
For systems with a long life, maintenance costs may be several times development costs.
- ✧ Software engineering is concerned with **cost-effective** software development.

# Software project failure

---

## ✧ *Increasing system complexity*

- As new software engineering techniques help us to build larger, more complex systems, the demands change. Systems have to be built and delivered more quickly; larger, even more complex systems are required; systems have to have new capabilities that were previously thought to be impossible.

## ✧ *Failure to use software engineering methods*

- It is fairly easy to write computer programs without using software engineering methods and techniques. Many companies have drifted into software development as their products and services have evolved. They do not use software engineering methods in their everyday work. Consequently, their software is often more expensive and less reliable than it should be.

# **Professional software development**

# Frequently asked questions about software engineering

---

Question	Answer
What is software?	Computer programs and associated documentation. Software products may be developed for a particular customer or may be developed for a general market.
What are the attributes of good software?	Good software should deliver the required functionality and performance to the user and should be maintainable, dependable and usable.
What is software engineering?	Software engineering is an engineering discipline that is concerned with all aspects of software production.
What are the fundamental software engineering activities?	Software specification, software development, software validation and software evolution.
What is the difference between software engineering and computer science?	Computer science focuses on theory and fundamentals; software engineering is concerned with the practicalities of developing and delivering useful software.
What is the difference between software engineering and system engineering?	System engineering is concerned with all aspects of computer-based systems development including hardware, software and process engineering. Software engineering is part of this more general process.

# Frequently asked questions about software engineering

---

Question	Answer
What are the key challenges facing software engineering?	Coping with increasing diversity, demands for reduced delivery times and developing trustworthy software.
What are the costs of software engineering?	Roughly 60% of software costs are development costs, 40% are testing costs. For custom software, evolution costs often exceed development costs.
What are the best software engineering techniques and methods?	While all software projects have to be professionally managed and developed, different techniques are appropriate for different types of system. For example, games should always be developed using a series of prototypes whereas safety critical control systems require a complete and analyzable specification to be developed. You can't, therefore, say that one method is better than another.
What differences has the web made to software engineering?	The web has led to the availability of software services and the possibility of developing highly distributed service-based systems. Web-based systems development has led to important advances in programming languages and software reuse.

# Software products

---

## ✧ Generic products

- Stand-alone systems that are marketed and sold to any customer who wishes to buy them.
- Examples – PC software such as graphics programs, project management tools; CAD software; software for specific markets such as appointments systems for dentists.

## ✧ Customized products

- Software that is commissioned by a specific customer to meet their own needs.
- Examples – embedded control systems, air traffic control software, traffic monitoring systems.

# Product specification

---

## ✧ Generic products

- The specification of what the software should do is owned by the software developer and decisions on software change are made by the developer.

## ✧ Customized products

- The specification of what the software should do is owned by the customer for the software and they make decisions on software changes that are required.

# Essential attributes of good software

---

Product characteristic	Description
Maintainability	Software should be written in such a way so that it can evolve to meet the changing needs of customers. This is a critical attribute because software change is an inevitable requirement of a changing business environment.
Dependability and security	Software dependability includes a range of characteristics including reliability, security and safety. Dependable software should not cause physical or economic damage in the event of system failure. Malicious users should not be able to access or damage the system.
Efficiency	Software should not make wasteful use of system resources such as memory and processor cycles. Efficiency therefore includes responsiveness, processing time, memory utilisation, etc.
Acceptability	Software must be acceptable to the type of users for which it is designed. This means that it must be understandable, usable and compatible with other systems that they use.

# Software engineering

---

- ✧ Software engineering is an engineering discipline that is concerned with all aspects of software production from the early stages of system specification through to maintaining the system after it has gone into use.
- ✧ Engineering discipline
  - Using appropriate theories and methods to solve problems bearing in mind organizational and financial constraints.
- ✧ All aspects of software production
  - Not just technical process of development. Also project management and the development of tools, methods etc. to support software production.

# Importance of software engineering

---

- ✧ More and more, individuals and society rely on advanced software systems. We need to be able to produce reliable and trustworthy systems economically and quickly.
- ✧ It is usually cheaper, in the long run, to use software engineering methods and techniques for software systems rather than just write the programs as if it was a personal programming project. For most types of system, the majority of costs are the costs of changing the software after it has gone into use.

# Software process activities

---

- ✧ Software specification, where customers and engineers define the software that is to be produced and the constraints on its operation.
- ✧ Software development, where the software is designed and programmed.
- ✧ Software validation, where the software is checked to ensure that it is what the customer requires.
- ✧ Software evolution, where the software is modified to reflect changing customer and market requirements.

# General issues that affect software

---

## ✧ Heterogeneity

- Increasingly, systems are required to operate as distributed systems across networks that include different types of computer and mobile devices.

## ✧ Business and social change

- Business and society are changing incredibly quickly as emerging economies develop and new technologies become available. They need to be able to change their existing software and to rapidly develop new software.

# General issues that affect software

---

## ✧ Security and trust

- As software is intertwined with all aspects of our lives, it is essential that we can trust that software.

## ✧ Scale

- Software has to be developed across a very wide range of scales, from very small embedded systems in portable or wearable devices through to Internet-scale, cloud-based systems that serve a global community.

# Software engineering diversity

---

- ✧ There are many different types of software system and there is no universal set of software techniques that is applicable to all of these.
- ✧ The software engineering methods and tools used depend on the type of application being developed, the requirements of the customer and the background of the development team.

# Application types

---

## ✧ Stand-alone applications

- These are application systems that run on a local computer, such as a PC. They include all necessary functionality and do not need to be connected to a network.

## ✧ Interactive transaction-based applications

- Applications that execute on a remote computer and are accessed by users from their own PCs or terminals. These include web applications such as e-commerce applications.

## ✧ Embedded control systems

- These are software control systems that control and manage hardware devices. Numerically, there are probably more embedded systems than any other type of system.

# Application types

---

## ✧ Batch processing systems

- These are business systems that are designed to process data in large batches. They process large numbers of individual inputs to create corresponding outputs.

## ✧ Entertainment systems

- These are systems that are primarily for personal use and which are intended to entertain the user.

## ✧ Systems for modeling and simulation

- These are systems that are developed by scientists and engineers to model physical processes or situations, which include many, separate, interacting objects.

# Application types

---

## ✧ Data collection systems

- These are systems that collect data from their environment using a set of sensors and send that data to other systems for processing.

## ✧ Systems of systems

- These are systems that are composed of a number of other software systems.

# Software engineering fundamentals

---

- ✧ Some fundamental principles apply to all types of software system, irrespective of the development techniques used:
  - Systems should be developed using a managed and understood development process. Of course, different processes are used for different types of software.
  - Dependability and performance are important for all types of system.
  - Understanding and managing the software specification and requirements (what the software should do) are important.
  - Where appropriate, you should reuse software that has already been developed rather than write new software.

# Internet software engineering

---

- ✧ The Web is now a platform for running application and organizations are increasingly developing web-based systems rather than local systems.
- ✧ Web services (discussed in Chapter 19) allow application functionality to be accessed over the web.
- ✧ Cloud computing is an approach to the provision of computer services where applications run remotely on the ‘cloud’.
  - Users do not buy software but pay according to use.

# Web-based software engineering

---

- ✧ Web-based systems are complex distributed systems but the fundamental principles of software engineering discussed previously are as applicable to them as they are to any other types of system.
- ✧ The fundamental ideas of software engineering apply to web-based software in the same way that they apply to other types of software system.

# Web software engineering

---

## ✧ Software reuse

- Software reuse is the dominant approach for constructing web-based systems. When building these systems, you think about how you can assemble them from pre-existing software components and systems.

## ✧ Incremental and agile development

- Web-based systems should be developed and delivered incrementally. It is now generally recognized that it is impractical to specify all the requirements for such systems in advance.

# Web software engineering

---

## ✧ Service-oriented systems

- Software may be implemented using service-oriented software engineering, where the software components are stand-alone web services.

## ✧ Rich interfaces

- Interface development technologies such as AJAX and HTML5 have emerged that support the creation of rich interfaces within a web browser.

---

# **Software engineering ethics**

# Software engineering ethics

---

- ✧ Software engineering involves wider responsibilities than simply the application of technical skills.
- ✧ Software engineers must behave in an honest and ethically responsible way if they are to be respected as professionals.
- ✧ Ethical behaviour is more than simply upholding the law but involves following a set of principles that are morally correct.

# Issues of professional responsibility

---

## ✧ Confidentiality

- Engineers should normally respect the confidentiality of their employers or clients irrespective of whether or not a formal confidentiality agreement has been signed.

## ✧ Competence

- Engineers should not misrepresent their level of competence. They should not knowingly accept work which is outwith their competence.

# Issues of professional responsibility

---

## ✧ Intellectual property rights

- Engineers should be aware of local laws governing the use of intellectual property such as patents, copyright, etc. They should be careful to ensure that the intellectual property of employers and clients is protected.

## ✧ Computer misuse

- Software engineers should not use their technical skills to misuse other people's computers. Computer misuse ranges from relatively trivial (game playing on an employer's machine, say) to extremely serious (dissemination of viruses).

# ACM/IEEE Code of Ethics

---

- ✧ The professional societies in the US have cooperated to produce a code of ethical practice.
- ✧ Members of these organisations sign up to the code of practice when they join.
- ✧ The Code contains eight Principles related to the behaviour of and decisions made by professional software engineers, including practitioners, educators, managers, supervisors and policy makers, as well as trainees and students of the profession.

# Rationale for the code of ethics

---

- *Computers have a central and growing role in commerce, industry, government, medicine, education, entertainment and society at large. Software engineers are those who contribute by direct participation or by teaching, to the analysis, specification, design, development, certification, maintenance and testing of software systems.*
- *Because of their roles in developing software systems, software engineers have significant opportunities to do good or cause harm, to enable others to do good or cause harm, or to influence others to do good or cause harm. To ensure, as much as possible, that their efforts will be used for good, software engineers must commit themselves to making software engineering a beneficial and respected profession.*

# The ACM/IEEE Code of Ethics

---

## Software Engineering Code of Ethics and Professional Practice

ACM/IEEE-CS Joint Task Force on Software Engineering Ethics and Professional Practices

### PREAMBLE

The short version of the code summarizes aspirations at a high level of the abstraction; the clauses that are included in the full version give examples and details of how these aspirations change the way we act as software engineering professionals. Without the aspirations, the details can become legalistic and tedious; without the details, the aspirations can become high sounding but empty; together, the aspirations and the details form a cohesive code.

Software engineers shall commit themselves to making the analysis, specification, design, development, testing and maintenance of software a beneficial and respected profession. In accordance with their commitment to the health, safety and welfare of the public, software engineers shall adhere to the following Eight Principles:

# Ethical principles

---

1. PUBLIC - Software engineers shall act consistently with the public interest.
2. CLIENT AND EMPLOYER - Software engineers shall act in a manner that is in the best interests of their client and employer consistent with the public interest.
3. PRODUCT - Software engineers shall ensure that their products and related modifications meet the highest professional standards possible.
4. JUDGMENT - Software engineers shall maintain integrity and independence in their professional judgment.
5. MANAGEMENT - Software engineering managers and leaders shall subscribe to and promote an ethical approach to the management of software development and maintenance.
6. PROFESSION - Software engineers shall advance the integrity and reputation of the profession consistent with the public interest.
7. COLLEAGUES - Software engineers shall be fair to and supportive of their colleagues.
8. SELF - Software engineers shall participate in lifelong learning regarding the practice of their profession and shall promote an ethical approach to the practice of the profession.

# Case studies

# Ethical dilemmas

---

- ✧ Disagreement in principle with the policies of senior management.
- ✧ Your employer acts in an unethical way and releases a safety-critical system without finishing the testing of the system.
- ✧ Participation in the development of military weapons systems or nuclear systems.

# Case studies

---

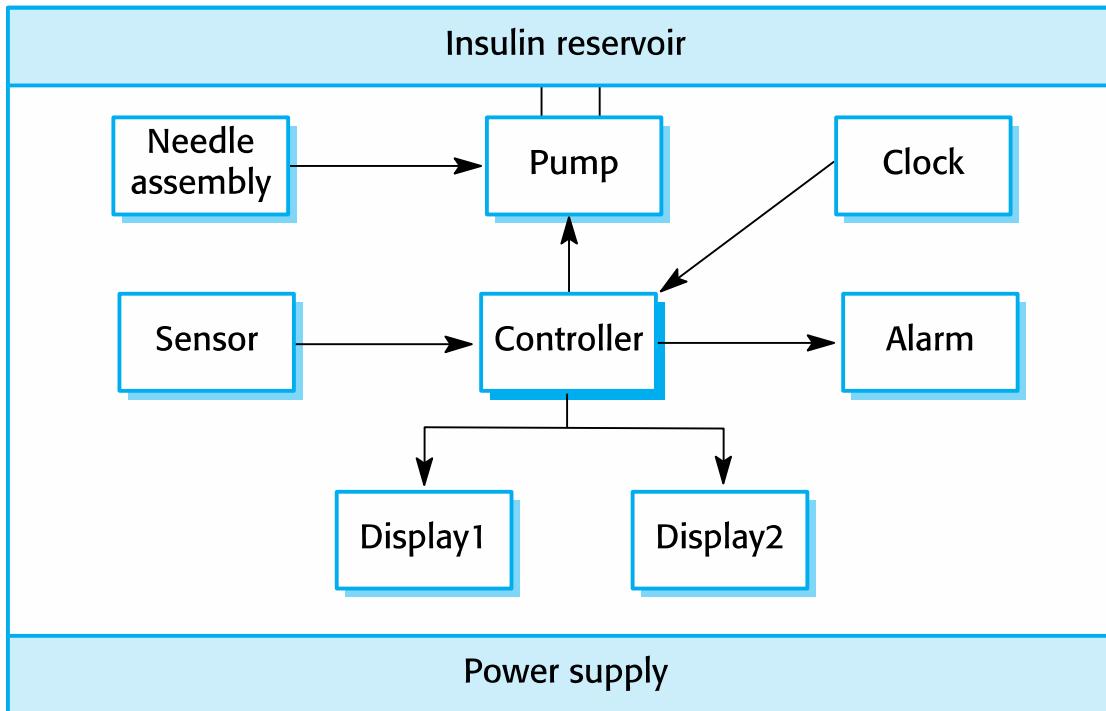
- ✧ A personal insulin pump
  - An embedded system in an insulin pump used by diabetics to maintain blood glucose control.
- ✧ A mental health case patient management system
  - Mentcare. A system used to maintain records of people receiving care for mental health problems.
- ✧ A wilderness weather station
  - A data collection system that collects data about weather conditions in remote areas.
- ✧ iLearn: a digital learning environment
  - A system to support learning in schools

# Insulin pump control system

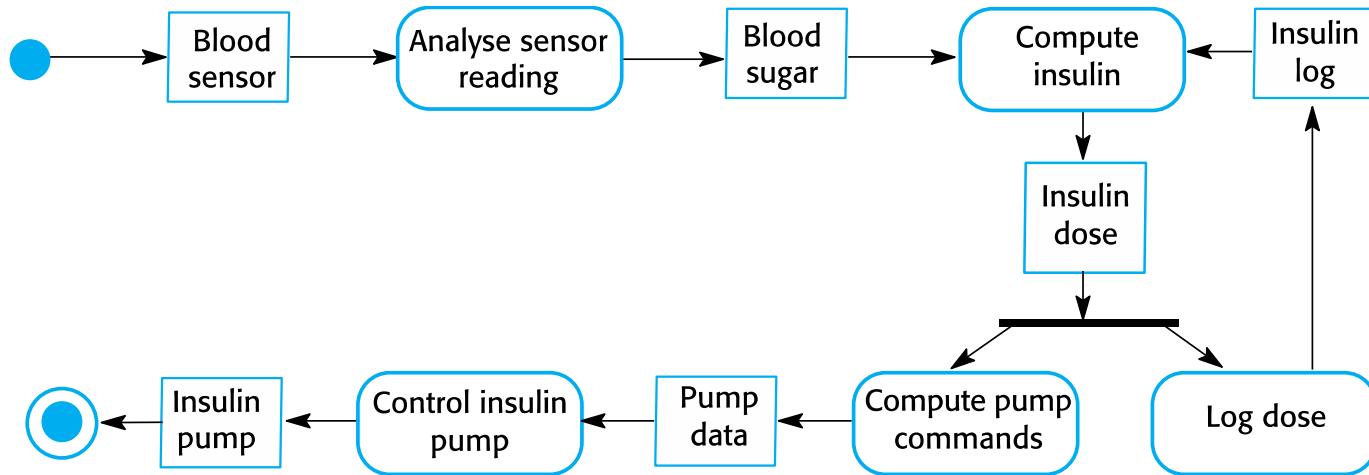
---

- ✧ Collects data from a blood sugar sensor and calculates the amount of insulin required to be injected.
- ✧ Calculation based on the rate of change of blood sugar levels.
- ✧ Sends signals to a micro-pump to deliver the correct dose of insulin.
- ✧ Safety-critical system as low blood sugars can lead to brain malfunctioning, coma and death; high-blood sugar levels have long-term consequences such as eye and kidney damage.

# Insulin pump hardware architecture



# Activity model of the insulin pump



# Essential high-level requirements

---

- ✧ The system shall be available to deliver insulin when required.
- ✧ The system shall perform reliably and deliver the correct amount of insulin to counteract the current level of blood sugar.
- ✧ The system must therefore be designed and implemented to ensure that the system always meets these requirements.

# Mentcare: A patient information system for mental health care

---

- ✧ A patient information system to support mental health care is a medical information system that maintains information about patients suffering from mental health problems and the treatments that they have received.
- ✧ Most mental health patients do not require dedicated hospital treatment but need to attend specialist clinics regularly where they can meet a doctor who has detailed knowledge of their problems.
- ✧ To make it easier for patients to attend, these clinics are not just run in hospitals. They may also be held in local medical practices or community centres.

# Mentcare

---

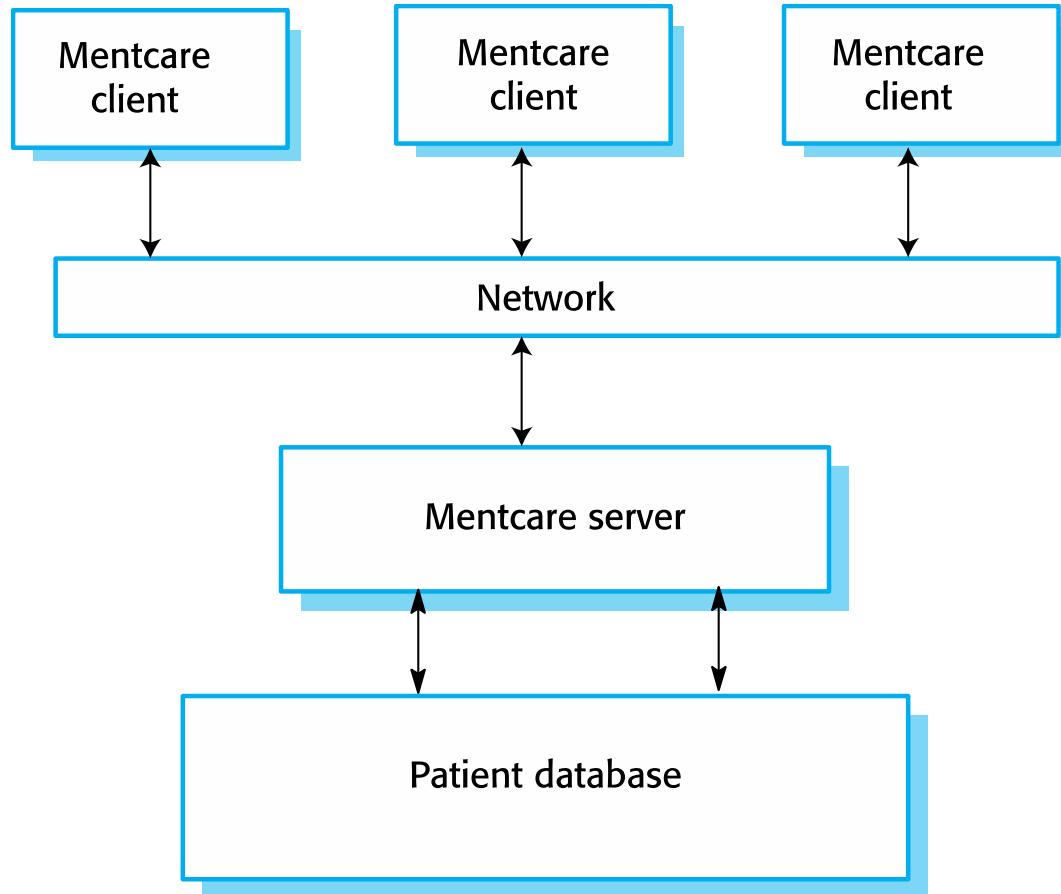
- ✧ Mentcare is an information system that is intended for use in clinics.
- ✧ It makes use of a centralized database of patient information but has also been designed to run on a PC, so that it may be accessed and used from sites that do not have secure network connectivity.
- ✧ When the local systems have secure network access, they use patient information in the database but they can download and use local copies of patient records when they are disconnected.

## Mentcare goals

---

- ✧ To generate management information that allows health service managers to assess performance against local and government targets.
- ✧ To provide medical staff with timely information to support the treatment of patients.

# The organization of the Mentcare system



# Key features of the Mentcare system

---

## ✧ Individual care management

- Clinicians can create records for patients, edit the information in the system, view patient history, etc. The system supports data summaries so that doctors can quickly learn about the key problems and treatments that have been prescribed.

## ✧ Patient monitoring

- The system monitors the records of patients that are involved in treatment and issues warnings if possible problems are detected.

## ✧ Administrative reporting

- The system generates monthly management reports showing the number of patients treated at each clinic, the number of patients who have entered and left the care system, number of patients sectioned, the drugs prescribed and their costs, etc.

# Mentcare system concerns

---

## ✧ Privacy

- It is essential that patient information is confidential and is never disclosed to anyone apart from authorised medical staff and the patient themselves.

## ✧ Safety

- Some mental illnesses cause patients to become suicidal or a danger to other people. Wherever possible, the system should warn medical staff about potentially suicidal or dangerous patients.
- The system must be available when needed otherwise safety may be compromised and it may be impossible to prescribe the correct medication to patients.

# Wilderness weather station

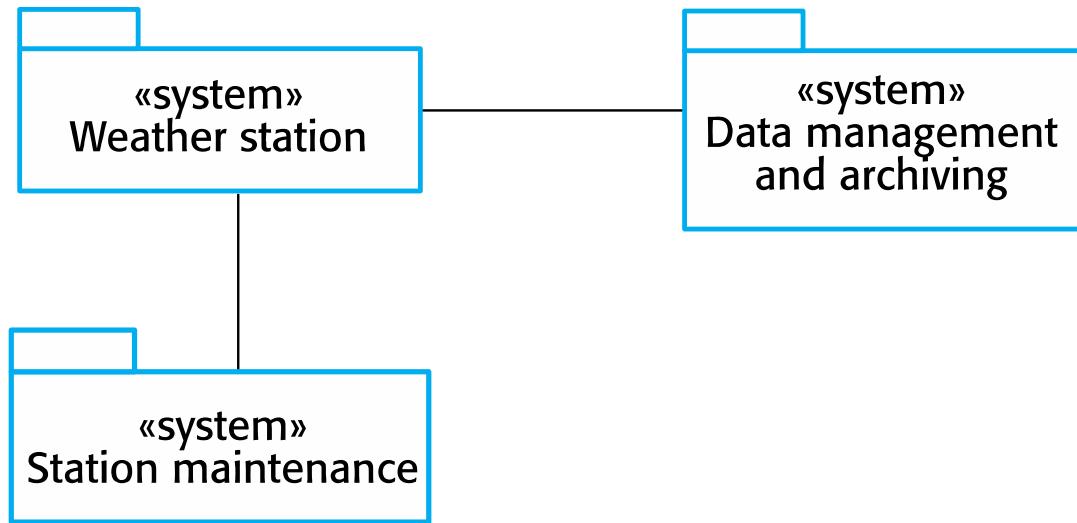
---

- ✧ The government of a country with large areas of wilderness decides to deploy several hundred weather stations in remote areas.
- ✧ Weather stations collect data from a set of instruments that measure temperature and pressure, sunshine, rainfall, wind speed and wind direction.
  - The weather station includes a number of instruments that measure weather parameters such as the wind speed and direction, the ground and air temperatures, the barometric pressure and the rainfall over a 24-hour period. Each of these instruments is controlled by a software system that takes parameter readings periodically and manages the data collected from the instruments.



# The weather station's environment

---



# Weather information system

---

- ✧ The weather station system
  - This is responsible for collecting weather data, carrying out some initial data processing and transmitting it to the data management system.
- ✧ The data management and archiving system
  - This system collects the data from all of the wilderness weather stations, carries out data processing and analysis and archives the data.
- ✧ The station maintenance system
  - This system can communicate by satellite with all wilderness weather stations to monitor the health of these systems and provide reports of problems.

## Additional software functionality

---

- ✧ Monitor the instruments, power and communication hardware and report faults to the management system.
- ✧ Manage the system power, ensuring that batteries are charged whenever the environmental conditions permit but also that generators are shut down in potentially damaging weather conditions, such as high wind.
- ✧ Support dynamic reconfiguration where parts of the software are replaced with new versions and where backup instruments are switched into the system in the event of system failure.

# iLearn: A digital learning environment

---

- ✧ A digital learning environment is a framework in which a set of general-purpose and specially designed tools for learning may be embedded plus a set of applications that are geared to the needs of the learners using the system.
- ✧ The tools included in each version of the environment are chosen by teachers and learners to suit their specific needs.
  - These can be general applications such as spreadsheets, learning management applications such as a Virtual Learning Environment (VLE) to manage homework submission and assessment, games and simulations.

# Service-oriented systems

---

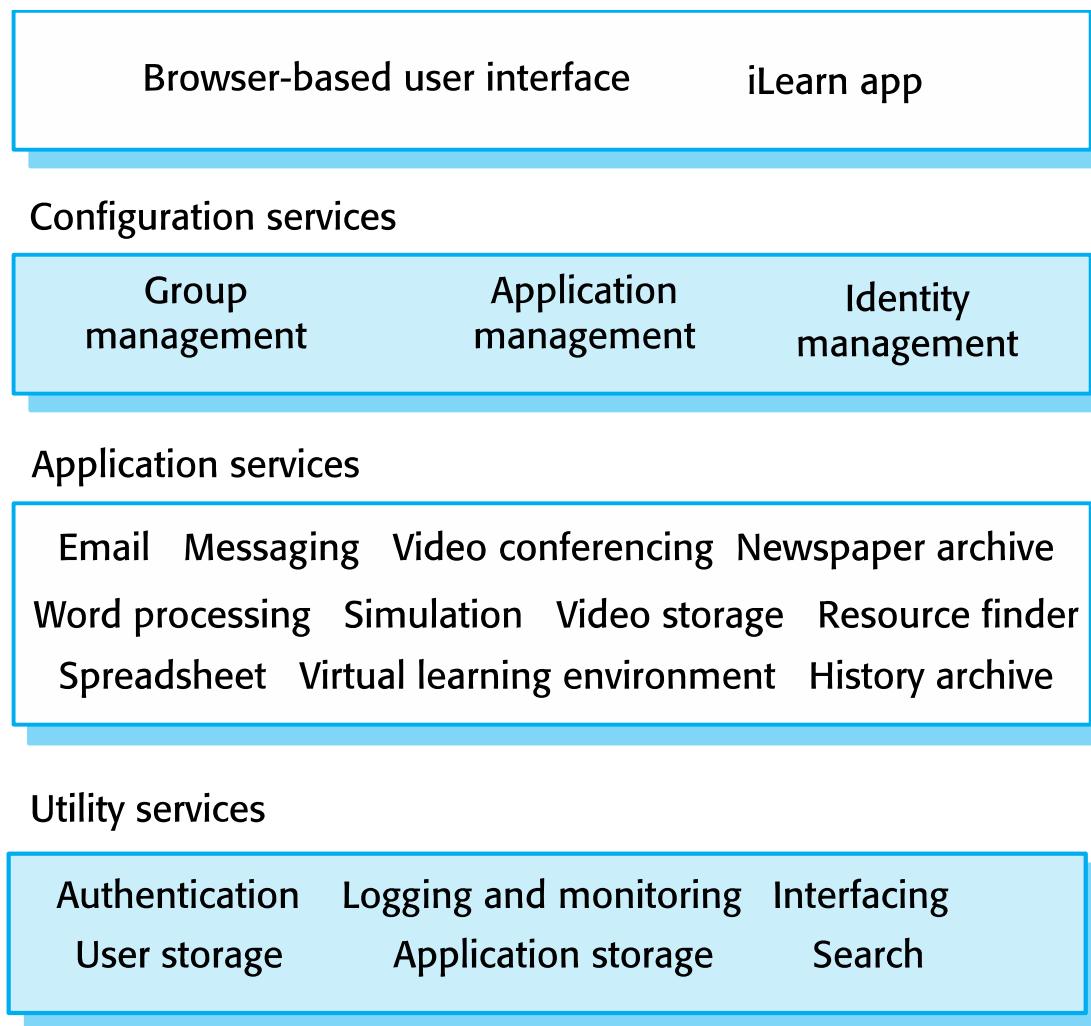
- ✧ The system is a service-oriented system with all system components considered to be a replaceable service.
- ✧ This allows the system to be updated incrementally as new services become available.
- ✧ It also makes it possible to rapidly configure the system to create versions of the environment for different groups such as very young children who cannot read, senior students, etc.

# iLearn services

---

- ✧ *Utility services* that provide basic application-independent functionality and which may be used by other services in the system.
- ✧ *Application services* that provide specific applications such as email, conferencing, photo sharing etc. and access to specific educational content such as scientific films or historical resources.
- ✧ *Configuration services* that are used to adapt the environment with a specific set of application services and do define how services are shared between students, teachers and their parents.

# iLearn architecture



# iLearn service integration

---

- ✧ *Integrated services* are services which offer an API (application programming interface) and which can be accessed by other services through that API. Direct service-to-service communication is therefore possible.
- ✧ *Independent services* are services which are simply accessed through a browser interface and which operate independently of other services. Information can only be shared with other services through explicit user actions such as copy and paste; re-authentication may be required for each independent service.

## Key points

---

- ✧ Software engineering is an engineering discipline that is concerned with all aspects of software production.
- ✧ Essential software product attributes are maintainability, dependability and security, efficiency and acceptability.
- ✧ The high-level activities of specification, development, validation and evolution are part of all software processes.
- ✧ The fundamental notions of software engineering are universally applicable to all types of system development.

# Key points

---

- ✧ There are many different types of system and each requires appropriate software engineering tools and techniques for their development.
- ✧ The fundamental ideas of software engineering are applicable to all types of software system.
- ✧ Software engineers have responsibilities to the engineering profession and society. They should not simply be concerned with technical issues.
- ✧ Professional societies publish codes of conduct which set out the standards of behaviour expected of their members.

# **UML**

## **The Unified Modeling Language**

# Introduction

- **Modeling:** drawing a flowchart listing the steps carried out by an application.
- **Why do we use modeling?**

Defining a model makes it easier to break up a complex application or a huge system into simple, discrete pieces that can be individually studied. We can focus more easily on the smaller parts of a system and then understand the "big picture."
- **The reasons behind modeling can be summed up in two words:**
  - Readability
  - Reusability

- **Readability:** brings clarity—ease of understanding. Understanding a system is the first step in either building or enhancing a system. This involves knowing what a system is made up of, how it behaves, and so forth. Depicting a system to make it readable involves capturing the structure of a system and the behavior of the system.
- **Reusability:** is the byproduct of making a system readable. After a system has been modeled to make it easy to understand, we tend to identify similarities or redundancy, be they in terms of functionality, features, or structure. UML provides the ability to capture the characteristics of a system by using notations. UML provides a wide array of simple notations for documenting systems based on the object-oriented design principles. These notations are called the nine diagrams of UML.

# What is UML?

The Unified Modeling Language (UML) is a standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling and other non-software systems. The UML is a very important part of developing object oriented software and the software development process. The UML uses graphical notations to express the design of software projects. Using the UML helps project teams communicate, explore potential designs, and validate the architectural design of the software.

# Goals of UML

**The primary goals in the design of the UML were:**

- Provide users with a ready-to-use, expressive visual modeling language so they can develop and exchange meaningful models.
- Provide extensibility and specialization mechanisms to extend the core concepts.
- Be independent of particular programming languages and development processes.
- Provide a formal basis for understanding the modeling language.
- Encourage the growth of the OO tools market.

# Why use UML

The industry looks for techniques to automate the production of software and to improve quality and reduce cost and time-to-market. These techniques include component technology, visual programming, and frameworks. Businesses also seek techniques to manage the complexity of systems as they increase in scope and scale. They recognize the need to solve the architectural problems, such as physical distribution, concurrency, security, load balancing and fault tolerance. Additionally, the development for the World Wide Web.

→ The Unified Modeling Language (UML) was designed to respond to these needs.

# UML Diagrams

UML is made up of nine diagrams that can be used to model a system at different points of time in the software life cycle of a system. The nine UML diagrams are:

- **Use case diagram**
- **Class diagram**
- **Object diagram**
- **State diagram**
- **Activity diagram**
- **Sequence diagram**
- **Collaboration diagram**
- **Component diagram**
- **Deployment diagram**

# UML Diagram Classification

- A software system can be said to have three distinct characteristics: *static*, *dynamic*, and *implementation*.
- **Static:** the structural aspect of the system, define what parts the system is made up of.
- **Dynamic:** The behavioral features of a system; for example, the ways a system behaves in response to certain events or actions are the dynamic characteristics of a system.
- **Implementation:** The implementation characteristic of a system is an entirely new feature that describes the different elements required for deploying a system.

The UML diagrams that fall under each of these categories are:

- **Static**

- Use case diagram
- Class diagram

- **Dynamic**

- Object diagram
- State diagram
- Activity diagram
- Sequence diagram
- Collaboration diagram

- **Implementation**

- Component diagram
- Deployment diagram

# Use Case Diagram

The Use case diagram is used to identify the primary elements and processes that form the system. The primary elements are termed as "actors" and the processes are called "use cases." The Use case diagram shows which actors interact with each use case.

- UML Use Case Diagrams (**UCDs**) can be used to describe the functionality of a system, they capture the functional aspects and business process in the system.
- UCDs have only 4 major elements: The **actors** that the system you are describing interacts with, the **system** itself, the **use cases**, or services, that the system knows how to perform, and the lines that represent **relationships** between these elements.

- **Actors:** An actor portrays any entity (or entities) that performs certain roles in a given system. The most obvious candidates for actors are the humans in the system. If your system interacts with other systems (databases, servers maintained by other people, legacy systems) you will be best to treat these as actors.



- **Use case:** A use case in a use case diagram is a visual representation of a distinct business functionality in a system.

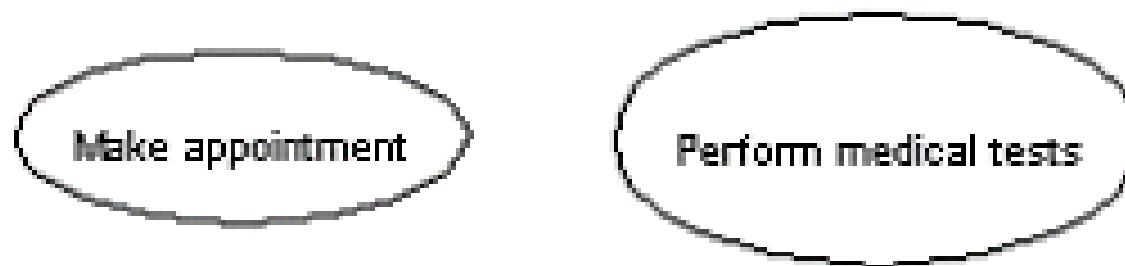
- **As-is scenarios** describe a current situation. During reengineering, for example, the current system is understood by observing users and describing their actions as scenarios. These scenarios can then be validated for correctness and accuracy with the users.
- **Visionary scenarios** describe a future system. Visionary scenarios are used both as a point in the modeling space by developers as they refine their ideas of the future system and as a communication medium to elicit requirements from users. Visionary scenarios can be viewed as an inexpensive prototype.
- **Evaluation scenarios** describe user tasks against which the system is to be evaluated. The collaborative development of evaluation scenarios by users and developers also improves the definition of the functionality tested by these scenarios.
- **Training scenarios** are tutorials used for introducing new users to the system. These are step-by-step instructions designed to hand-hold the user through common tasks.

### **Questions for identifying scenarios**

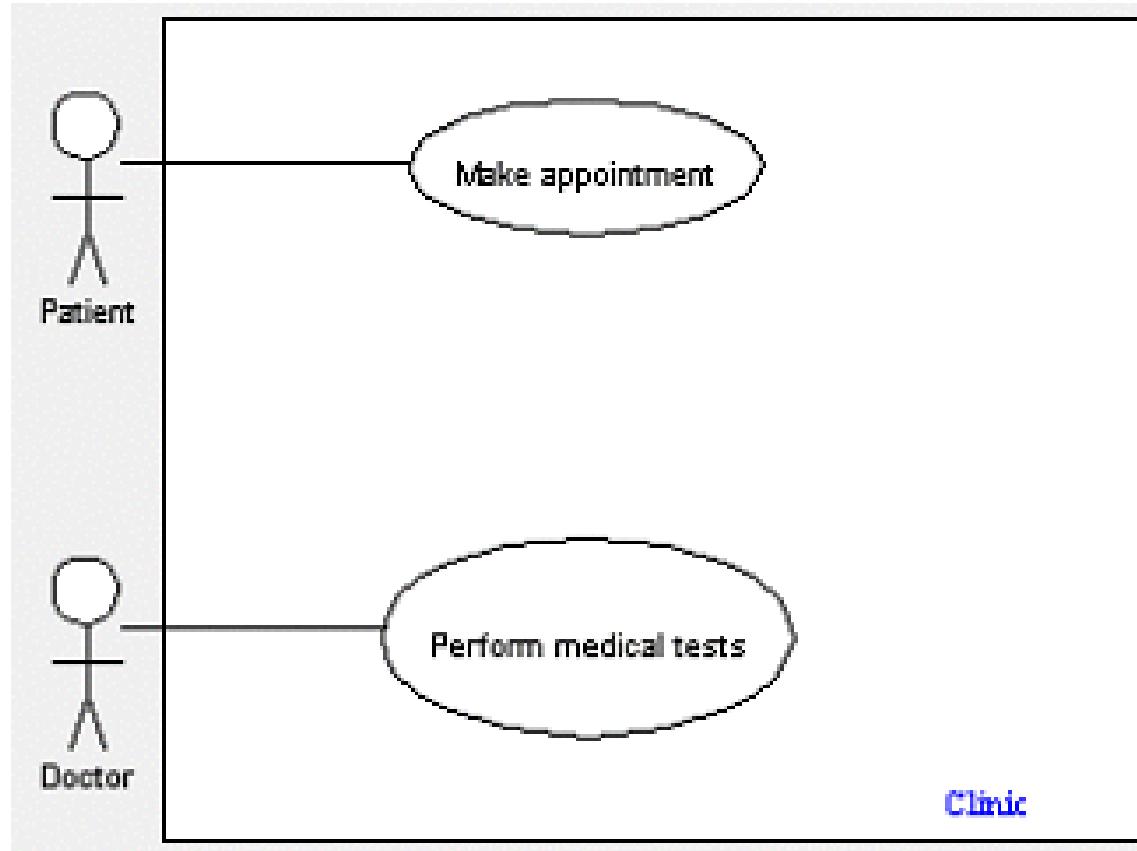
- What are the tasks that the actor wants the system to perform?
- What information does the actor access? Who creates that data? Can it be modified or removed? By whom?
- Which external changes does the actor need to inform the system about? How often? When?
- Which events does the system need to inform the actor about? With what latency?

As the first step in identifying use cases, you should list the discrete business functions in your problem statement. Each of these business functions can be classified as a potential use case.

→ A use case is an external view of the system that represents some action the user might perform in order to complete a task.



- **System boundary:** A system boundary defines the scope of what a system will be. A system boundary of a use case diagram defines the limits of the system.

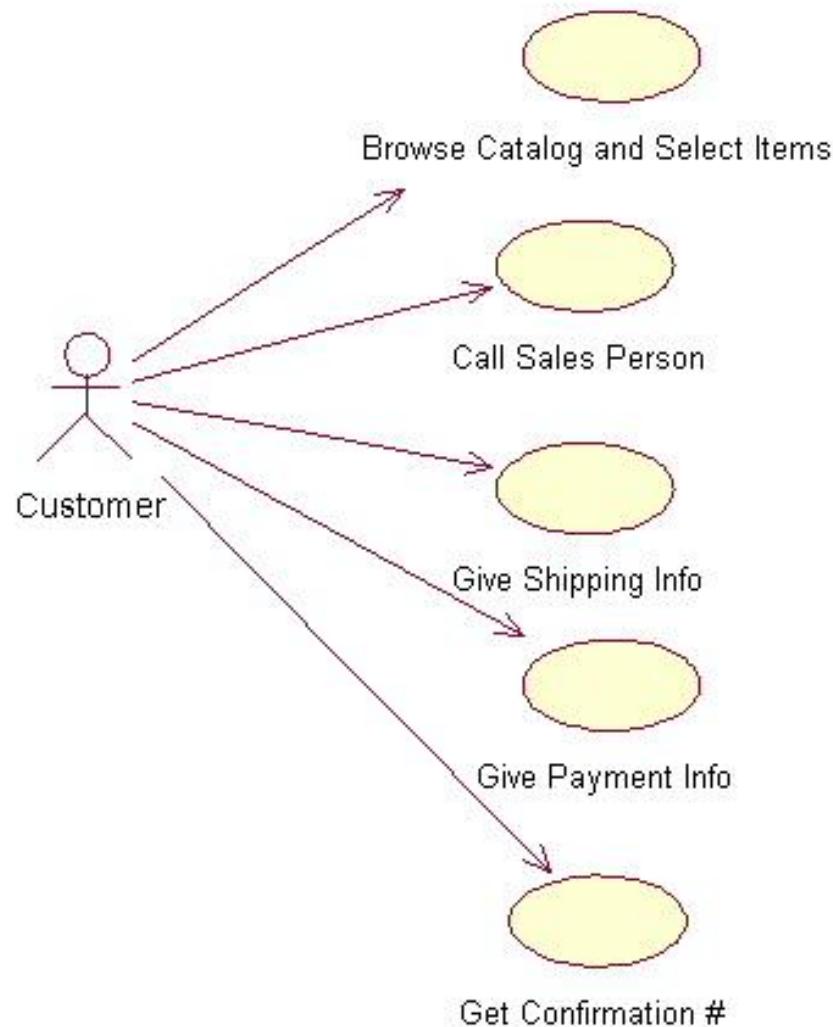


## Example 1

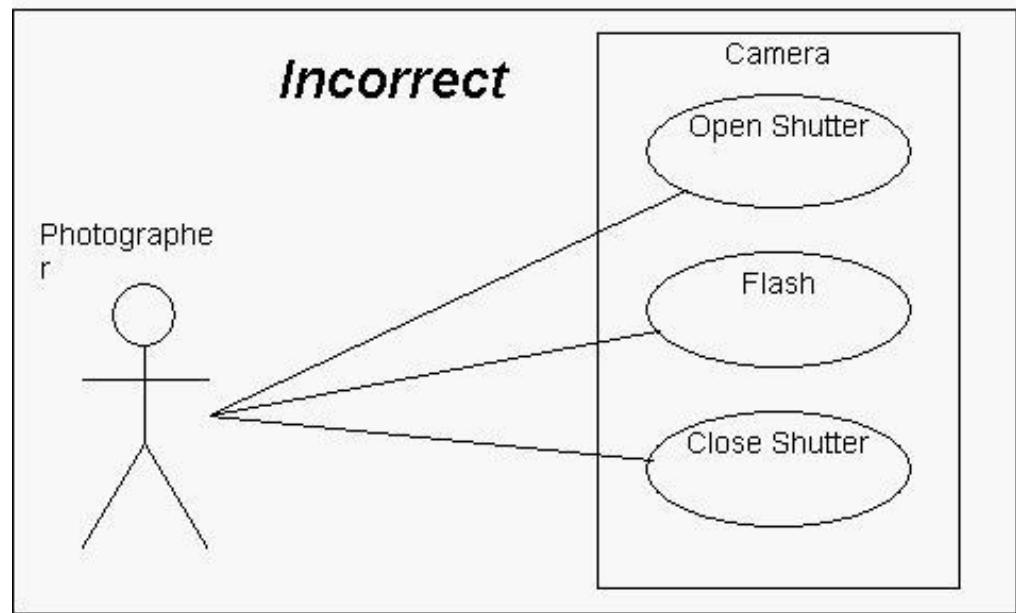
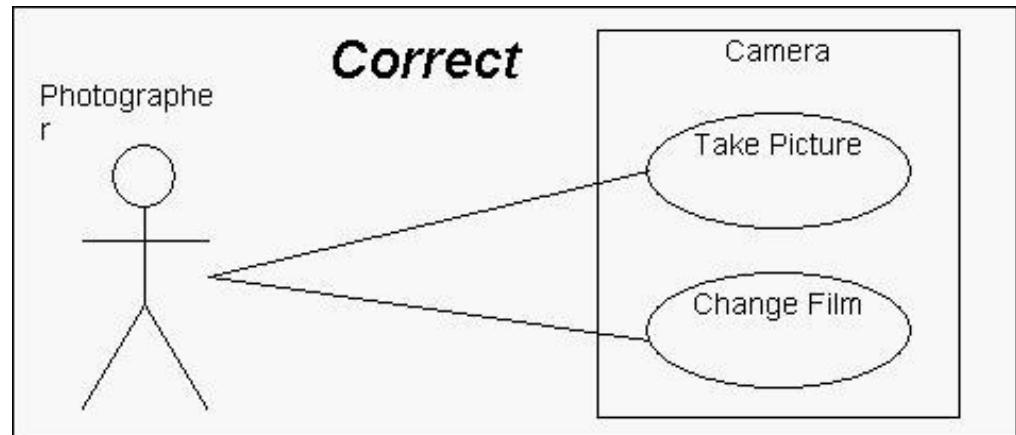
a user placing an order with a sales company might follow these steps.

- Browse catalog and select items.
- Call sales representative.
- Supply shipping information.
- Supply payment information.
- Receive conformation number from salesperson.

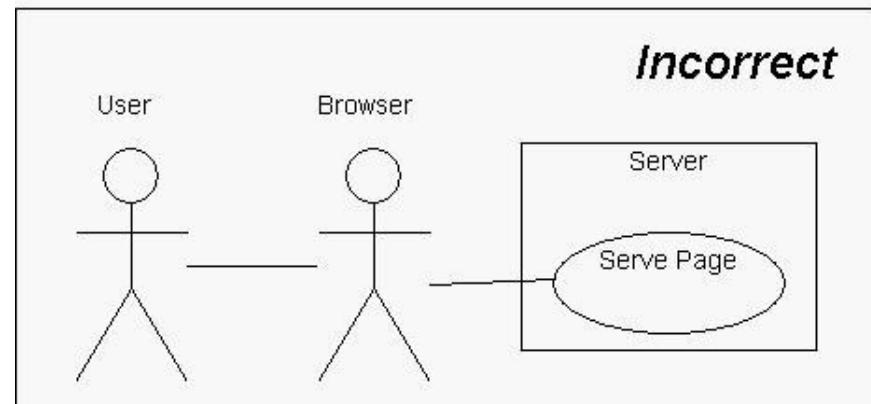
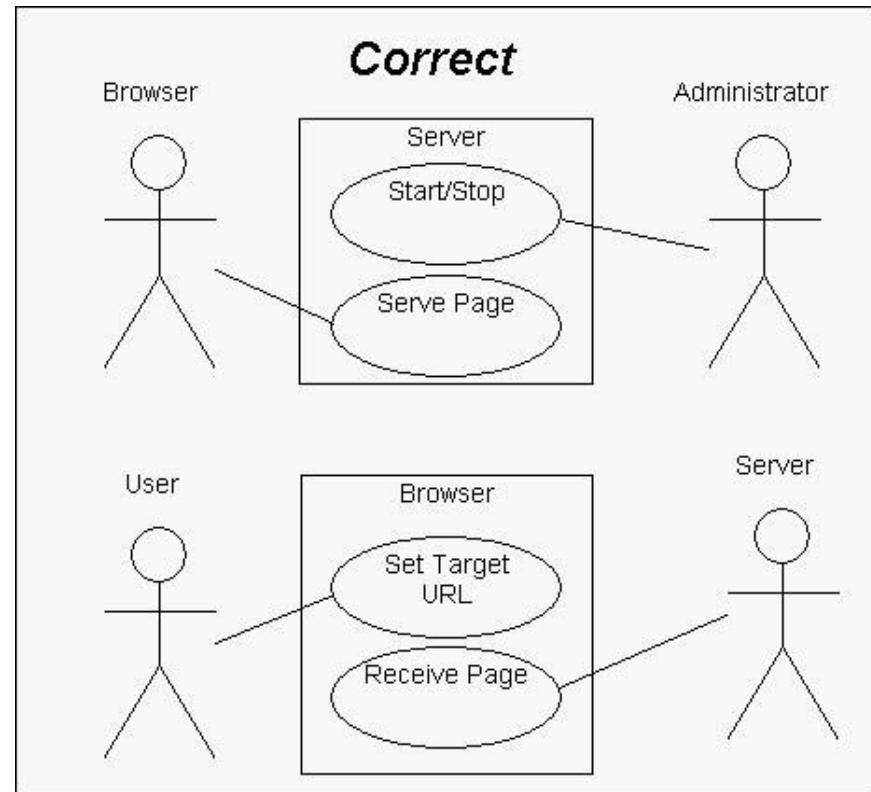
**The previous steps would generate this simple use case diagram:**



**Example2:** In the diagram below we would like to represent the use cases for a camera



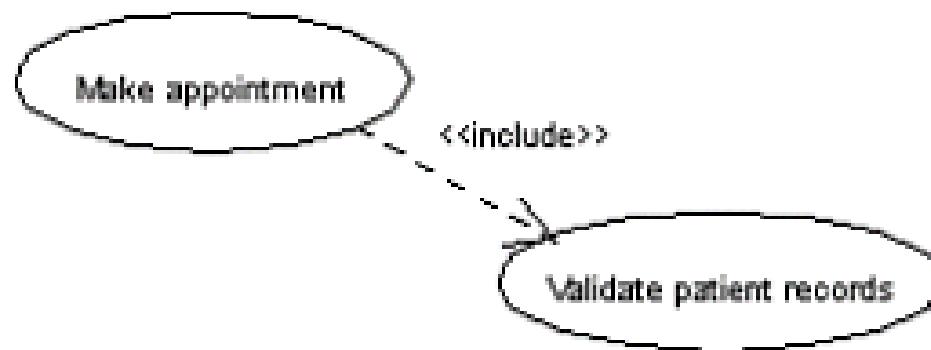
**Example3:** Suppose you wanted to diagram the interactions between a user, a web browser, and the server it contacts.



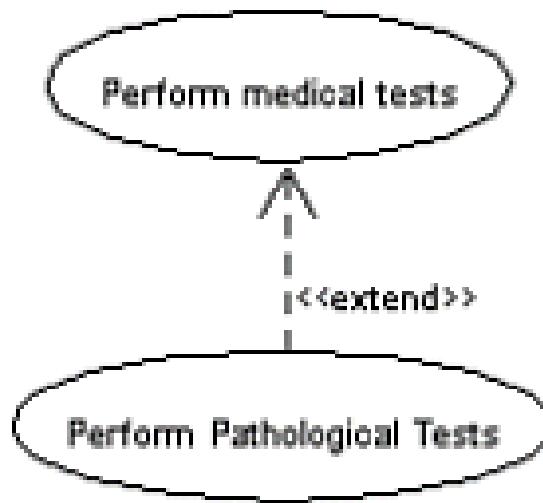
# Relationships in Use Cases

- A relationship between two use cases is basically a dependency between the two use cases. Defining a relationship between two use cases is the decision of the modeler of the use case diagram.
- Reuse of an existing use case using different types of relationships reduces the overall effort required in defining use cases in a system.
- Use case relationships can be one of the following:
  - **Include/uses**
  - **Extend**

- **Include/uses:** When a use case is depicted as using the functionality of another use case in a diagram, this relationship between the use cases is named as an *include* relationship.
  - Literally speaking, in an *include* relationship, a use case includes the functionality described in the another use case as a part of its business process flow.
  - An include relationship is depicted with a directed arrow having a dotted shaft. The tip of the arrowhead points to the parent use case and the child use case is connected at the base of the arrow.

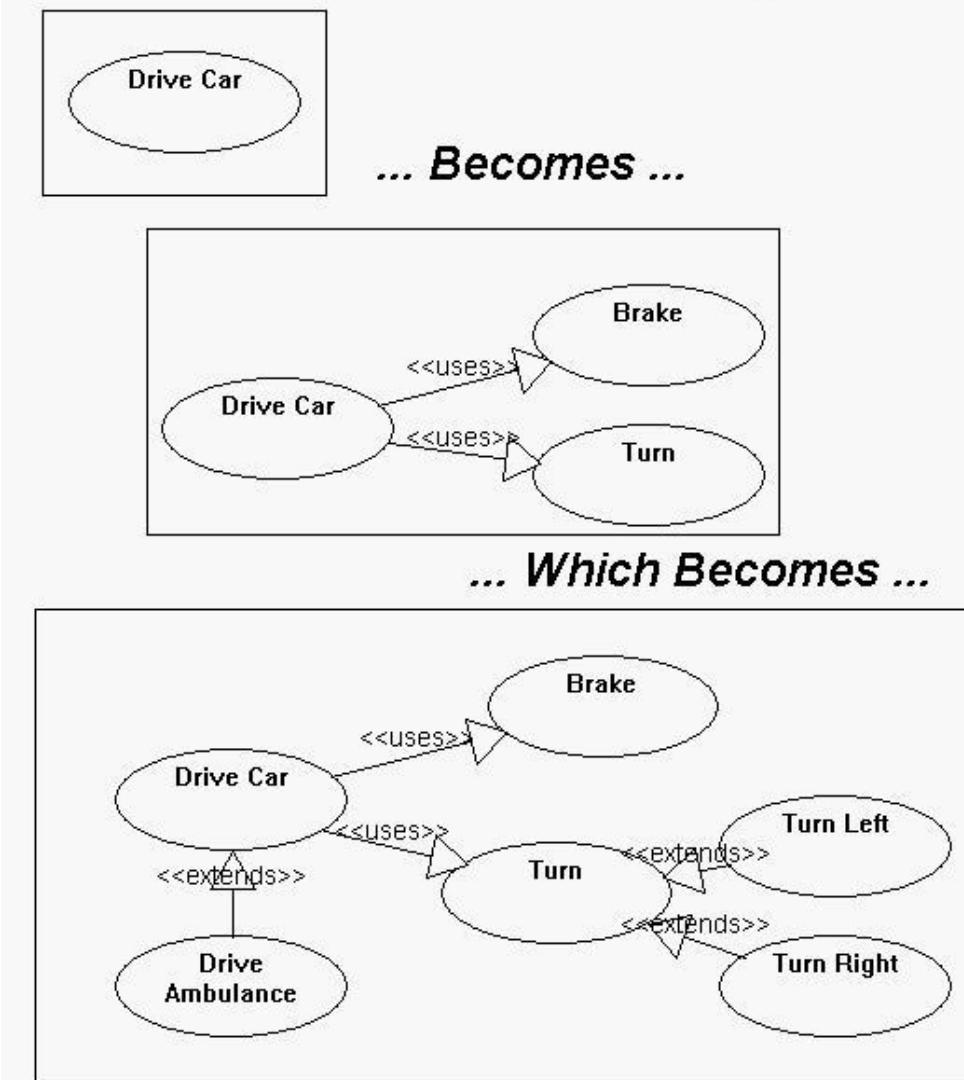


- **Extend:** In an *extend* relationship between two use cases, the child use case adds to the existing functionality and characteristics of the parent use case.
- The tip of the arrowhead points to the parent use case and the child use case is connected at the base of the arrow. The stereotype "<<extend>>" identifies the relationship as an extend relationship.



## *Evolution of a UML Use Case Diagram*

### Example 4



# **Assignment No. 1:** Airline Reservation system with focus on Checkin, Weigh Luggage and Assign seat.

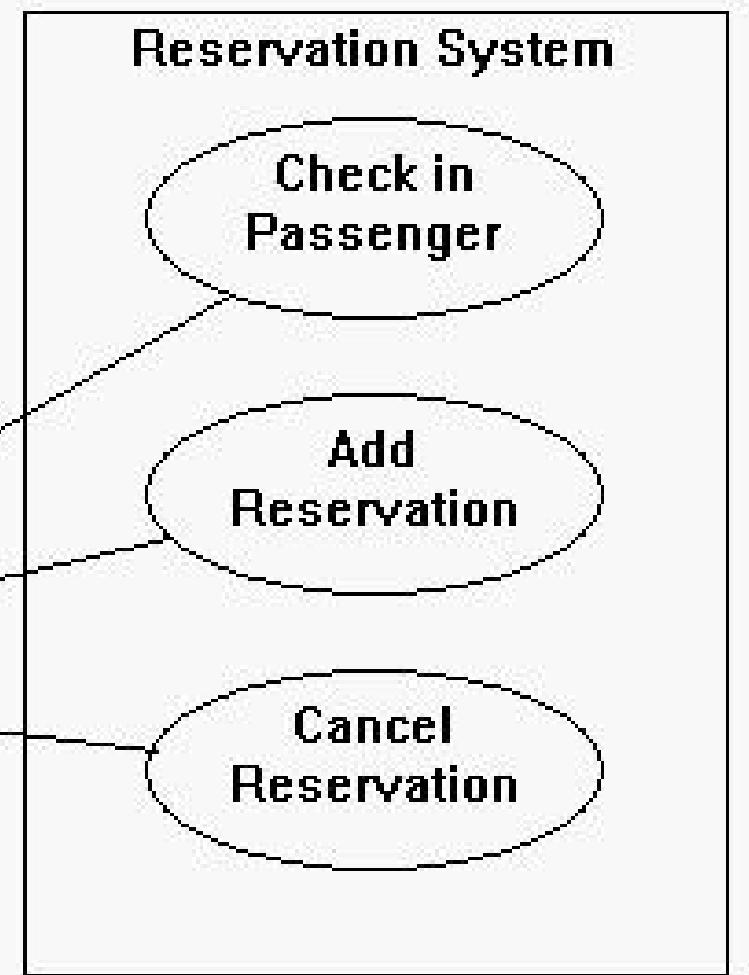
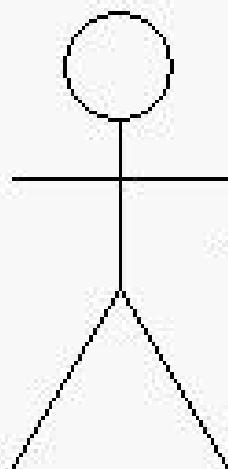
## **Example 5:**

Suppose you wanted to add detail to the diagram shown in the next slide, representing an airline reservation system. First, you would create a separate diagram for the top-level services, and then you would add new use cases that make up the top-level ones. There is a uses edge from "Check in Passenger" to "Weigh Luggage" and from "Check in Passenger" to "Assign Seat"; this indicates that *in order to Check in a Passenger, Luggage must be Weighed and a Seat must be Assigned*.

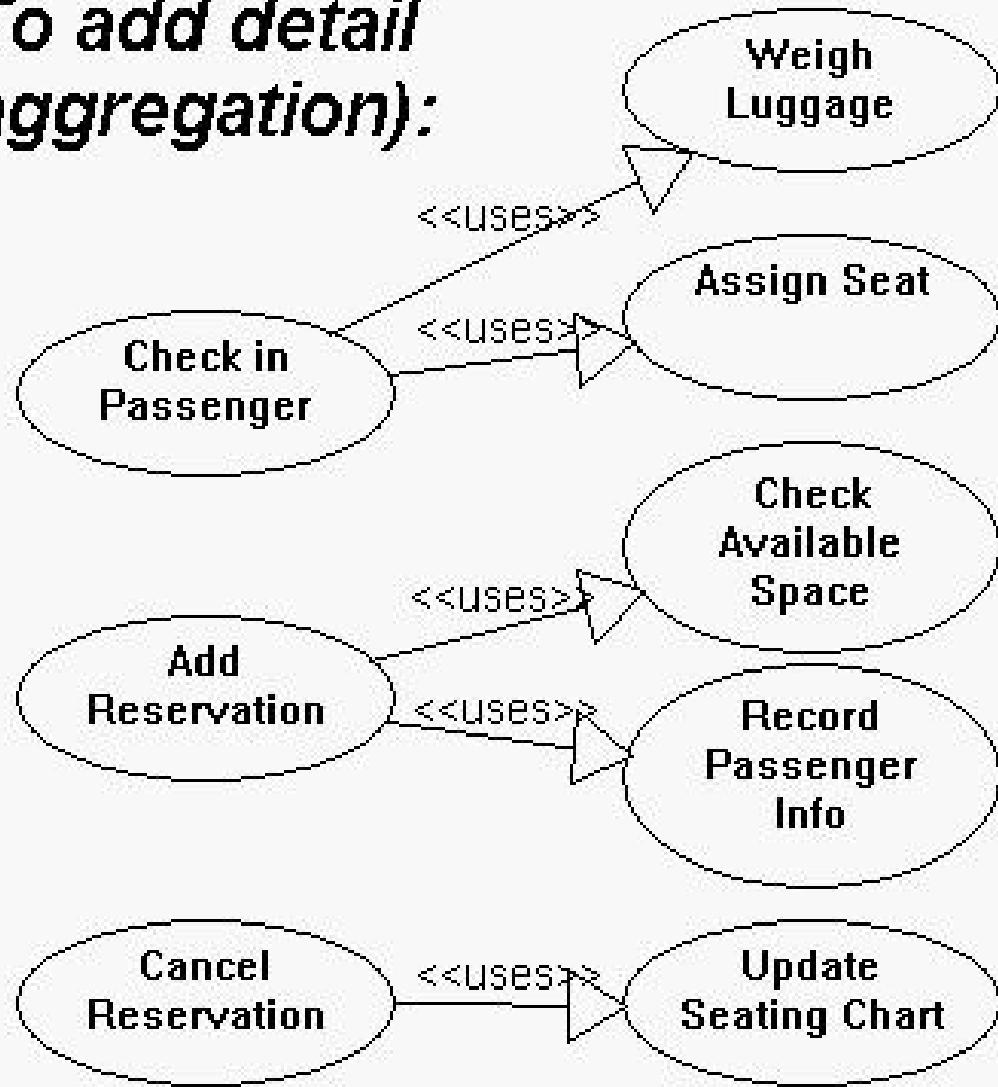
Similarly, the diagram indicates that in order to add a reservation to the system, the available space must be checked and the passenger's information must be recorded

# *Initial Design:*

Ticket Clerk



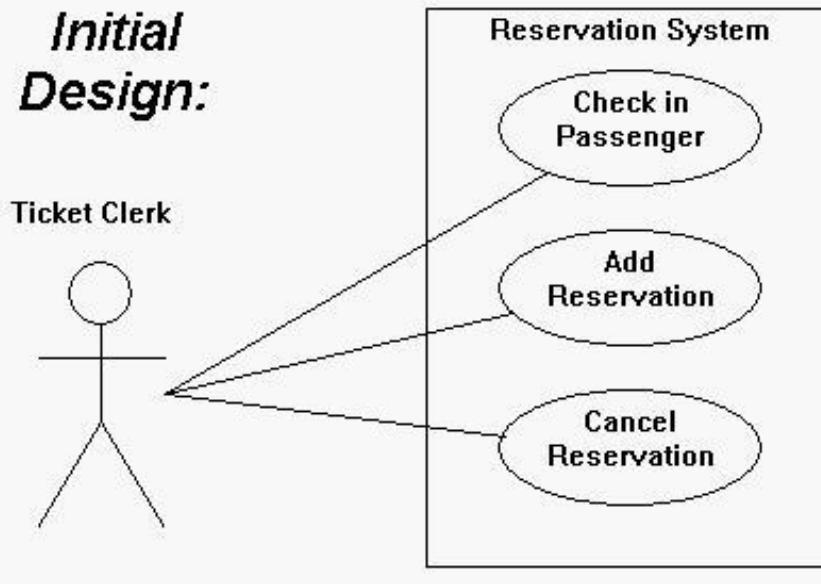
**To add detail  
(aggregation):**



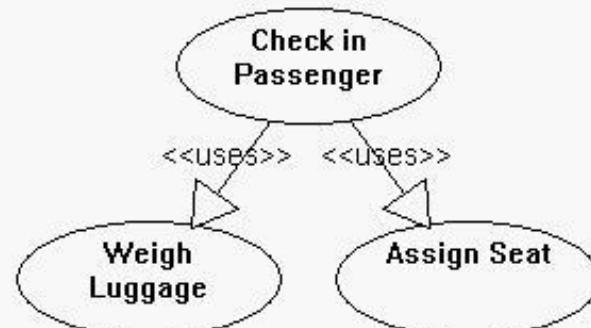
**More detailed  
design**

what you would like to show is that not all of the seats aboard the airplane are exactly alike (some window and some aisle seats), and sometimes passengers will express a preference for one of these types of seats but not the other. But of course, they cannot just be given their preference right away, because the seat they want might not be available. Therefore, the process of assigning a window seat involves checking for the availability of window seats, whereas the process of assigning an aisle seat involves checking for the availability of aisle seats

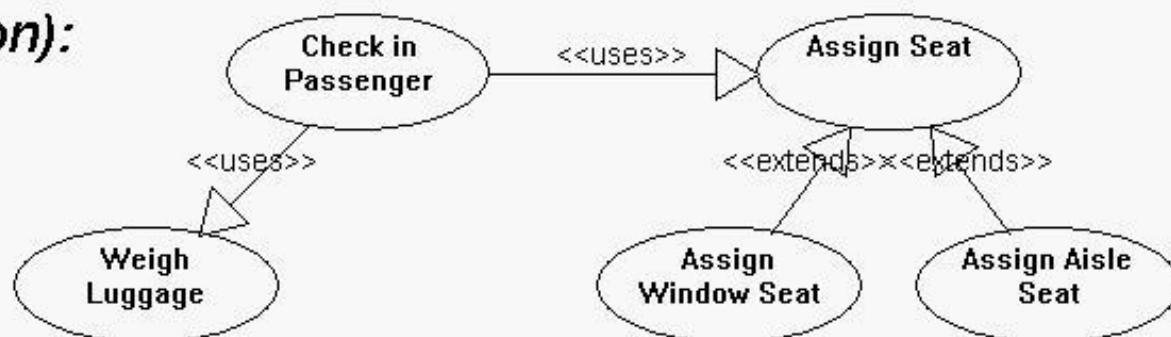
## *Initial Design:*



## *Sub-Diagram:*



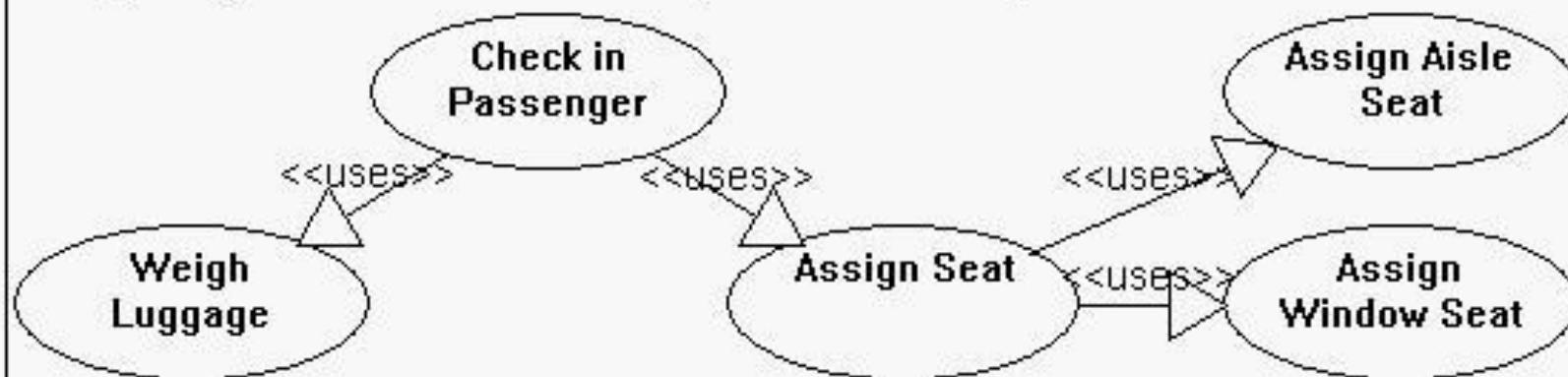
## *To add detail (extension):*



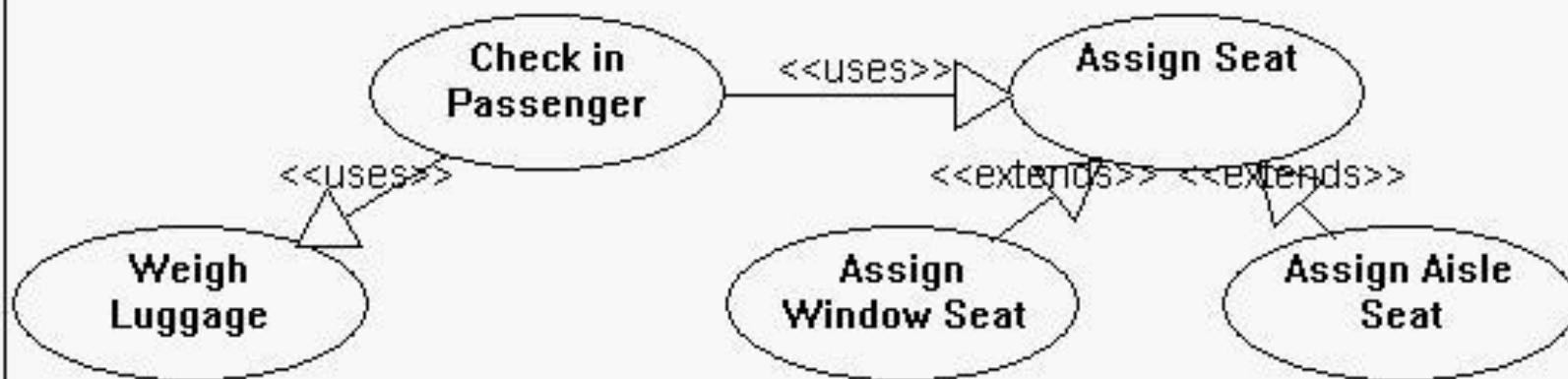
# What is the difference between extends and uses/include?

- "*X uses Y*" or "*X include Y*" indicates that the task "*X*" **has a** subtask "*Y*"; that is, in the process of completing task "*X*", task "*Y*" will be completed at least once
- "*X extends Y*" indicates that "*X*" **is a** task to the same type as "*Y*", but "*X*" is a special, more specific case of doing "*Y*". That is, doing *X* is a lot like doing *Y*, but *X* has a few extra processes to it that go above and beyond the things that must be done in order to complete *Y*.

*Trying to add detail (incorrect):*



*Trying to add detail (correct):*



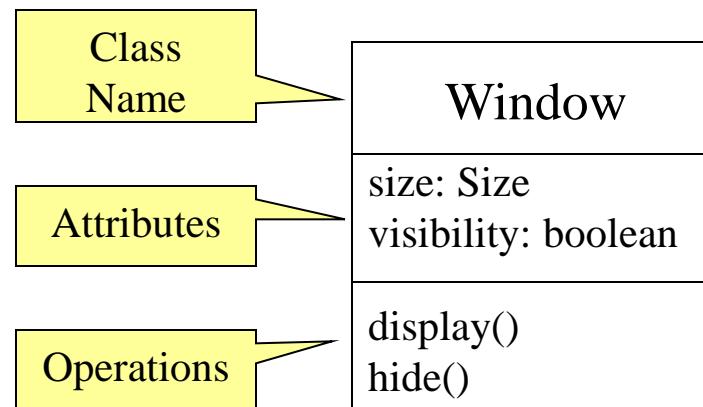
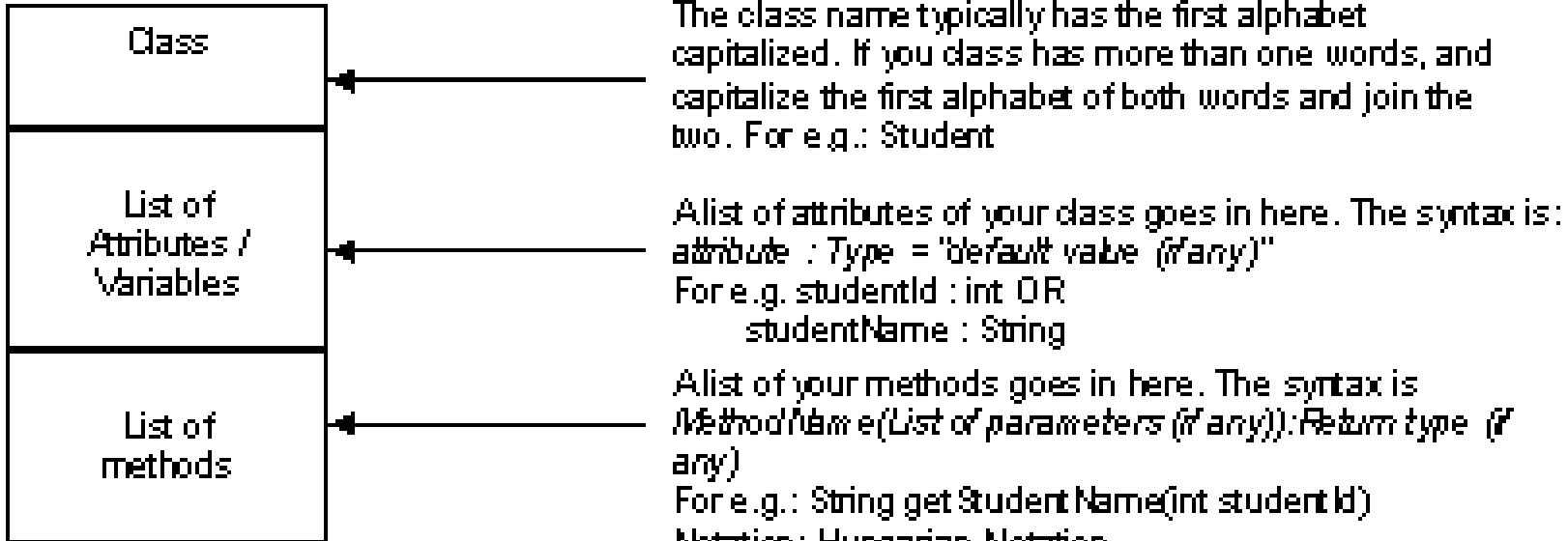
# Class Diagram

- **Definition:** A class diagram is a diagram showing a collection of classes and interfaces, along with the collaborations and relationships among classes and interfaces.
- When you designed the use cases, you must have realized that the use cases talk about "what are the requirements" of a system?
- The aim of designing classes is to convert this "what" to a "how" for each requirement. Each use case is further analyzed and broken up into atomic components that form the basis for the classes that need to be designed.

- A class diagram is a pictorial representation of the *detailed* system design.
- A thing to remember is that a class diagram is a static view of a system. The structure of a system is represented using class diagrams. Class diagrams are referenced time and it used by the developers while implementing the system.
- Class diagrams are used in nearly all Object Oriented software designs. Use them to describe the Classes of the system and their relationships to each other.
- Note that these diagrams describe the relationships between *classes*, not those between specific *objects* instantiated from those classes

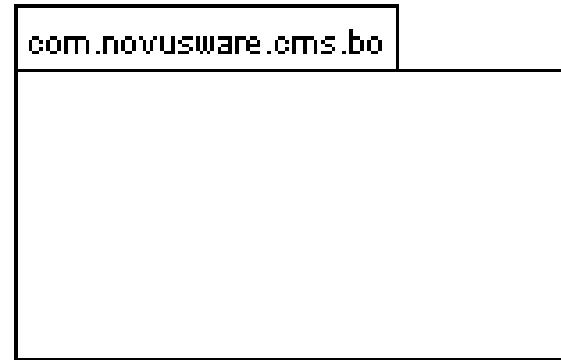
## Elements of a Class Diagram

- **Class:** A class represents an entity of a given system that provides an encapsulated implementation of certain functionality of a given entity. These are exposed by the class to other classes as *methods*. A class also has properties that reflect unique features of a class. The properties of a class are called *attributes*.
- As an example, let us take a class named Student. A Student class represents student entities in a system. The Student class encapsulates student information such as student id #, student name, and so forth. Student id, student name, and so on are the attributes of the Student class. The Student class also exposes functionality to other classes by using methods such as `getStudentName()`, `getStudentId()`, and the like



- *Interface*: An interface is a variation of a class. As we saw from the previous point, a class provides an encapsulated implementation of certain business functionality of a system. An interface on the other hand provides only a definition of business functionality of a system. A separate class implements the actual business functionality.
- You can define an abstract class that declares business functionality as abstract methods. A child class can provide the actual implementation of the business functionality

- *Package*: A package provides the ability to group together classes and/or interfaces that are either similar in nature or related. Grouping these design elements in a package element provides for better readability of class diagrams, especially complex class diagrams.



# Relationships Between Classes

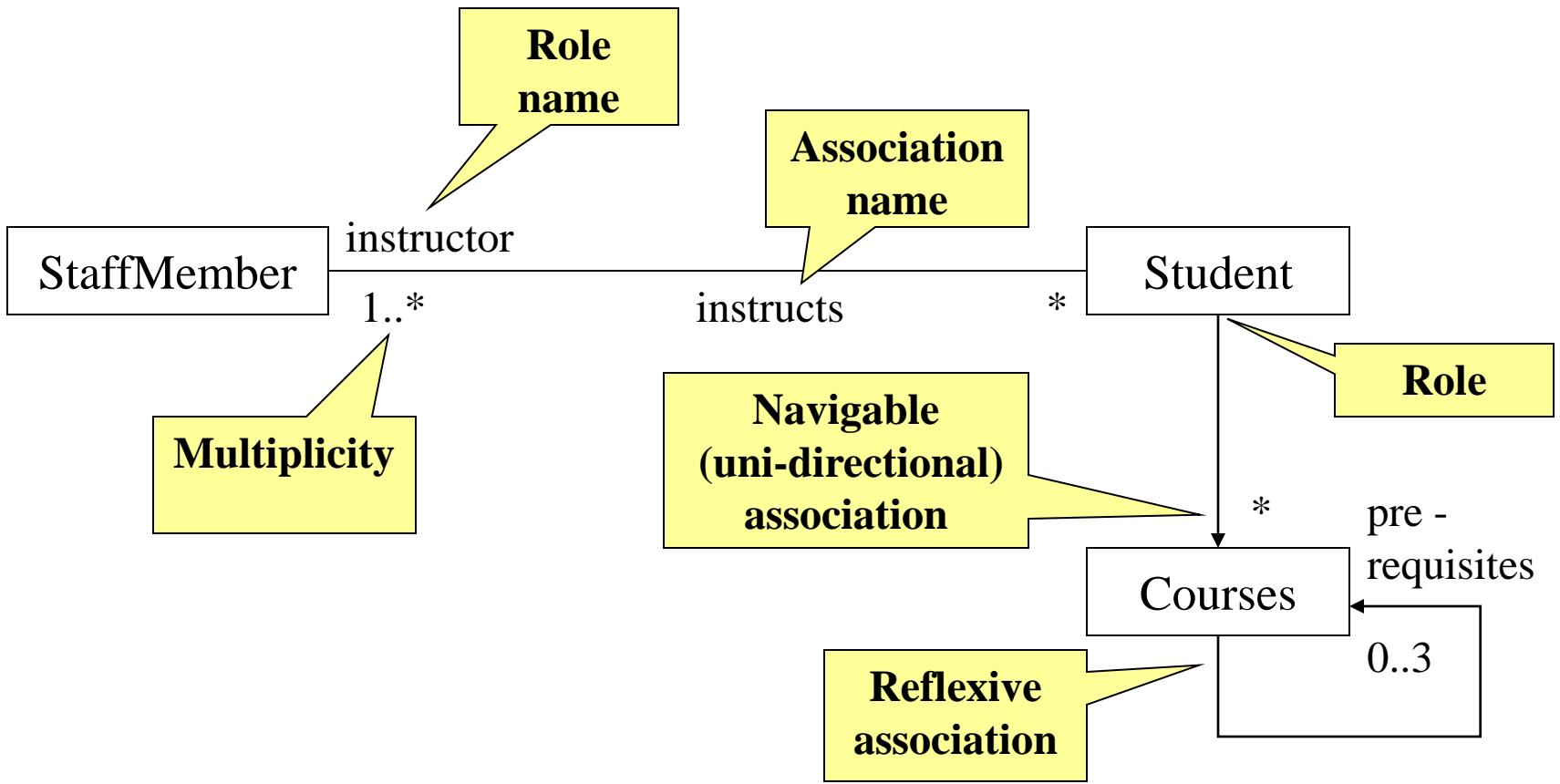
- *Lines* that model the relationships between classes and interfaces in the system.
- *Generalization*
  - *Inheritance*: a solid line with a solid arrowhead that points from a sub-class to a superclass or from a sub-interface to its super-interface.
  - *Implementation*: a dotted line with a solid arrowhead that points from a class to the interface that it implements
- *Association* -- a solid line with an open arrowhead that represents a "has a" relationship. The arrow points from the containing to the contained class

- A semantic relationship between two or more classes that specifies connections among their instances.
- A structural relationship, specifying that objects of one class are connected to objects of a second (possibly the same) class.
- Example: “An Employee works for a Company”



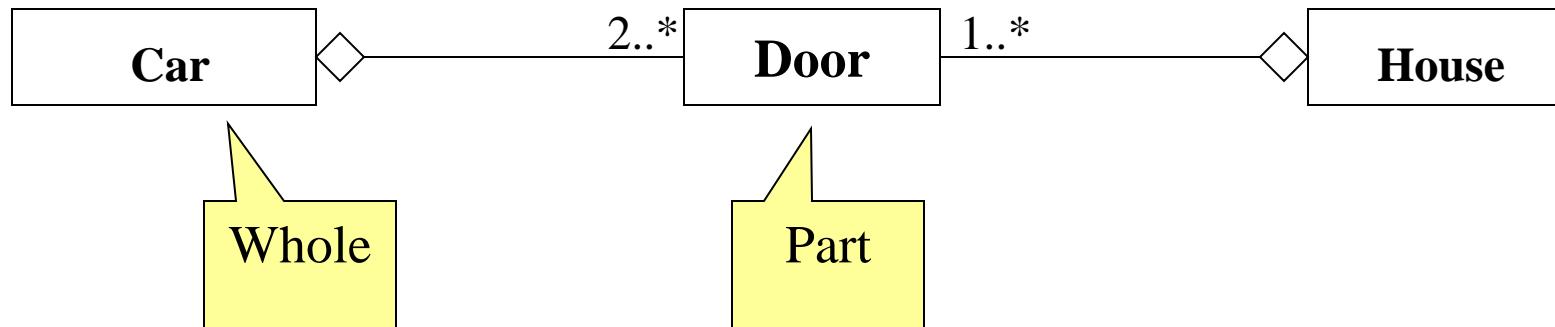
## **Associations can be one of the following two types:**

- *Composition*: Represented by an association line with a solid diamond at the tail end. A composition models the notion of one object "owning" another and thus being responsible for the creation and destruction of another object.
- *Aggregation*: Represented by an association line with a hollow diamond at the tail end. An aggregation models the notion that one object uses another object without "owning" it and thus is *not* responsible for its creation or destruction.



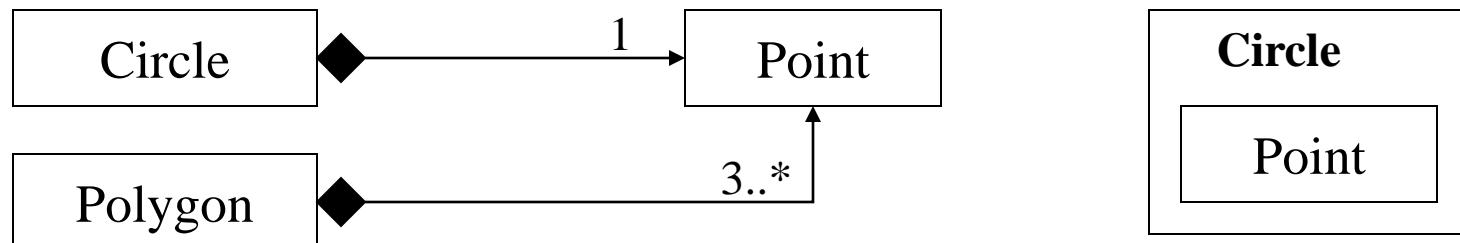
# Aggregation

- A special form of association that models a whole-part relationship between an aggregate (the whole) and its parts.
  - Models a “is a part-of” relationship.



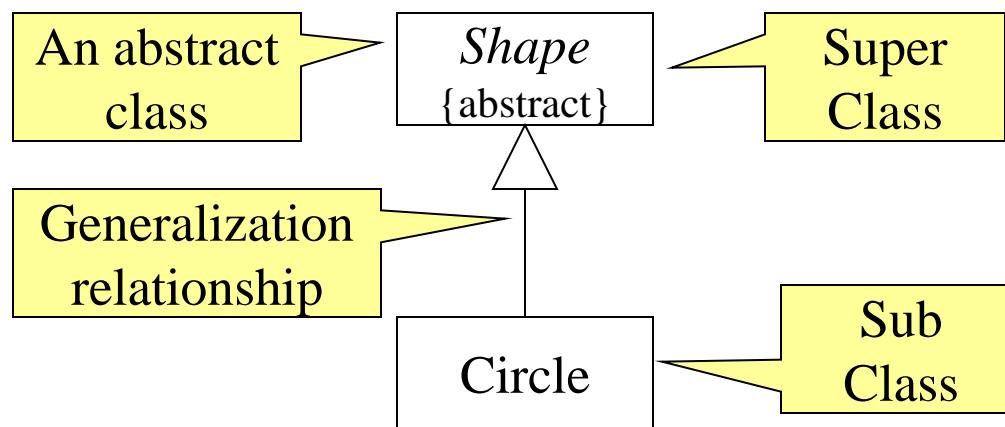
# Composition

- A strong form of aggregation
  - The whole is the sole owner of its part.
    - The part *object* may belong to only one whole
  - The life time of the part is dependent upon the whole.
    - The composite must manage the creation and destruction of its parts.



# Generalization

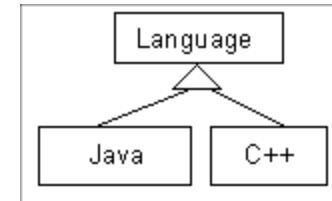
- Indicates that objects of the specialized class (subclass) are substitutable for objects of the generalized class (super-class).
  - “is a” relationship.



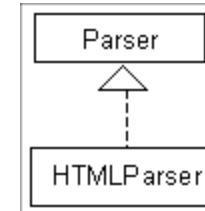
- A sub-class inherits from its super-class
  - Attributes
  - Operations
  - Relationships
- A sub-class may
  - Add attributes and operations
  - Add relationships
  - Refine (override) inherited operations

*Generalization*

Inheritance

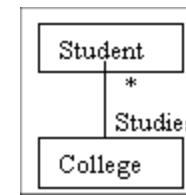


Implementation

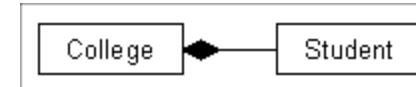


*Association*

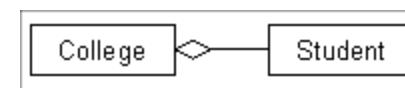
Multiplicity :  
many students  
belonging to same  
college.



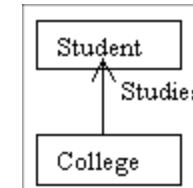
Composition



Aggregation



Directed Association



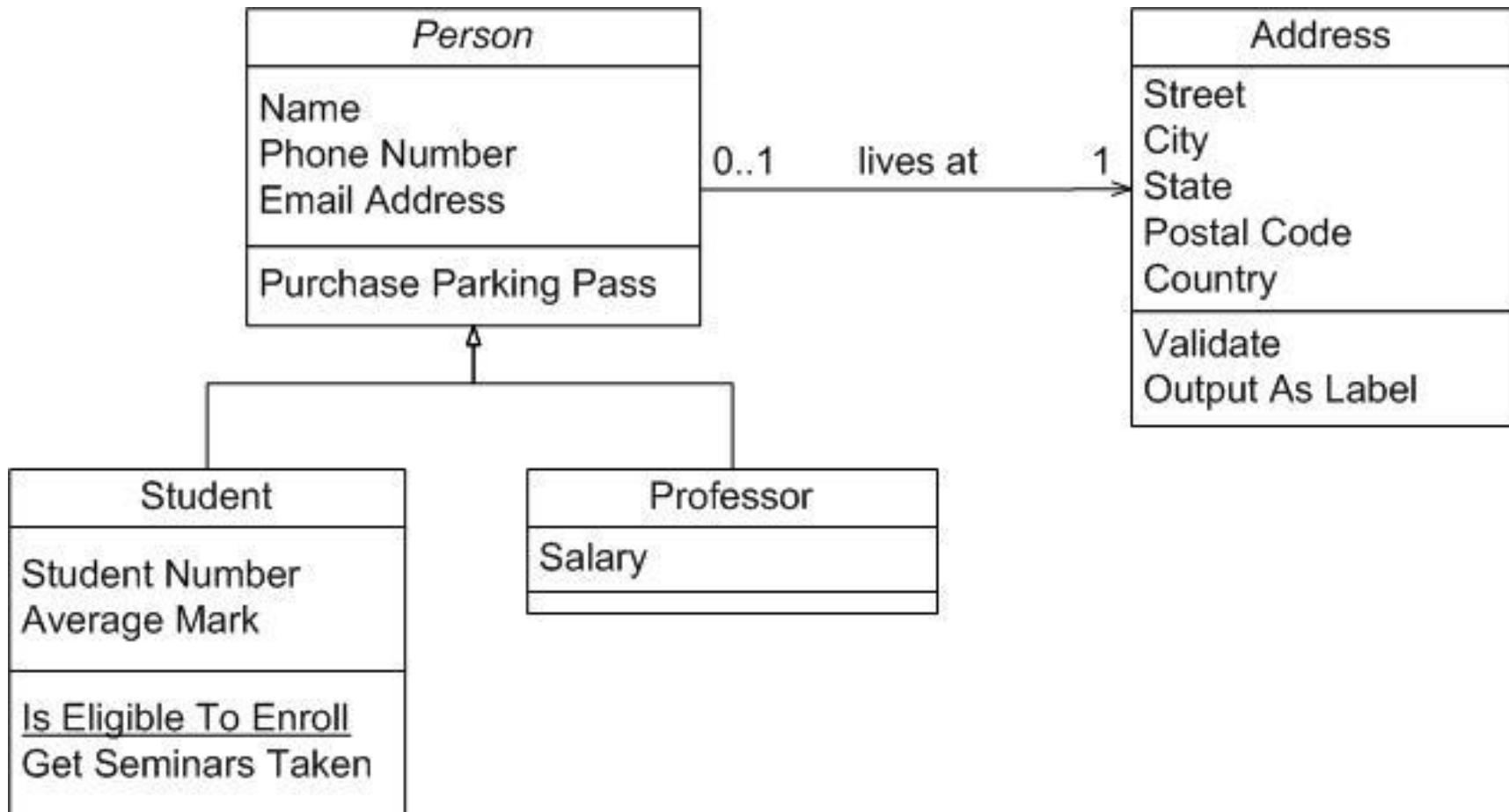
## A Few Terms

- *Responsibility of a class:* It is the statement defining what the class is expected to provide.
- *Stereotypes:* Classes created at the early phases of the system design.
- *Boundary class:* Users interact with the system through the boundary classes. (interface classes)
- *Control class:* A control class typically does not perform any business functions, but only redirects to the appropriate business function class depending on the function requested by the boundary class or the user.
- *Entity class:* An entity class consists of all the business logic and interactions with databases.

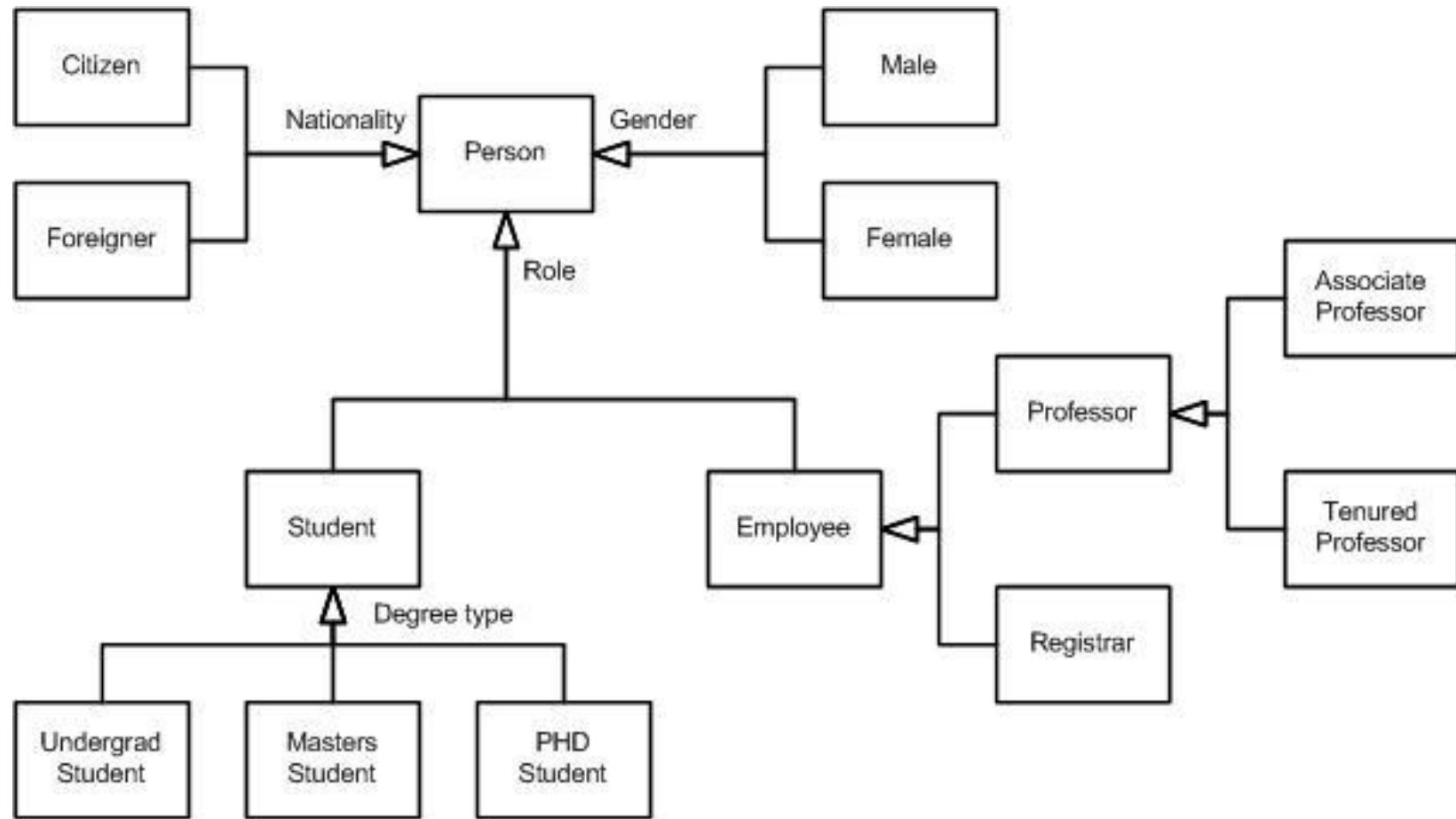
# Multiplicity Indicators.

Indicator	Meaning
0..1	Zero or one
1	One only
0..*	Zero or more
1..*	One or more
n	Only $n$ (where $n > 1$ )
0..n	Zero to $n$ (where $n > 1$ )
1..n	One to $n$ (where $n > 1$ )

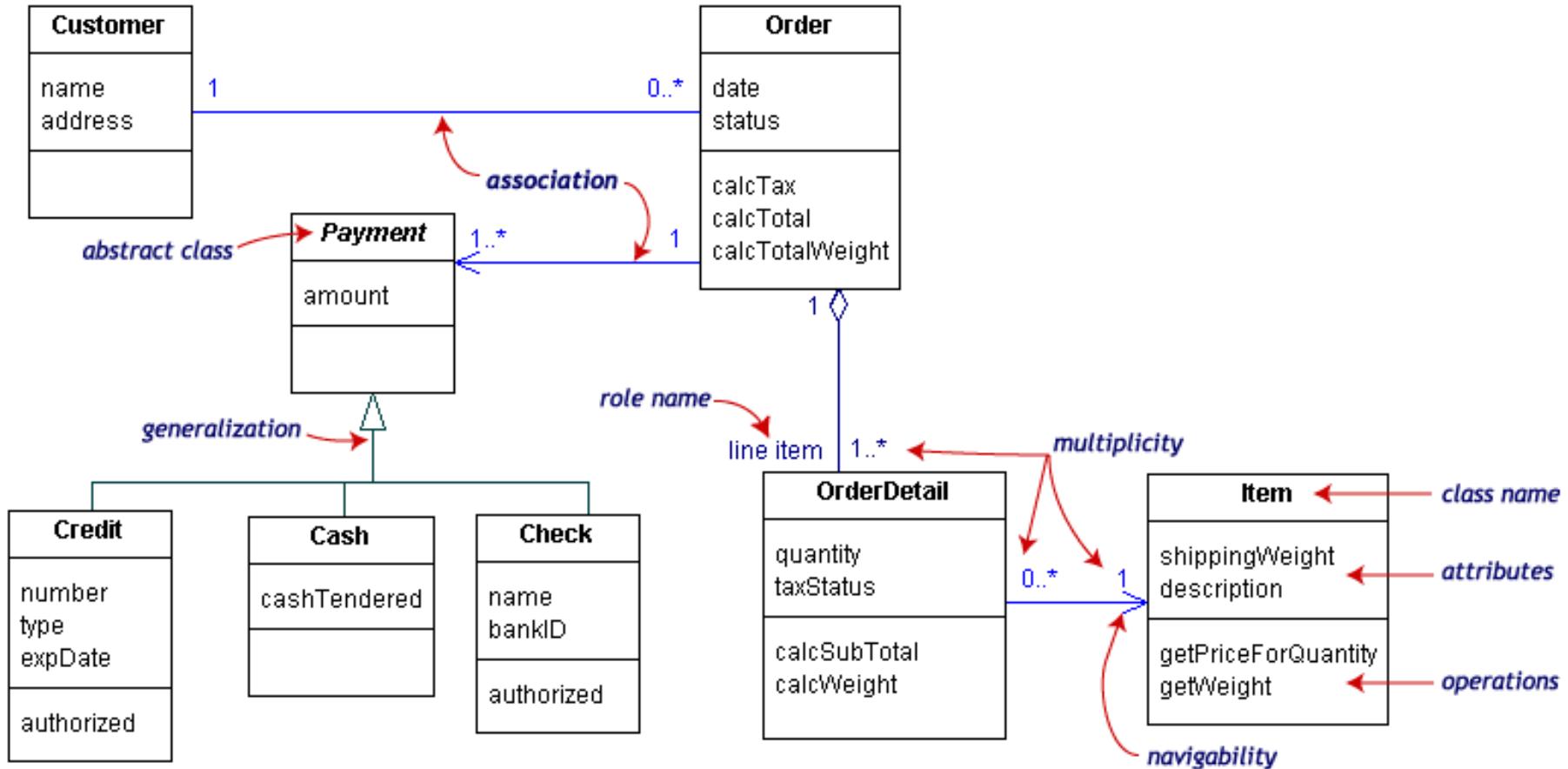
# Assignment No. 2: Relationship between Student, Professor and Employee of the company (Page 108, UML User Guide)

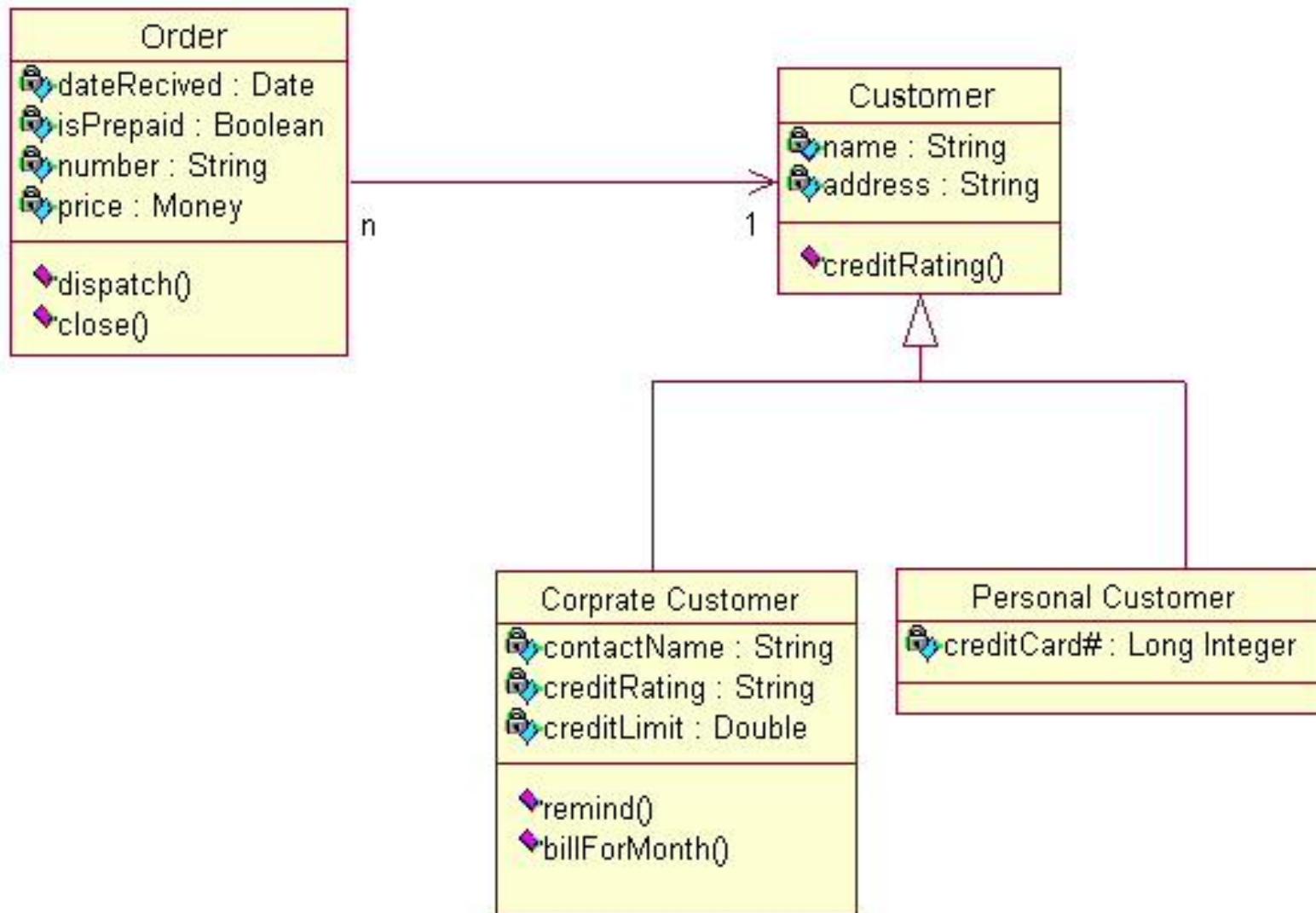


# Assignment No. 2: Relationship between Student, Professor and Employee of the company



# Customer Order & Payment



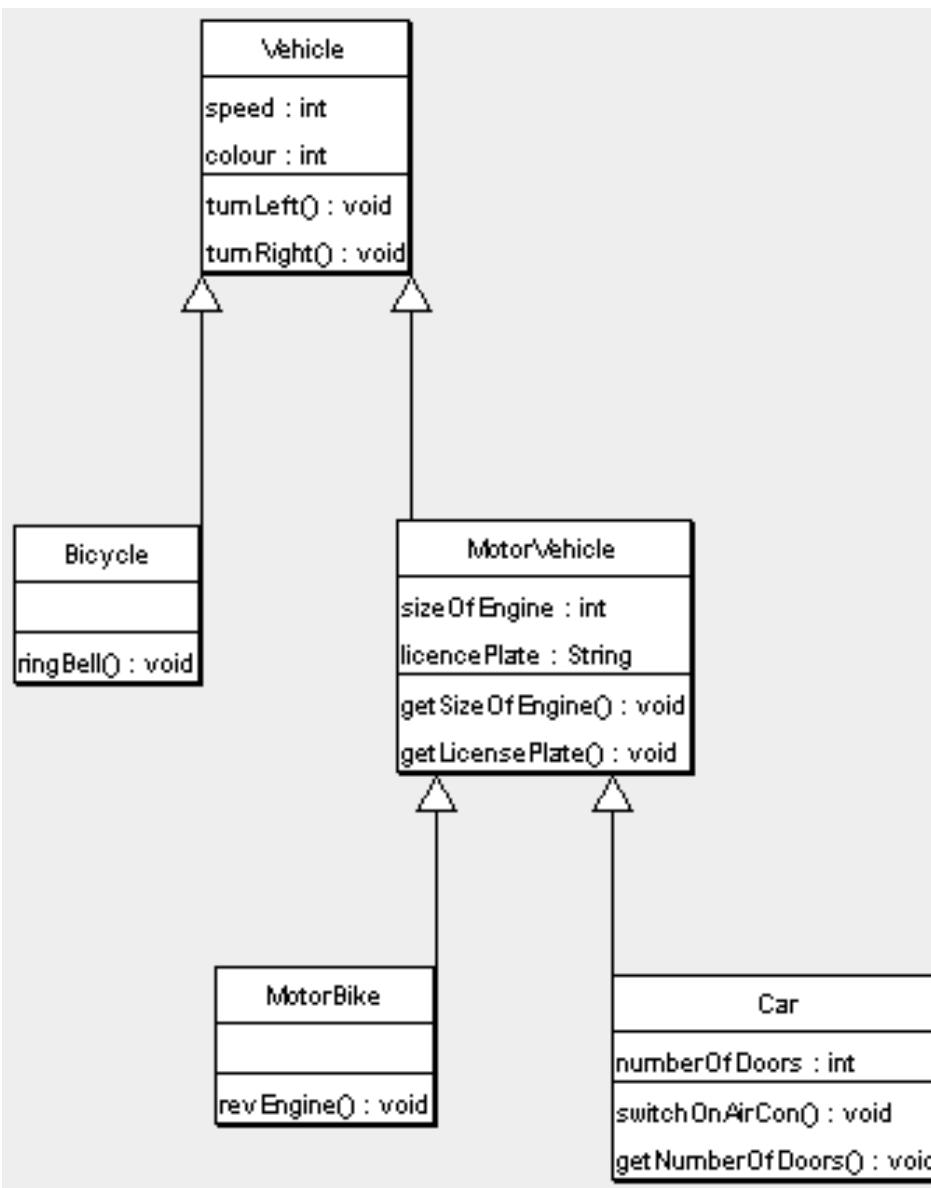


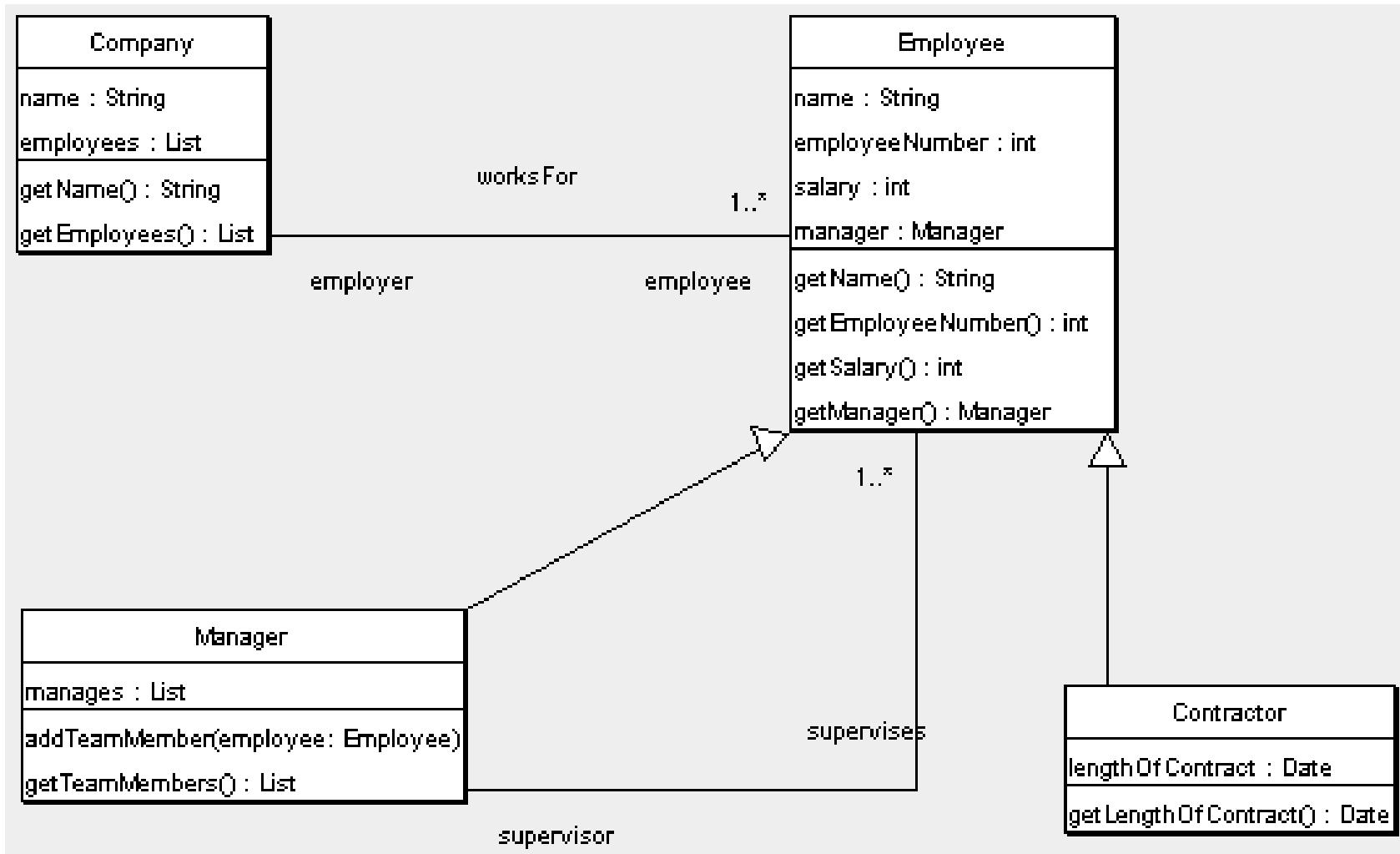
## Example Vehicles

We would like to model an application that shows different kinds of vehicles such as bicycles, motor bike and cars.

Notes:

- All Vehicles have some common attributes (speed and colour) and common behaviour (turnLeft, turnRight)
- Bicycle and MotorVehicle are both kinds of Vehicle and are therefore shown to inherit from Vehicle. To put this another way, Vehicle is the superclass of both Bicycle and MotorVehicle
- In our model MotorVehicles have engines and license plates. Attributes have been added accordingly, along with some behaviour that allows us to examine those attributes
- MotorVehicles is the base class of both MotorBike and Car, therefore these classes not only inherit the speed and colour properties from Vehicle, but also the additional attributes and behaviour from MotorVehicle
- Both MotorBike and Car have additional attributes and behaviour which are specific to those kinds of object.





Show a relationship between Rectangle, Circle and Polygon

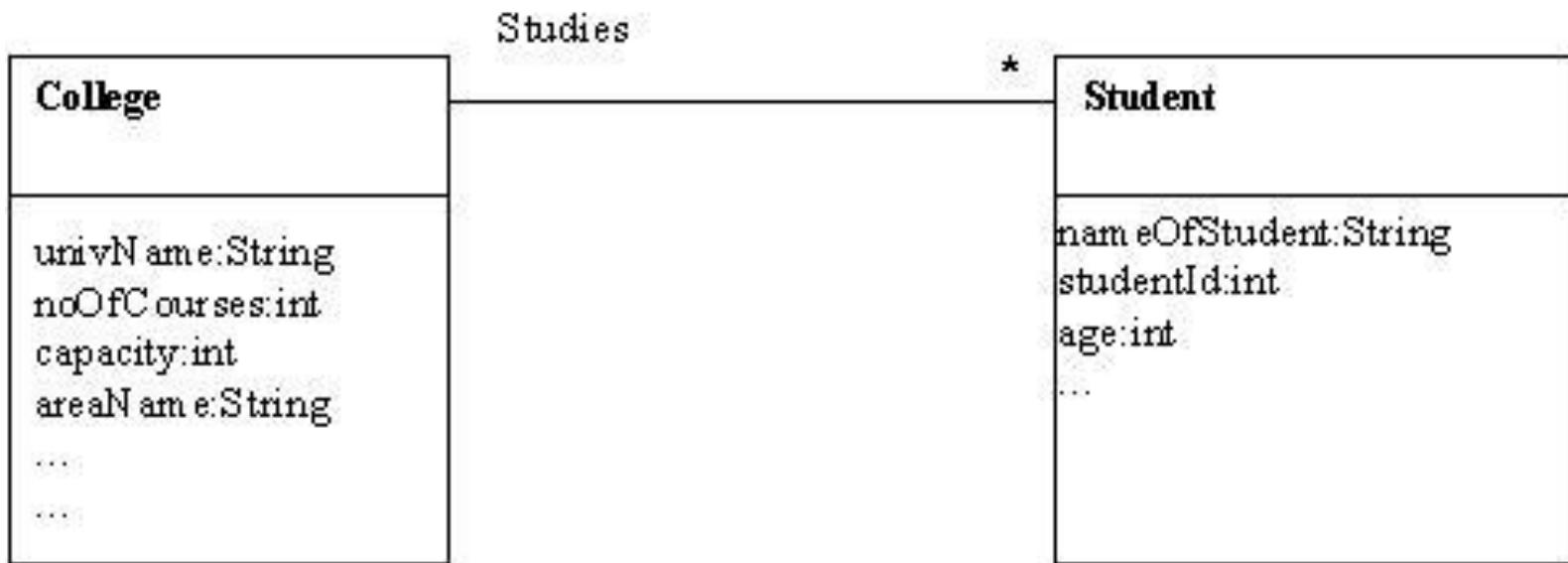
UML – User Guider Page No. 61

# Object Diagrams in UML

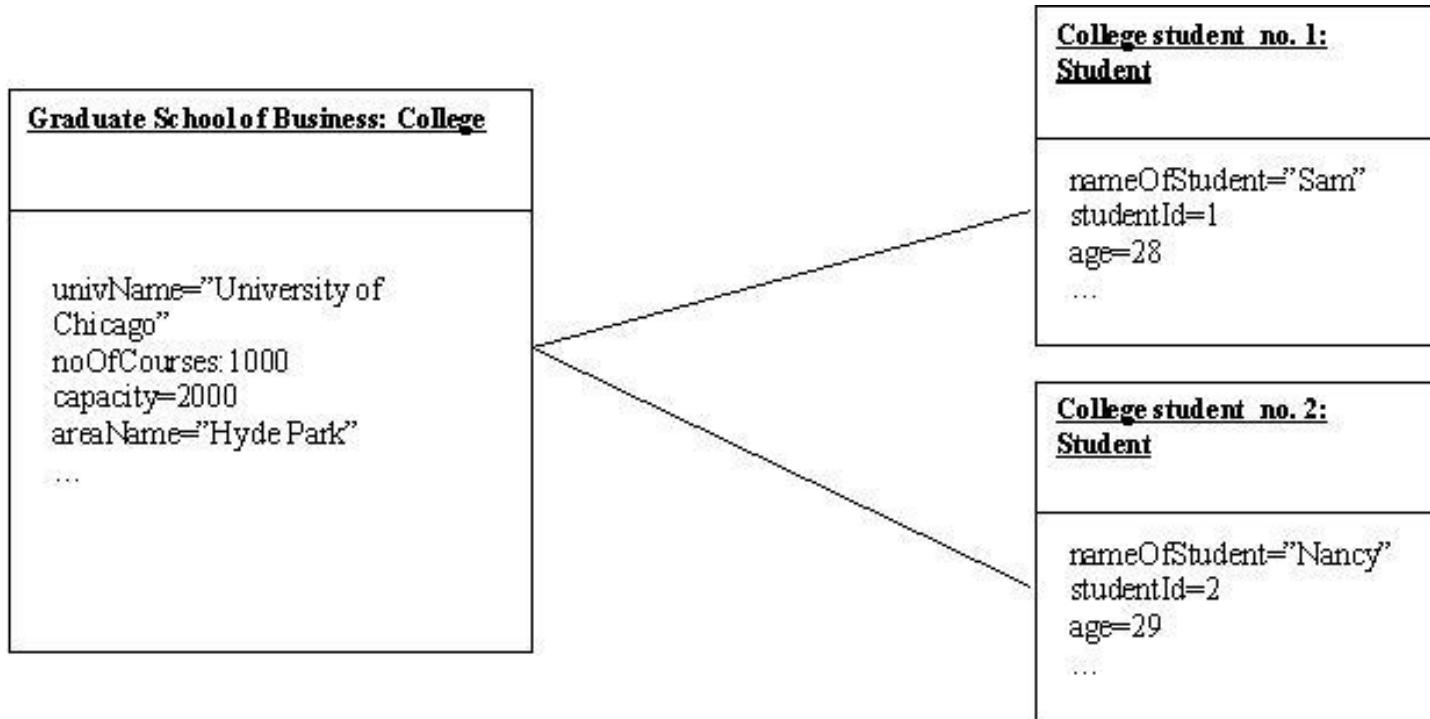
- In a live application classes are not directly used, but instances or objects of these classes are used. A pictorial representation of the relationships between these instantiated classes at any point of time (called objects) is called an "**Object diagram.**"
- It looks very similar to a class diagram, and uses the similar notations to denote relationships.
- It reflects the picture of how classes interact with each others at runtime. and in the actual system, how the objects created at runtime are related to the classes.
- shows this relation between the instantiated classes and the defined class, and the relation between these objects.

# Elements of an Object Diagram

The minor difference between class diagram and the object diagram is that, the class diagram shows a class with attributes and methods declared. However, in an object diagram, these attributes and method parameters are allocated values.



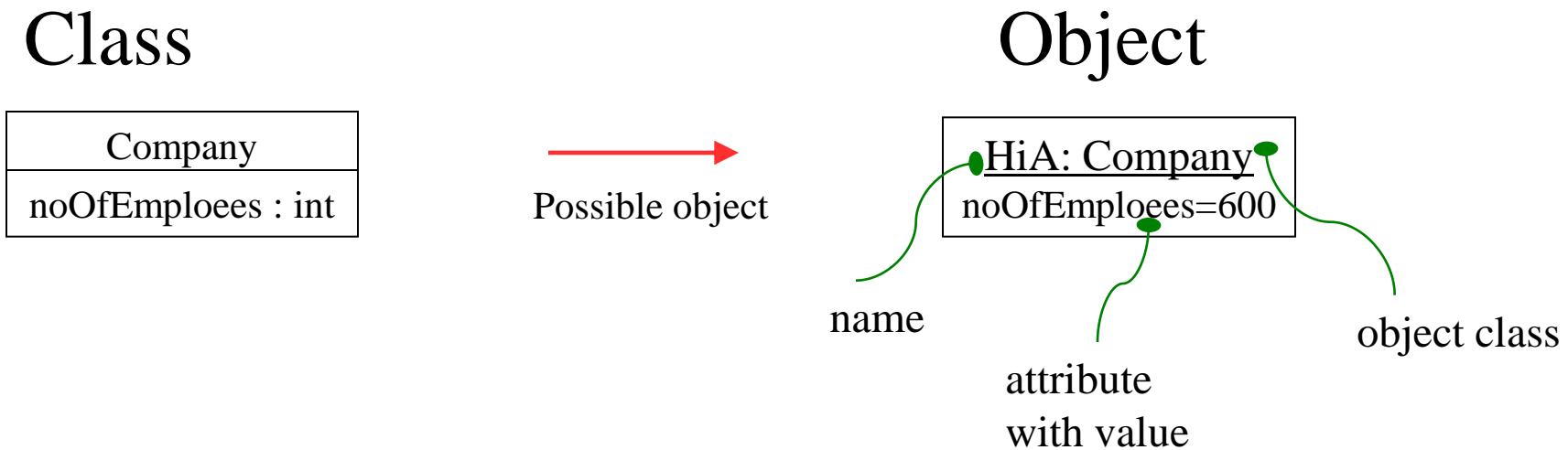
Now, when an application with the class diagram as shown above is run, instances of College and Student class will be created, with values of the attributes initialized. The object diagram for such a scenario will be represented as shown below:



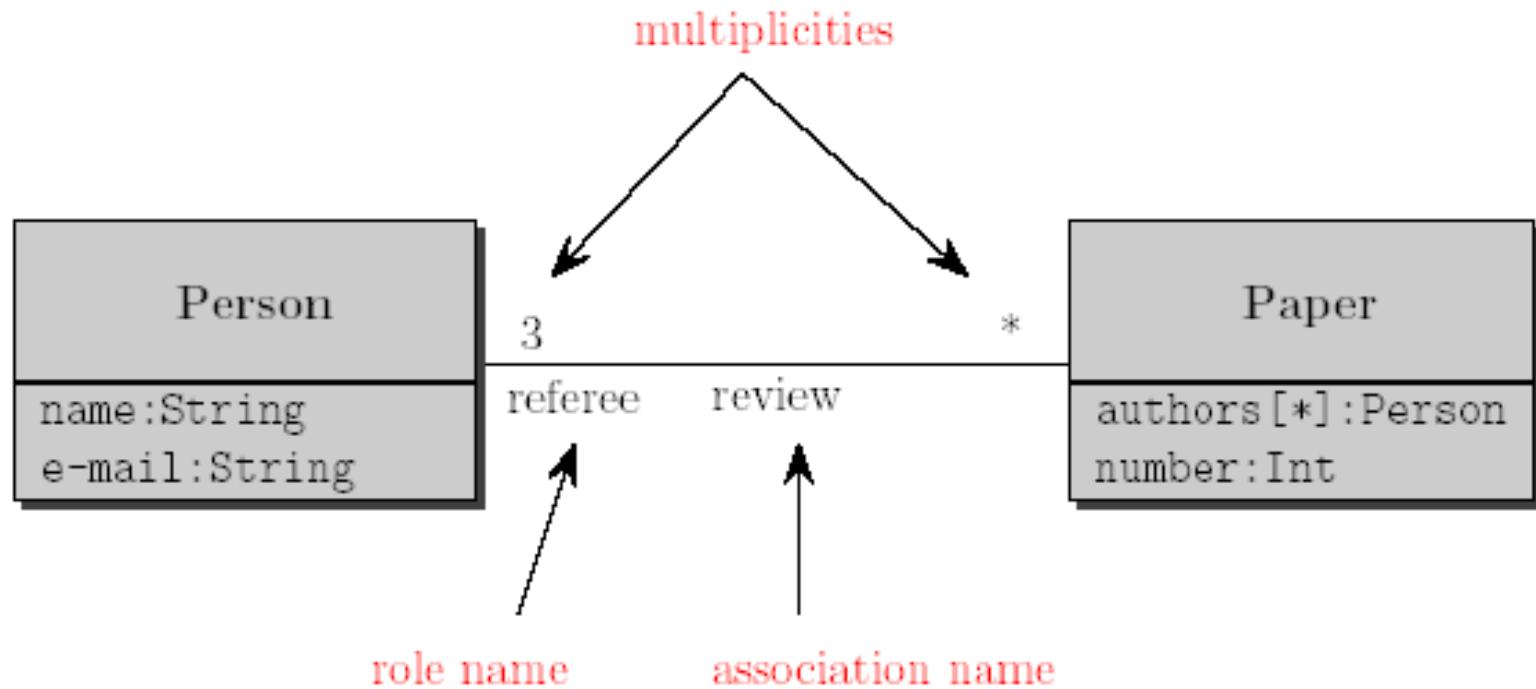
The object diagram shows the name of the instantiated object, separated from the class name by a ":" , and underlined, to show an instantiation.

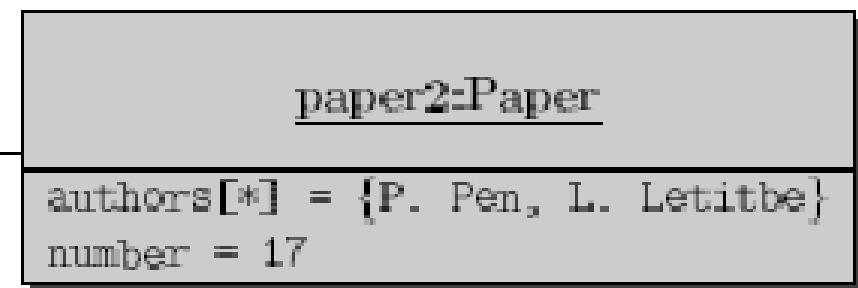
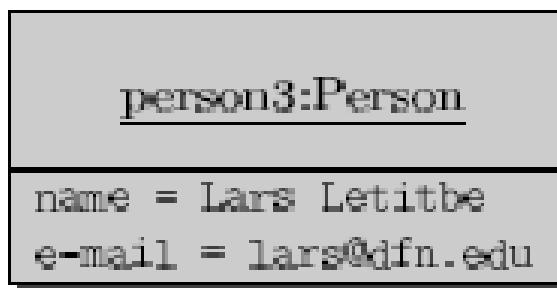
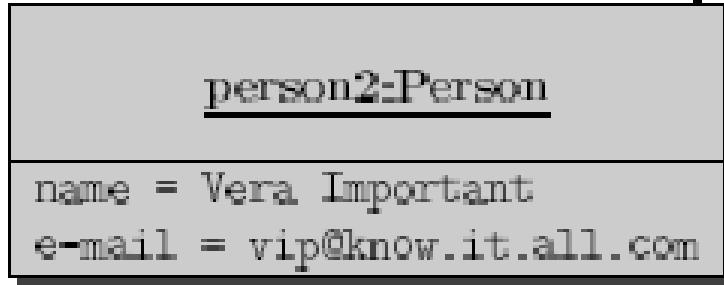
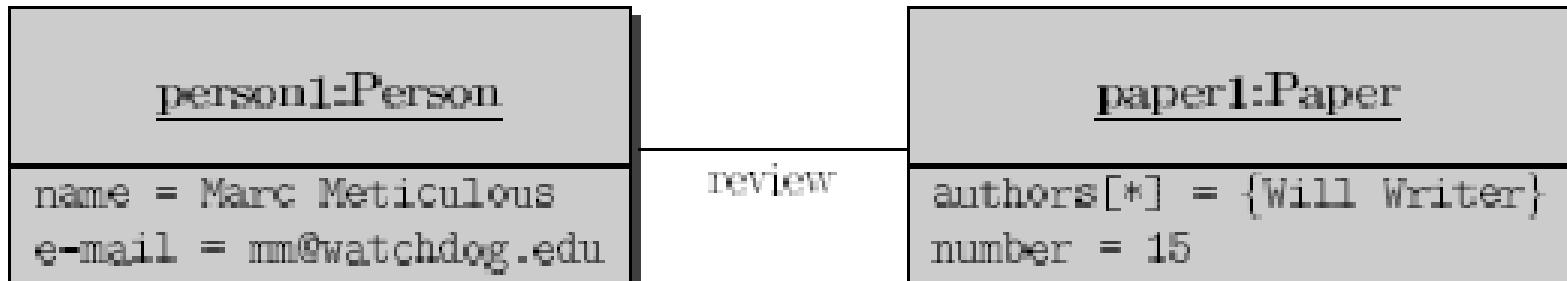
Eg. Graduate School of Business: College

Example2:



## Example 3





## **When to use object diagrams?**

- Use the object diagram as a means of debugging the functionality of your system.
- Check whether the system has been designed as per the requirements, and behaves how the business functionality needs the system to respond.

## **Be careful !!**

- Avoid representing all the objects of your system in an object diagram → complex → unreadable. Use object diagram to represent the state of objects in important or critical flows in your application.

# State Diagram

## Basics

- We are now taking a deeper look at system dynamics.
- Some of the dynamic behavior will be specified in terms of sequencing / timing
- Some of the dynamic behavior will be specified in terms of functions (transformations / computations)
- State diagrams are used to describe the behavior of a system. State diagrams describe all of the possible states of an object as events occur.
- It is important to note that having a State diagram for your system is not a mandatory, but must be defined only on a need basis (to understand the behavior of the object through the entire system)

**Event:** Is Something that happens at a point of time . E.g.  
user presses a button.

One event may logically precede other or follow another.

State may occur between two events

Every event is a unique occurrence

Event may be external or Internal. External events are those that pass between the system and its actor.

Event conveys information from one object to another

**Signals:** A message is a named object that is sent asynchronously by one object and the received by another.

A signal is a classifier for messages; it is a message type. Signals have lot of common with plain classes.

**Call Events:** It represents the receipt by an object of a call request for operations on the object. A call event may trigger a state transition in a state machine or it may invoke a method on the target object.

**Time and Change Events:** A time event is an that represents the passage of time.

**States:** A states is an abstraction of attribute values and links of an object.

Set of values are grouped together into a state according to properties that affect the gross behavior of the object.

A state specifies the response of the object to the input events.

**Scenario:** I a sequence of events that occurs during one particular execution of a system.

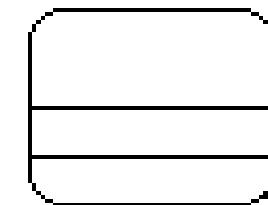
It may include all events in the system or may include only those events impinging or generated by certain objects in the system.

# Elements of a State diagram

**Initial State:** This shows the starting point or first activity of the flow



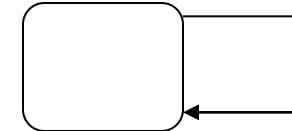
**State:** Represents the state of object at an instant of time. In a state diagram, there will be multiple of such symbols, one for each state of the Object



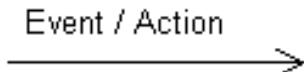
**Transition:** An arrow indicating the Object to transition from one state to the other. The actual trigger event and action causing the transition are written beside the arrow.



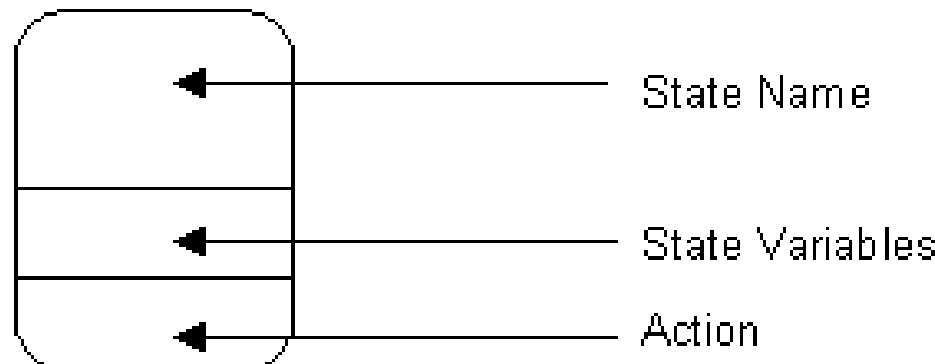
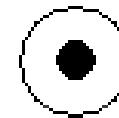
**Self Transitions:** Sometimes an object is required to perform some action when it recognizes an event, but it ends up in the same state it started in



**Event and Action:** A trigger that causes a transition to occur is called as an event or action.

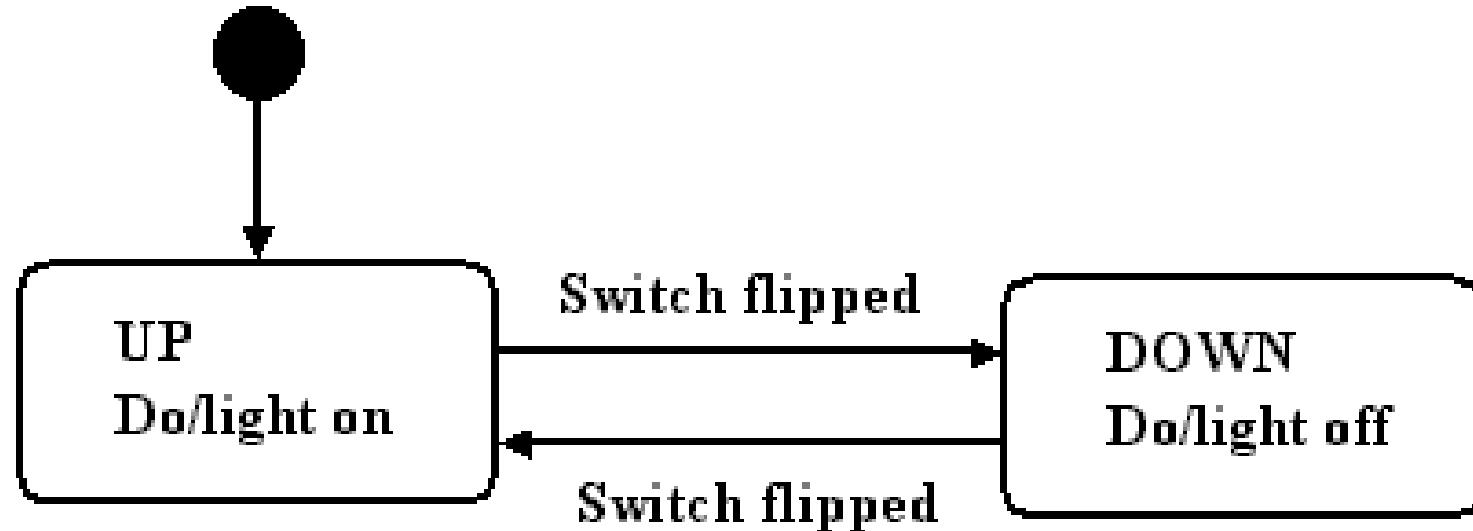


**Final State:** The end of the state diagram is shown by a bull's eye symbol, also called a final state.



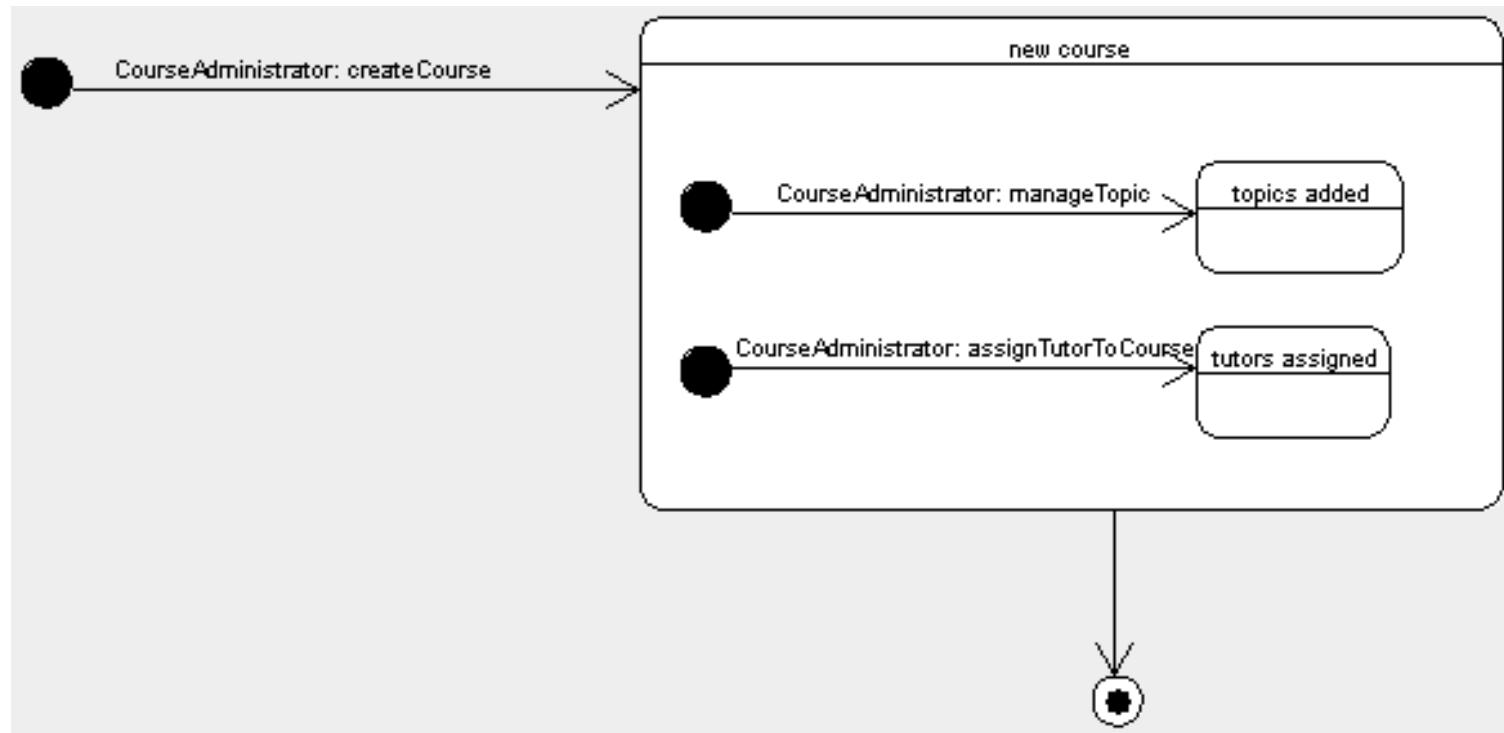
## Example: ordinary two-position light switch.

A light switch will have two states: up and down. (We could call them "on" and "off" if we liked.) In a UML state diagram, each state is represented by a rounded rectangle.



## Example: Identifying states and events of the Course object

- The events that occur in the lifecycle of the Course object are listed below:
  - Create new course—add information for the course
  - Add topics—add topics to the course
  - Assign tutors—assign the available tutors for the course
  - Close—finished adding or updating the course
- Assume that the admin. Is responsible for adding new course

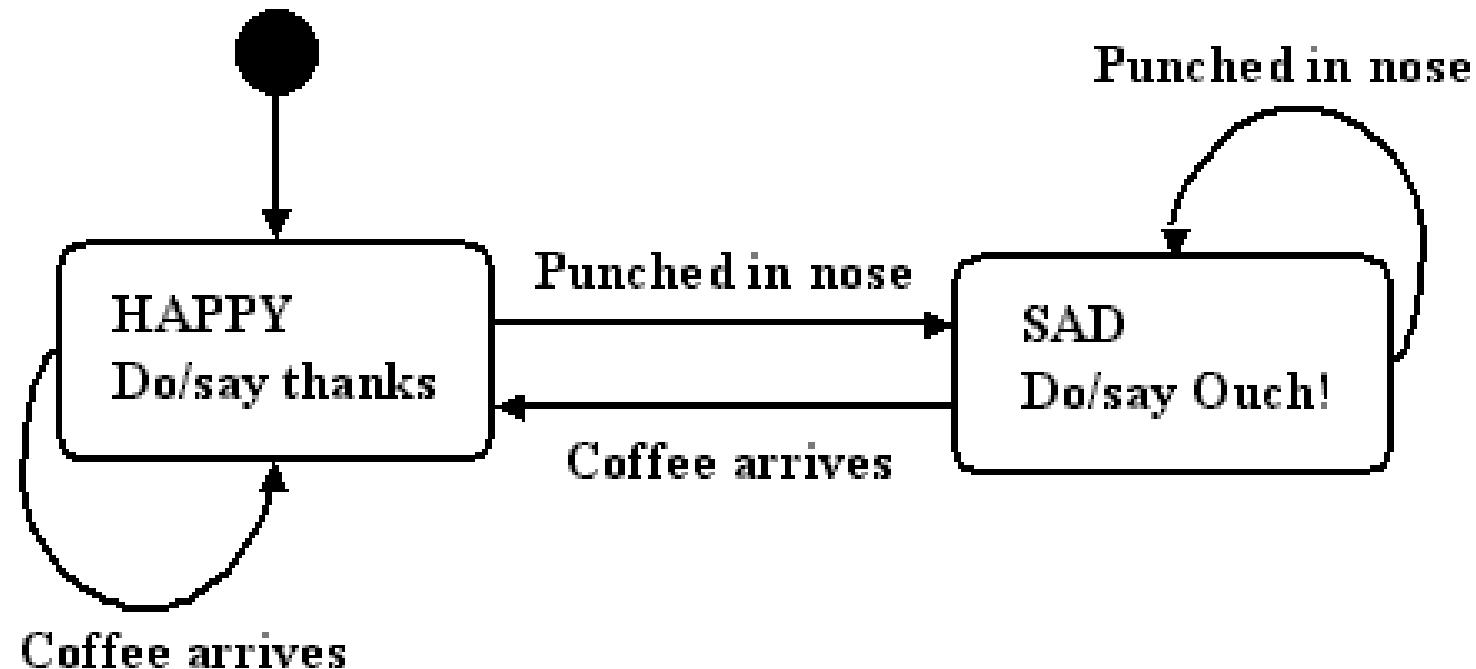


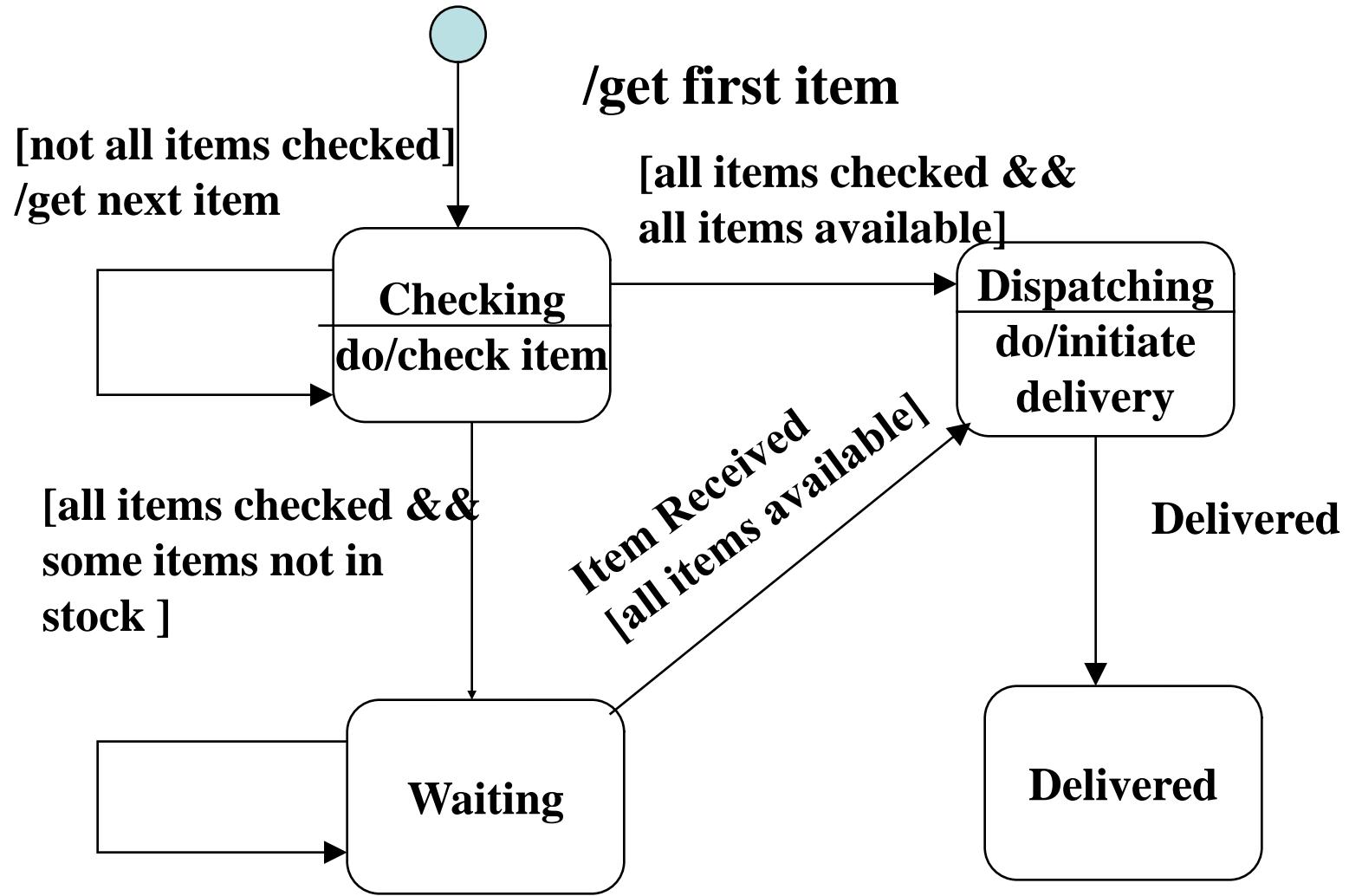
## Assignment No 3: Example: simplistic Teaching Assistant (TA)

TA only has two states:

- Happy when getting coffee.
- Sad when getting punched in the nose.

Suppose that TA is basically cheerful and starts out happy





**Item received**  
[some items not in stock]

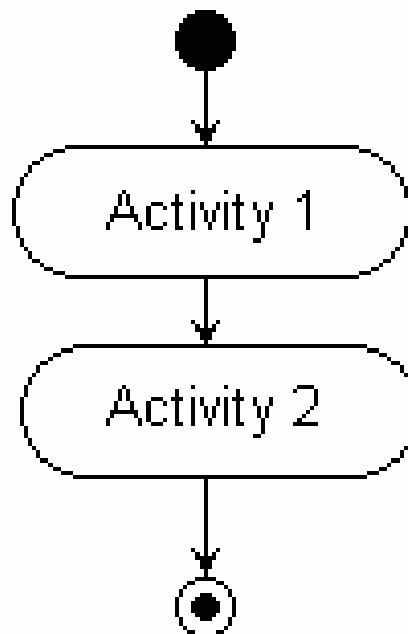
**Assignment No 4: Draw a Event State diagram for Landline phone**

**Assignment No 5: Draw a Event State diagram of a car using Ignition, Transmission, Brake and Accelerator**

# Activity Diagram

- The easiest way to visualize an Activity diagram is to think of a flowchart of a code.
- The flowchart is used to depict the business logic flow and the events that cause decisions and actions in the code to take place.
- An Activity diagram is a dynamic diagram that shows the activity and the event that causes the object to be in the particular state.
- The activity diagram is an extension of the state diagram. State diagrams highlight states and represent activities as arrows between states. Activity diagrams put the spotlight on the activities
- The Activity Diagrams are often used to model the paths through a use case. And to document the logic of a single use case.

- Each activity is represented by a rounded rectangle - narrower and more oval-shaped than the state icon
- The processing within an activity goes to completion and then an automatic transmission to the next activity occurs
- An arrow represents the transition from one activity to the next. Also an activity diagram has a starting point represented by filled-in circle, and endpoint represented by a bull's eye.



# Elements of an Activity diagram

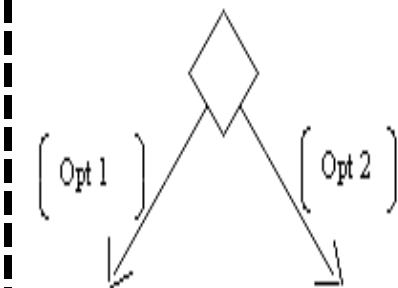
**Initial Activity:** This shows the starting point or first activity of the flow



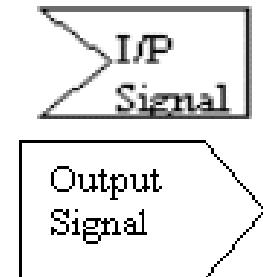
**Activity:** Represented by a rectangle with rounded (almost oval) edges.



**Decisions:** Similar to flowcharts, a logic where a decision is to be made is depicted by a diamond, with the options written on either sides of the arrows emerging from the diamond



**Signal:** When an activity sends or receives a message, that activity is called a signal. Signals are of two types: Input signal and Output signal

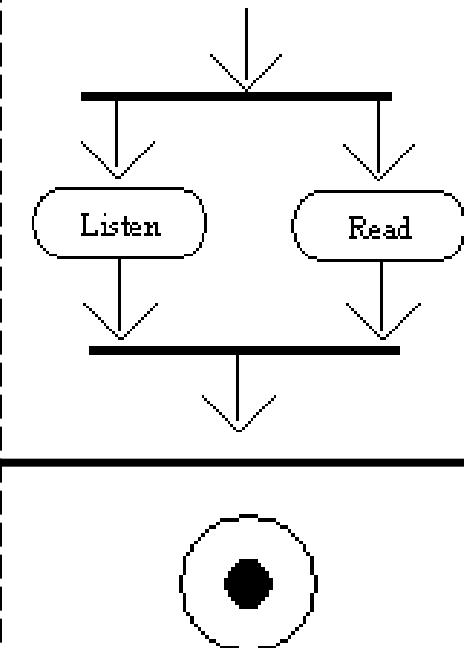


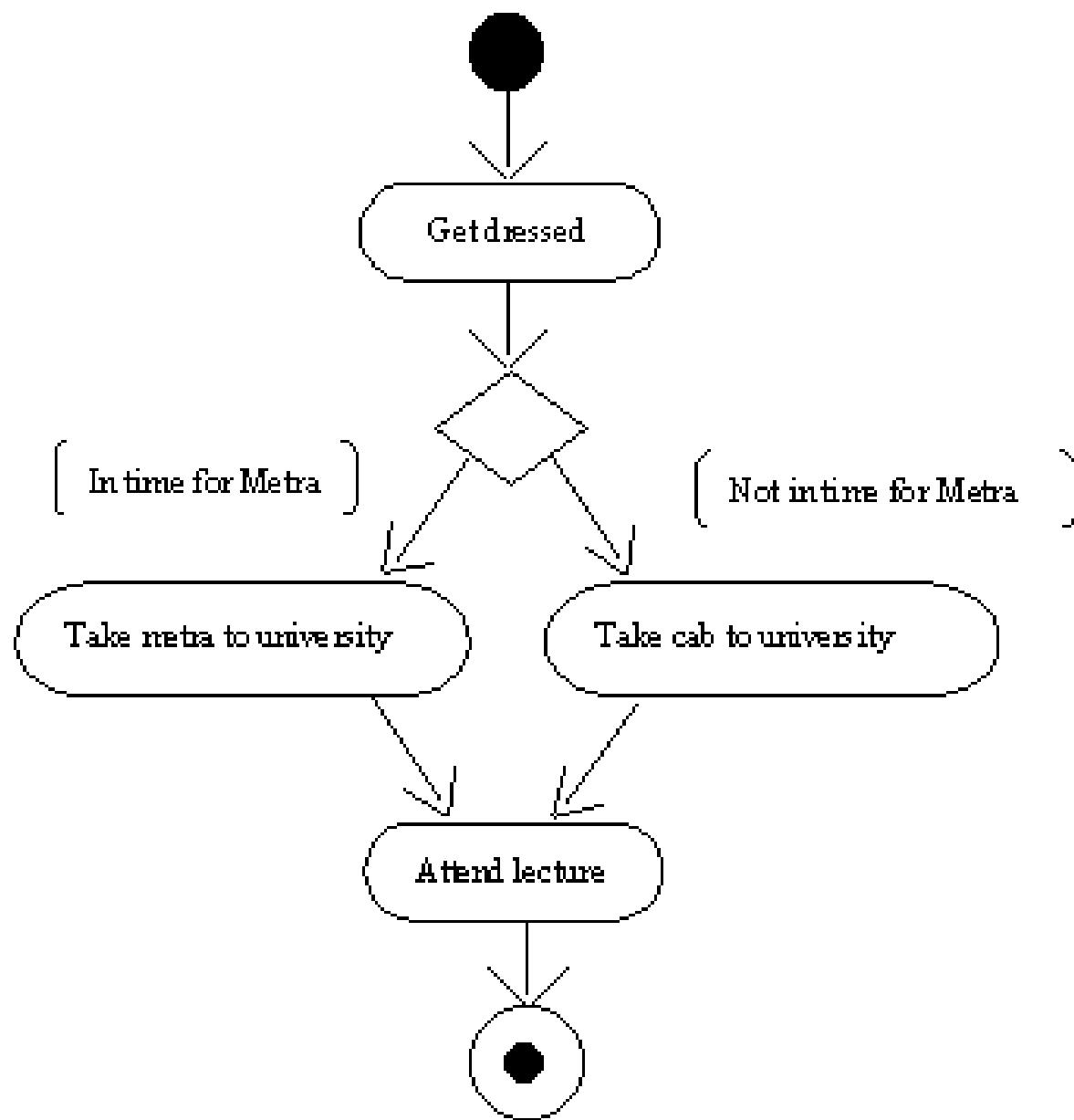
**Concurrent Activities:** Some activities occur simultaneously or in parallel. Such activities are called concurrent activities. For example, listening to the lecturer and looking at the blackboard is a parallel activity.

**Final Activity:** The end of the Activity diagram is shown by a bull's eye symbol, also called as a final activity.

- An activity diagram may have only one initial action state, but may have any number of final action states.

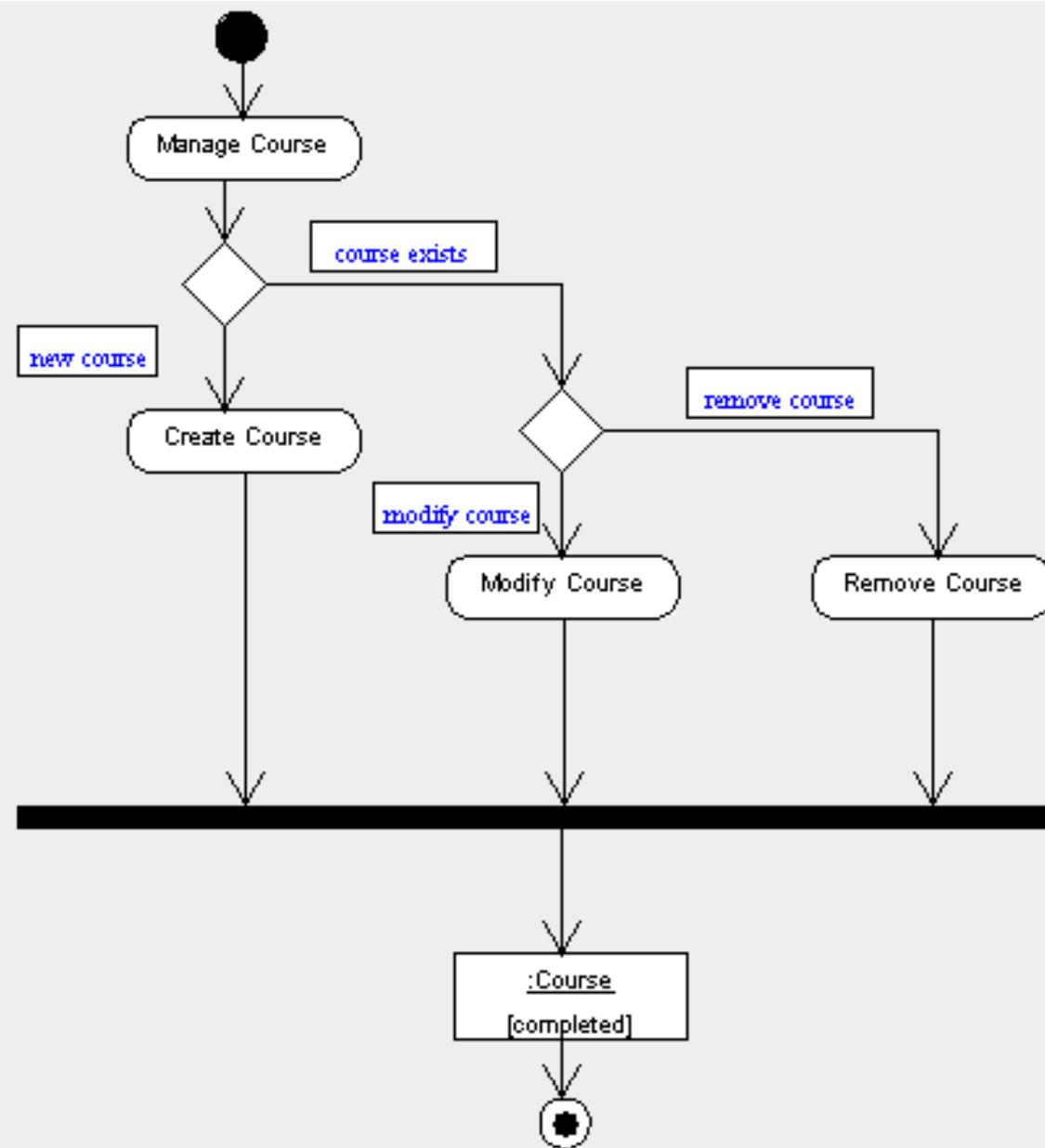
Consider the example of attending a course lecture, at 8 am.



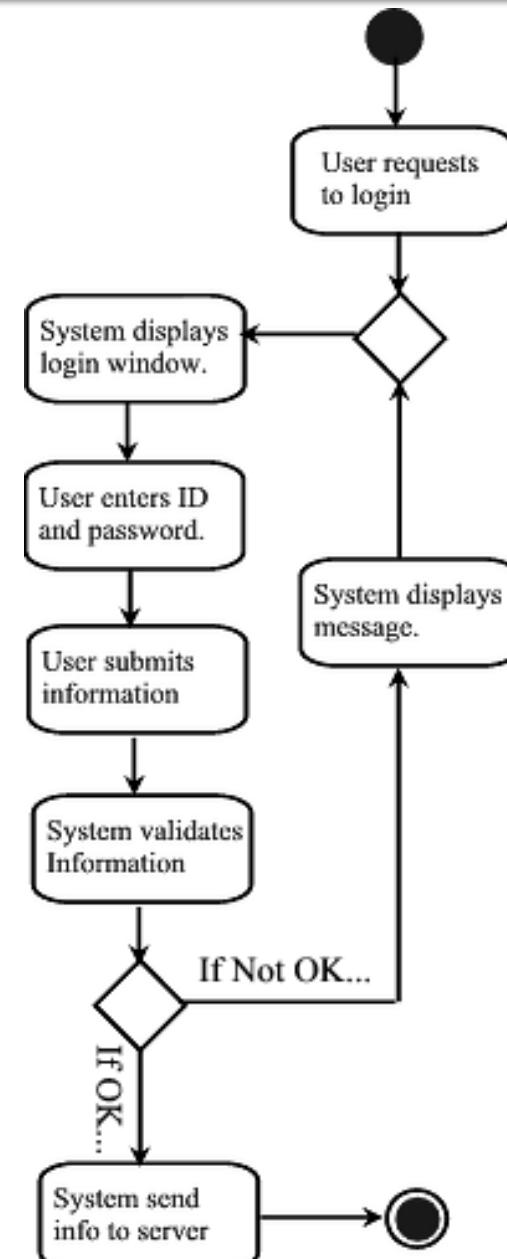


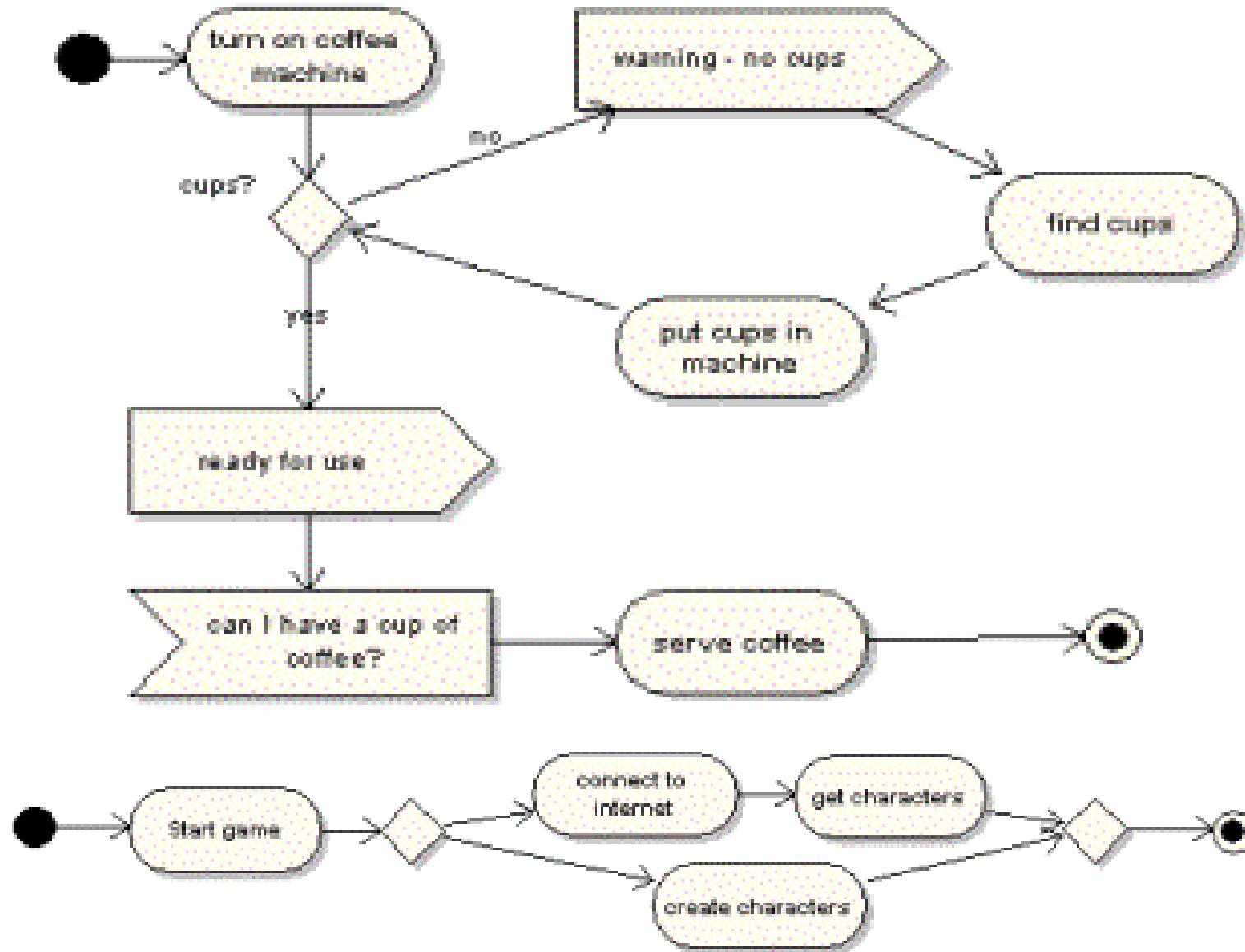
The course administrator is responsible for managing course information in a training center system, the course administrator carries out the following activities:

- Check if course exists
- If course is new, proceed to the "Create Course" step
- If course exists, check what operation is desired—whether to modify the course or remove the course
- If the modify course operation is selected by the course administrator, the "Modify Course" activity is performed
- If the remove course operation is selected by the course administrator, the "Remove Course" activity is performed



The following example shows the activity diagram for a login page.





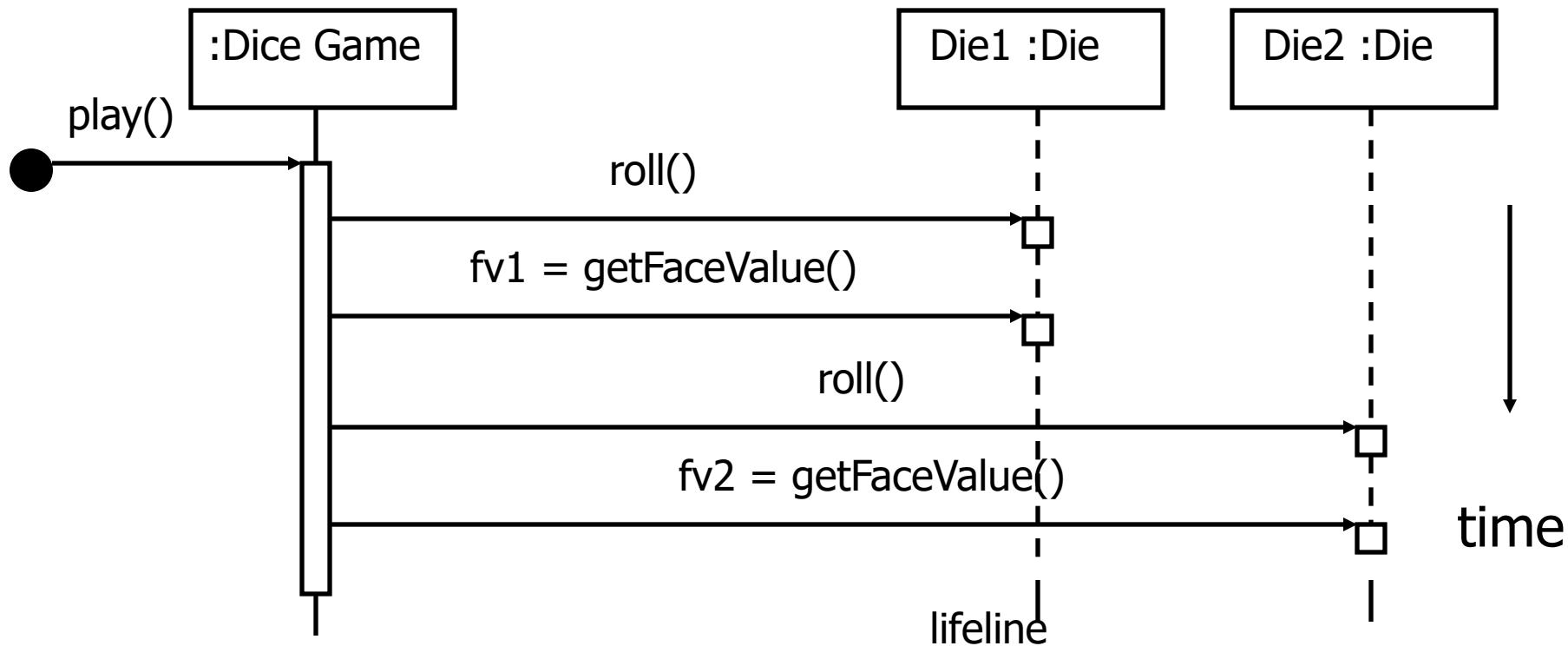
# Assignment No. 6: Draw a Activity diagram of processing orders

# Sequence Diagram in UML

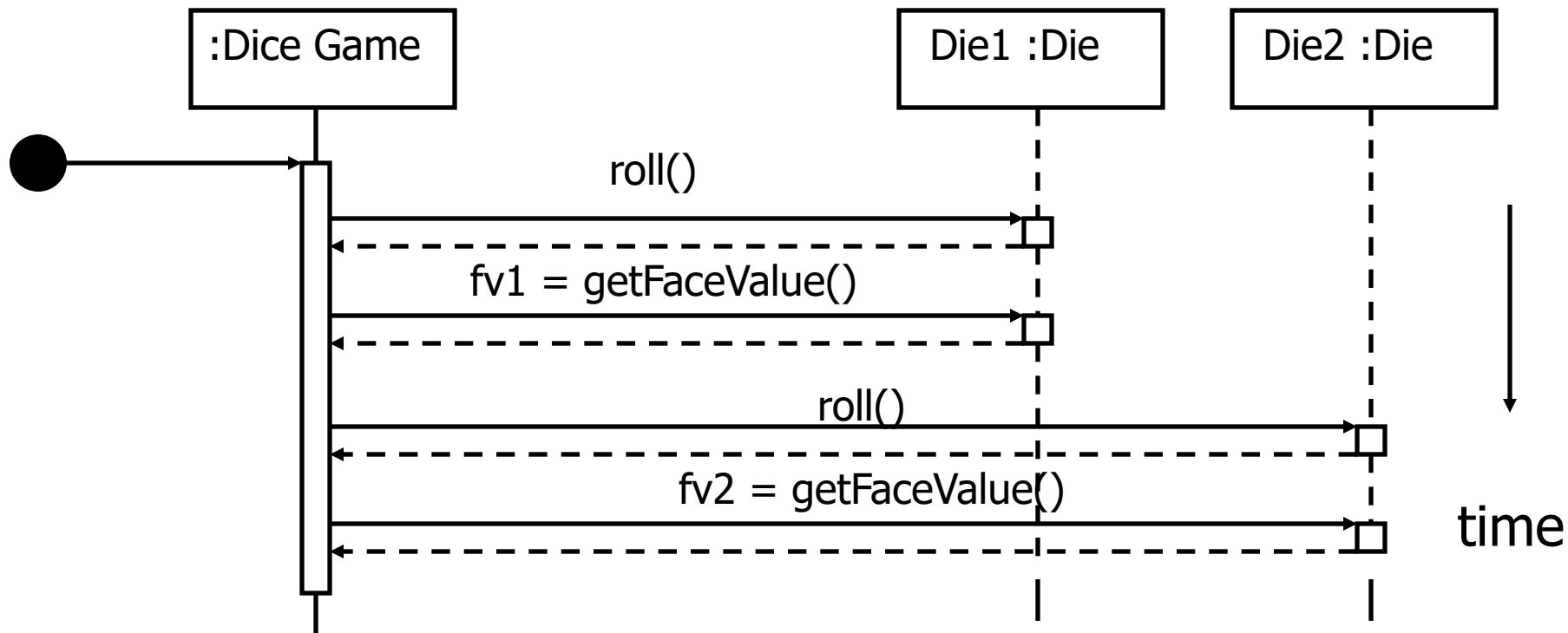
- A sequence diagram captures the behavior of a single scenario. The diagram shows a number of example objects and the messages that are passed between these objects within the use case.
- A Sequence diagram depicts the sequence of actions that occur in a system.
- The invocation of methods in each object, and the order in which the invocation occurs is captured in a Sequence diagram.
- A Sequence diagram is two-dimensional in nature. On the horizontal axis, it shows the life of the object that it represents, while on the vertical axis, it shows the sequence of the creation or invocation of these objects.

# Defining a sequence Diagram

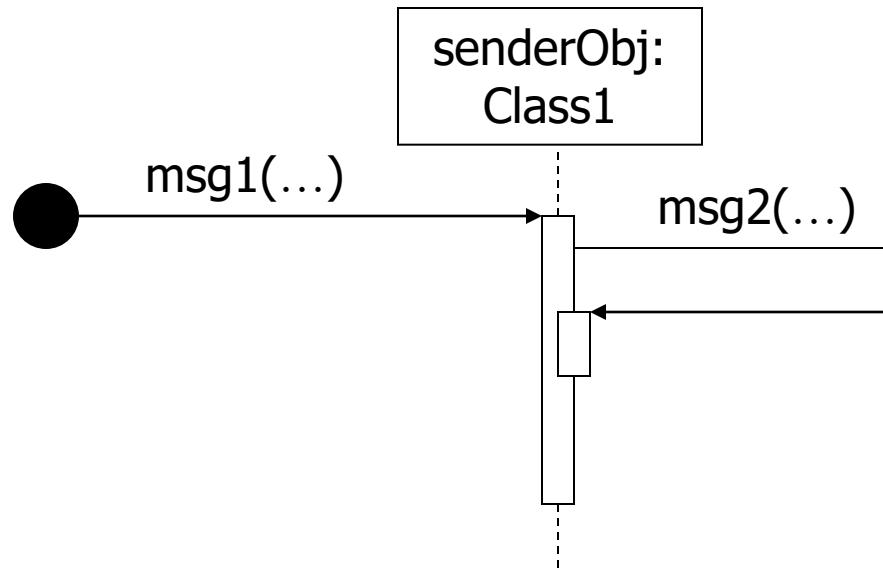
- A sequence diagram is made up of objects and messages. Objects are represented as rectangles with the underlined class name within the rectangle.

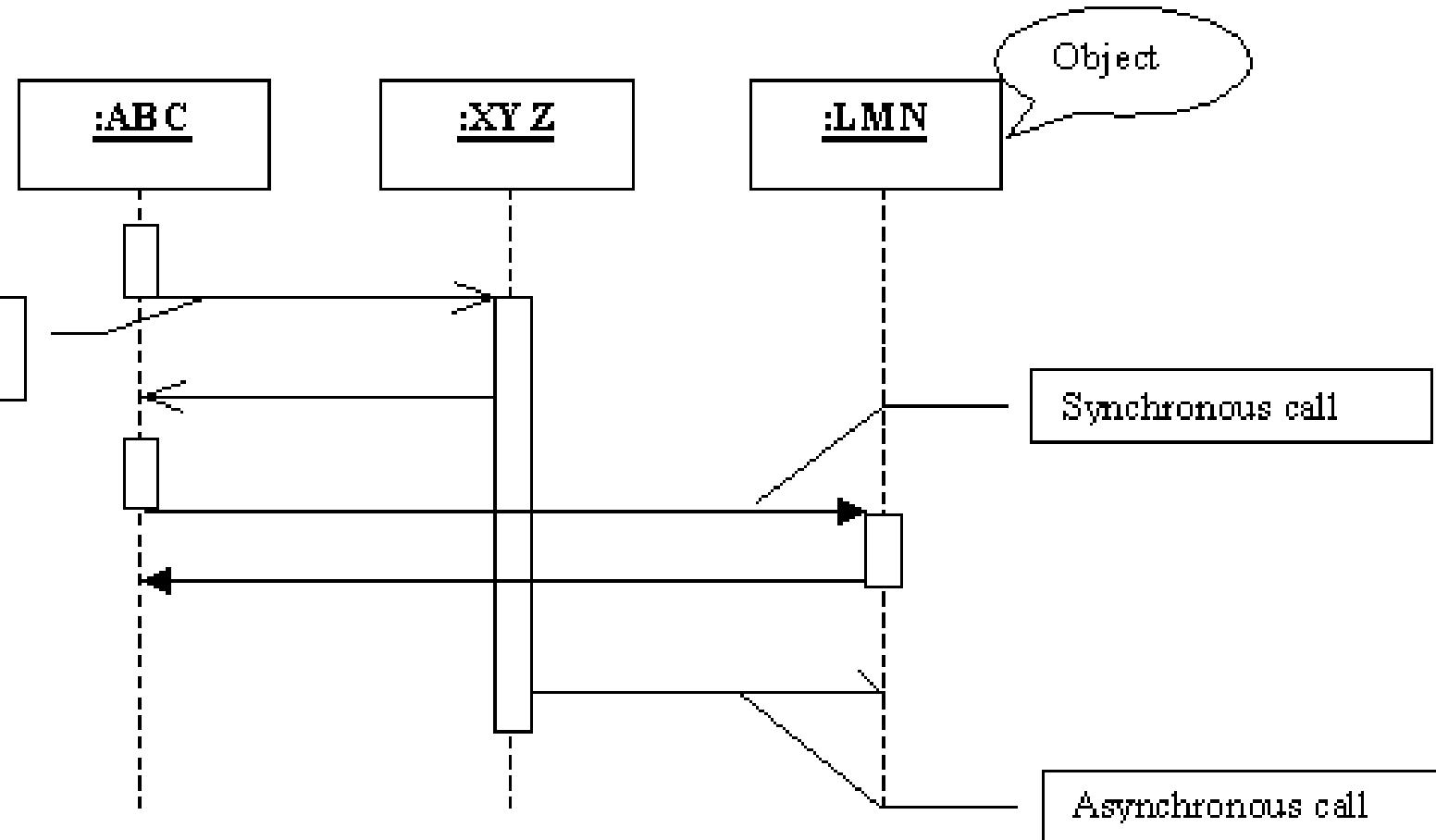


## Illustrating the return:



## Messages to Self:



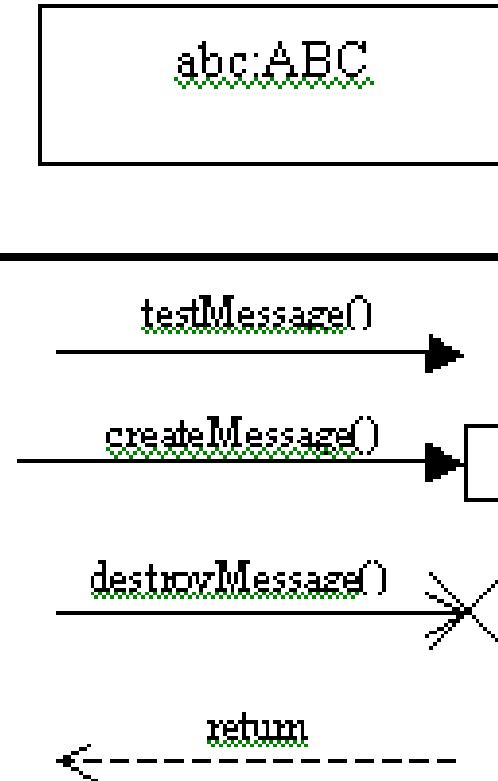


# Elements of a Sequence Diagram

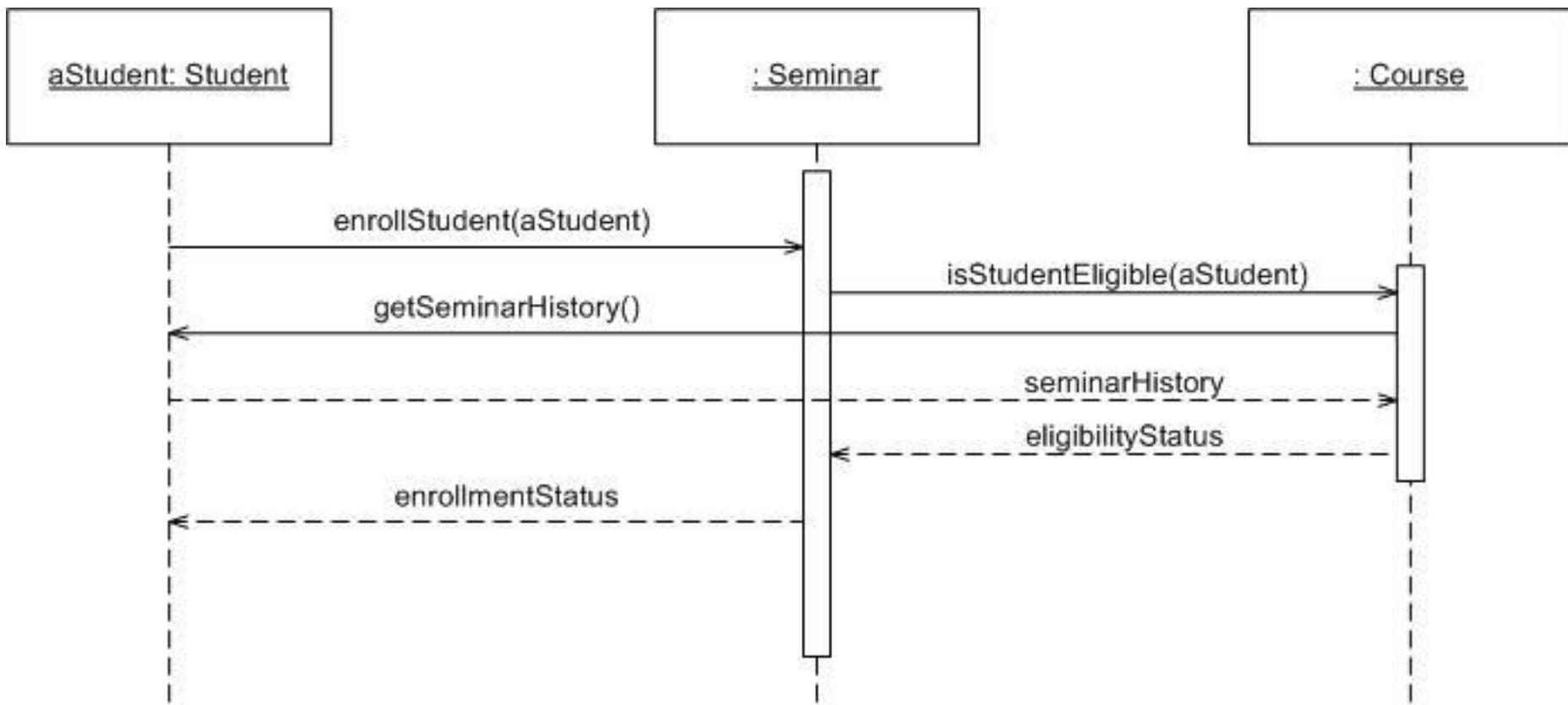
A Sequence diagram consists of the following behavioral elements:

**Object:** The primary element involved in a sequence diagram is an Object. A Sequence diagram consists of sequences of interaction among different objects over a period of time.

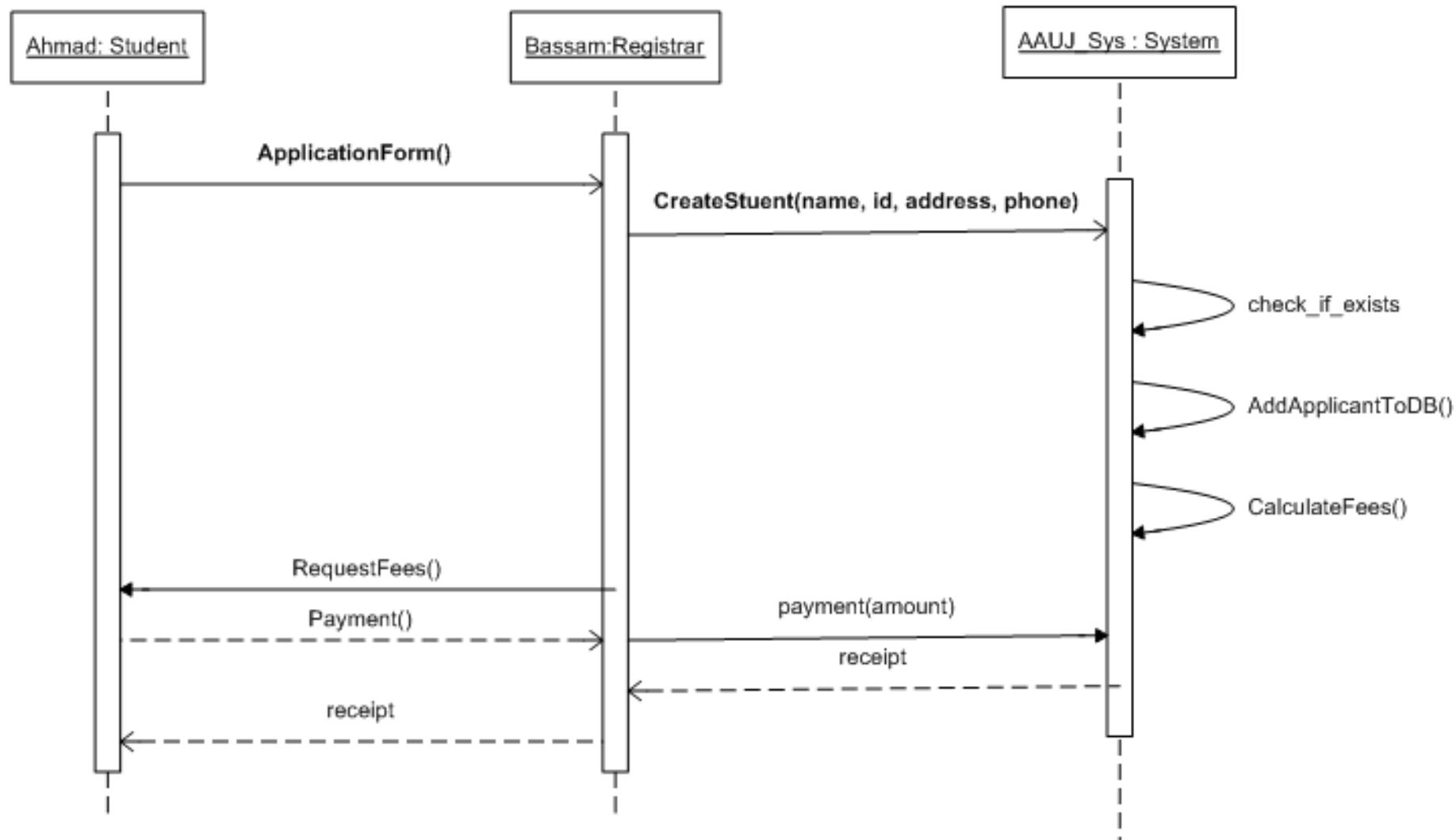
**Message:** The interaction between different objects in a sequence diagram is represented as messages. A message is represented by a directed arrow.



The following example shows the logic of how to enroll in a seminar.

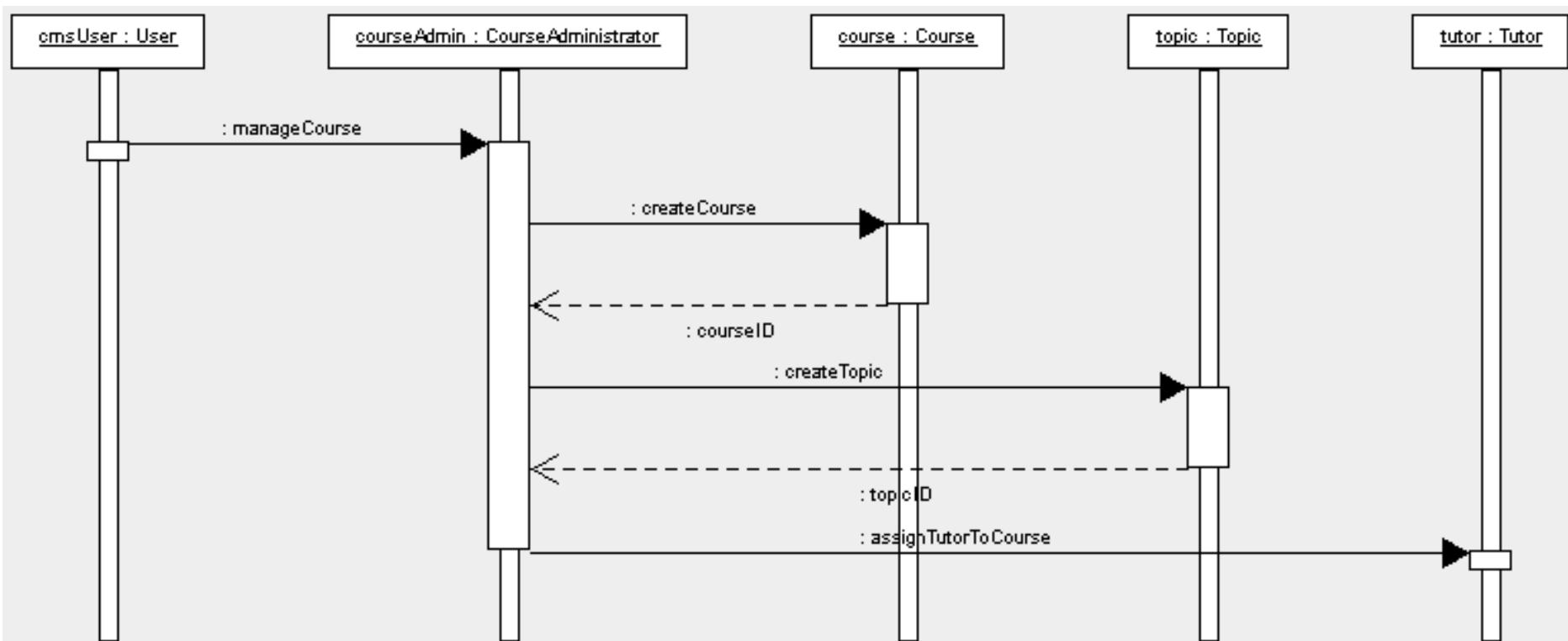


# Assignment No. 7: Sequence Diagram that for the Enroll in University Use Case



## **Identifying the activities and transitions for managing course information**

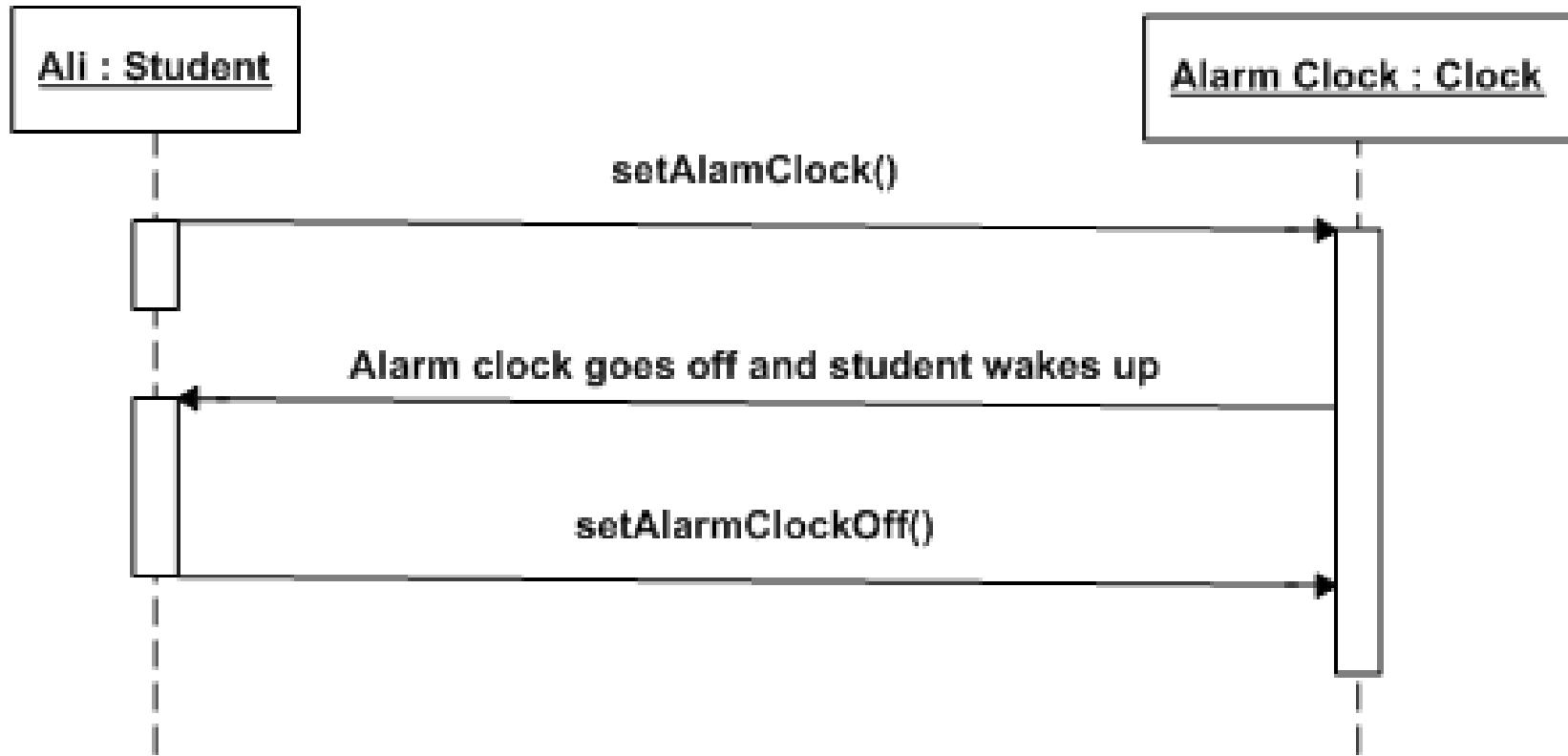
- A user who is a course administrator invokes the manage course functionality.
- The manage course functionality of the course administrator invokes either the course creation or course modification functionality of a course.
- After the course is either created or modified, the manage topic functionality of the course administrator calls the topic creation or modification functionality of a topic.
- Finally, the user invokes the assign tutor to course functionality of the course administrator to assign a tutor to the selected course.



# Assignment No. 8: Sequence Diagram that for the Enroll in University Use Case

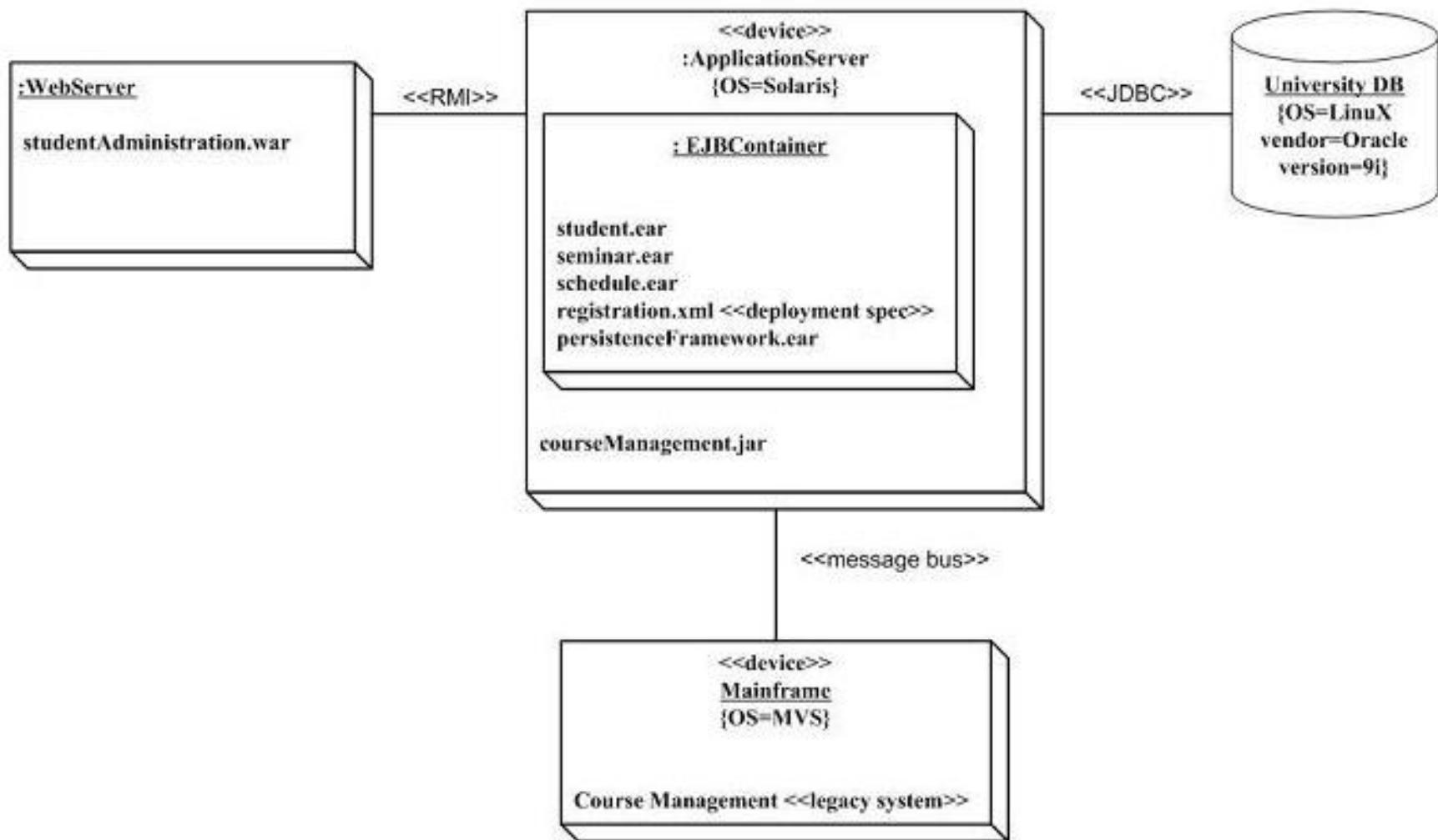
**Draw the sequence diagram for the following scenario**

- Student sets electric alarm clock to wake up time.
- Alarm clock goes off and student wakes up.
- Student sets alarm clock to off.

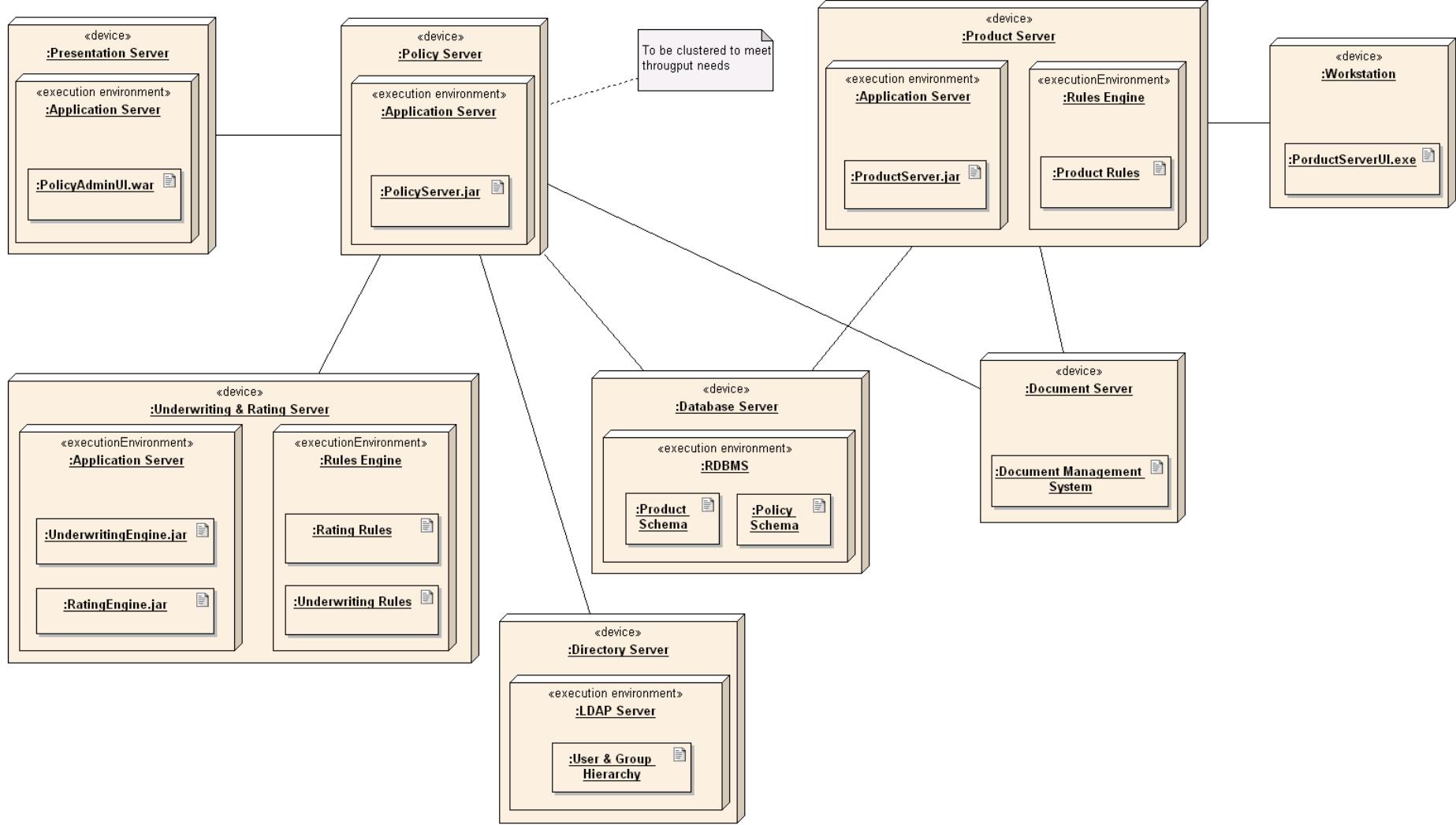


## Deployment Diagrams

- A **deployment diagram** in the [Unified Modeling Language](#) models the *physical* deployment of [artifacts](#) on [nodes](#).<sup>[1]</sup> To describe a web site, for example, a deployment diagram would show what hardware components ("nodes") exist (e.g., a web server, an application server, and a database server), what software components ("artifacts") run on each node (e.g., web application, database), and how the different pieces are connected (e.g. JDBC, REST, RMI).
- The nodes appear as boxes, and the artifacts allocated to each node appear as rectangles within the boxes. Nodes may have subnodes, which appear as nested boxes. A single node in a deployment diagram may conceptually represent multiple physical nodes, such as a cluster of database servers.
- There are two types of Nodes:
  - Device Node
  - Execution Environment Node
- Device nodes are physical computing resources with processing memory and services to execute software, such as typical computers or mobile phones. An execution environment node (EEN) is a software computing resource that runs within an outer node and which itself provides a service to host and execute other executable software elements.



## dd Deployment of Components



# Chapter 2 – Software Processes

# Topics covered

---

- ✧ Software process models
- ✧ Process activities
- ✧ Coping with change
- ✧ Process improvement

# The software process

---

- ✧ A structured set of activities required to develop a software system.
- ✧ Many different software processes but all involve:
  - Specification – defining what the system should do;
  - Design and implementation – defining the organization of the system and implementing the system;
  - Validation – checking that it does what the customer wants;
  - Evolution – changing the system in response to changing customer needs.
- ✧ A software process model is an abstract representation of a process. It presents a description of a process from some particular perspective.

# Software process descriptions

---

- ✧ When we describe and discuss processes, we usually talk about the activities in these processes such as specifying a data model, designing a user interface, etc. and the ordering of these activities.
- ✧ Process descriptions may also include:
  - Products, which are the outcomes of a process activity;
  - Roles, which reflect the responsibilities of the people involved in the process;
  - Pre- and post-conditions, which are statements that are true before and after a process activity has been enacted or a product produced.

# Plan-driven and agile processes

---

- ✧ Plan-driven processes are processes where all of the process activities are planned in advance and progress is measured against this plan.
- ✧ In agile processes, planning is incremental and it is easier to change the process to reflect changing customer requirements.
- ✧ In practice, most practical processes include elements of both plan-driven and agile approaches.
- ✧ There are no right or wrong software processes.

# Software process models

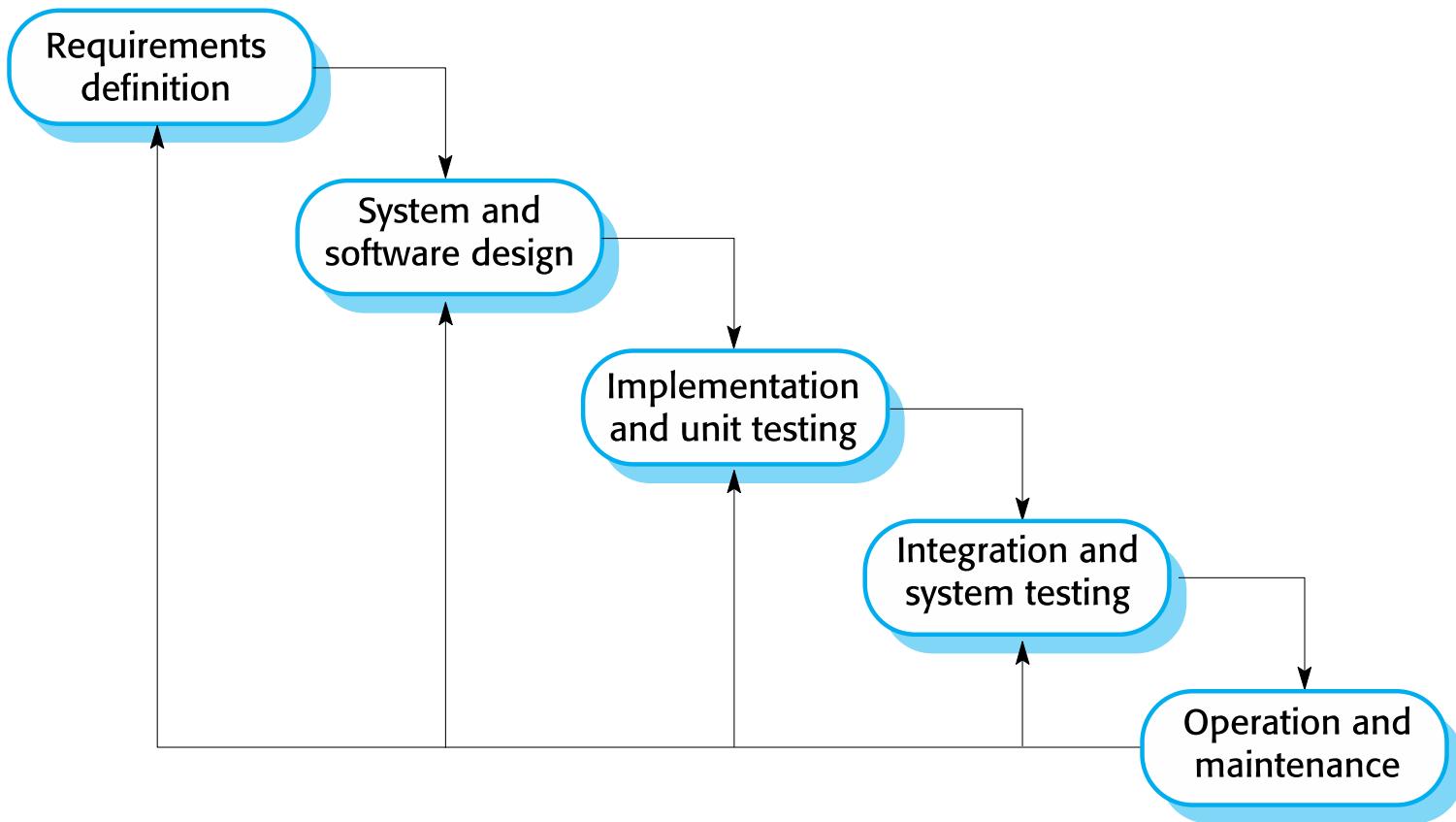
# Software process models

---

- ✧ The waterfall model
  - Plan-driven model. Separate and distinct phases of specification and development.
- ✧ Incremental development
  - Specification, development and validation are interleaved. May be plan-driven or agile.
- ✧ Integration and configuration
  - The system is assembled from existing configurable components. May be plan-driven or agile.
- ✧ In practice, most large systems are developed using a process that incorporates elements from all of these models.

# The waterfall model

---



# Waterfall model phases

---

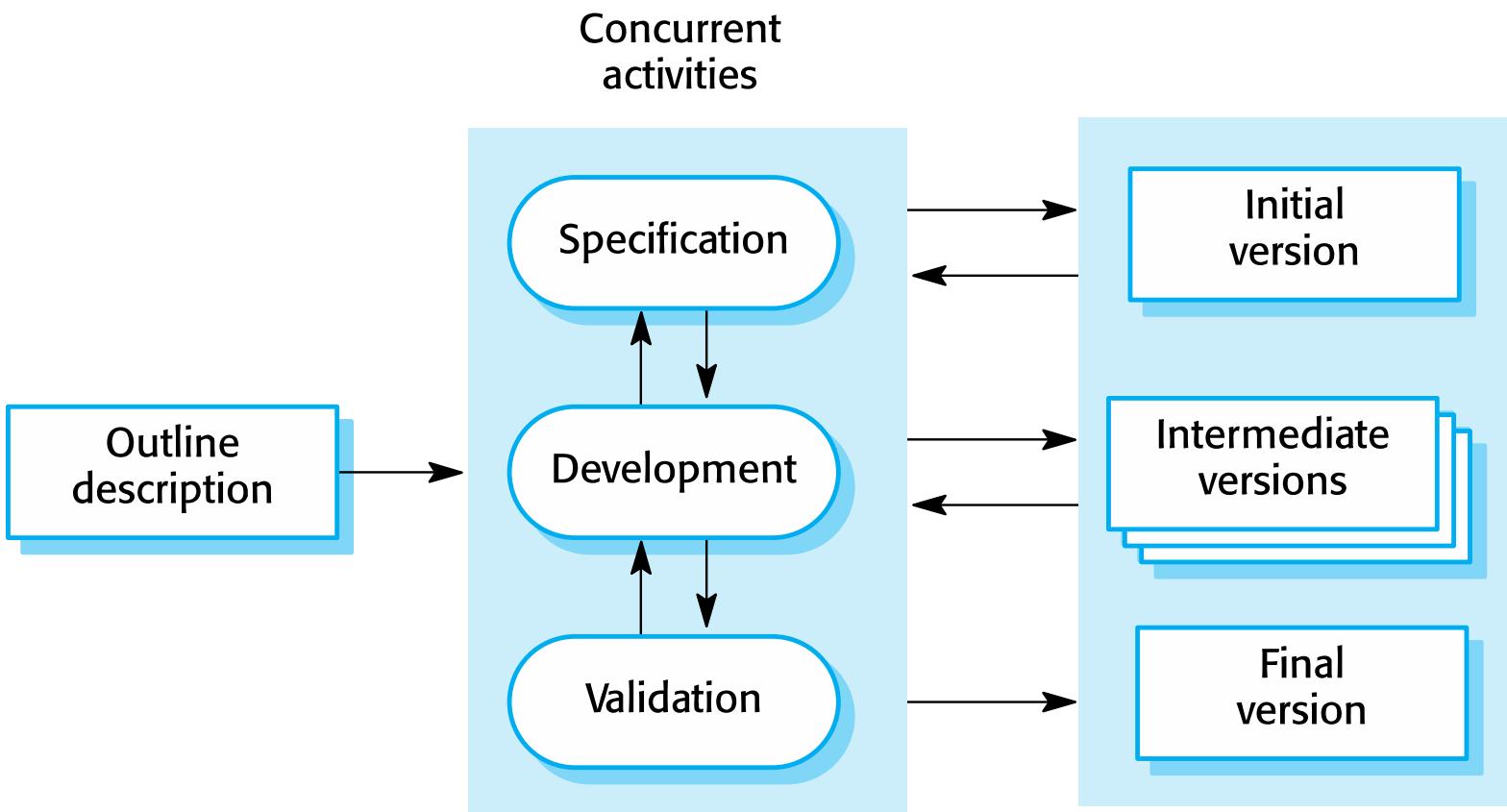
- ✧ There are separate identified phases in the waterfall model:
  - Requirements analysis and definition
  - System and software design
  - Implementation and unit testing
  - Integration and system testing
  - Operation and maintenance
- ✧ The main drawback of the waterfall model is the difficulty of accommodating change after the process is underway. In principle, a phase has to be complete before moving onto the next phase.

# Waterfall model problems

---

- ✧ Inflexible partitioning of the project into distinct stages makes it difficult to respond to changing customer requirements.
  - Therefore, this model is only appropriate when the requirements are well-understood and changes will be fairly limited during the design process.
  - Few business systems have stable requirements.
- ✧ The waterfall model is mostly used for large systems engineering projects where a system is developed at several sites.
  - In those circumstances, the plan-driven nature of the waterfall model helps coordinate the work.

# Incremental development



# Incremental development benefits

---

- ✧ The cost of accommodating changing customer requirements is reduced.
  - The amount of analysis and documentation that has to be redone is much less than is required with the waterfall model.
- ✧ It is easier to get customer feedback on the development work that has been done.
  - Customers can comment on demonstrations of the software and see how much has been implemented.
- ✧ More rapid delivery and deployment of useful software to the customer is possible.
  - Customers are able to use and gain value from the software earlier than is possible with a waterfall process.

# Incremental development problems

---

- ✧ The process is not visible.
  - Managers need regular deliverables to measure progress. If systems are developed quickly, it is not cost-effective to produce documents that reflect every version of the system.
- ✧ System structure tends to degrade as new increments are added.
  - Unless time and money is spent on refactoring to improve the software, regular change tends to corrupt its structure. Incorporating further software changes becomes increasingly difficult and costly.

# Integration and configuration

---

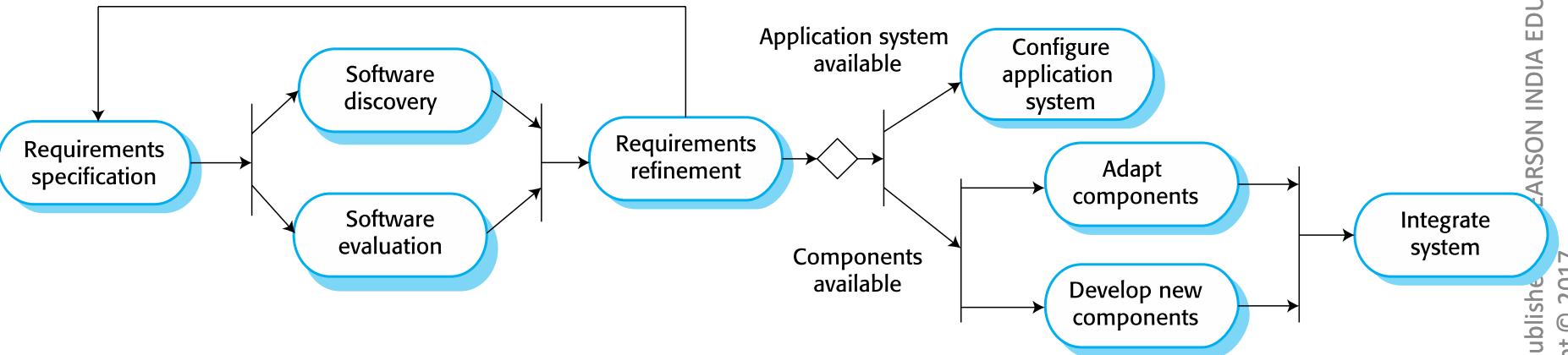
- ✧ Based on software reuse where systems are integrated from existing components or application systems (sometimes called COTS -Commercial-off-the-shelf) systems.
- ✧ Reused elements may be configured to adapt their behaviour and functionality to a user's requirements
- ✧ Reuse is now the standard approach for building many types of business system
  - Reuse covered in more depth in Chapter 15.

# Types of reusable software

---

- ✧ Stand-alone application systems (sometimes called COTS) that are configured for use in a particular environment.
- ✧ Collections of objects that are developed as a package to be integrated with a component framework such as .NET or J2EE.
- ✧ Web services that are developed according to service standards and which are available for remote invocation.

# Reuse-oriented software engineering



# Key process stages

---

- ✧ Requirements specification
- ✧ Software discovery and evaluation
- ✧ Requirements refinement
- ✧ Application system configuration
- ✧ Component adaptation and integration

# Advantages and disadvantages

---

- ✧ Reduced costs and risks as less software is developed from scratch
- ✧ Faster delivery and deployment of system
- ✧ But requirements compromises are inevitable so system may not meet real needs of users
- ✧ Loss of control over evolution of reused system elements

---

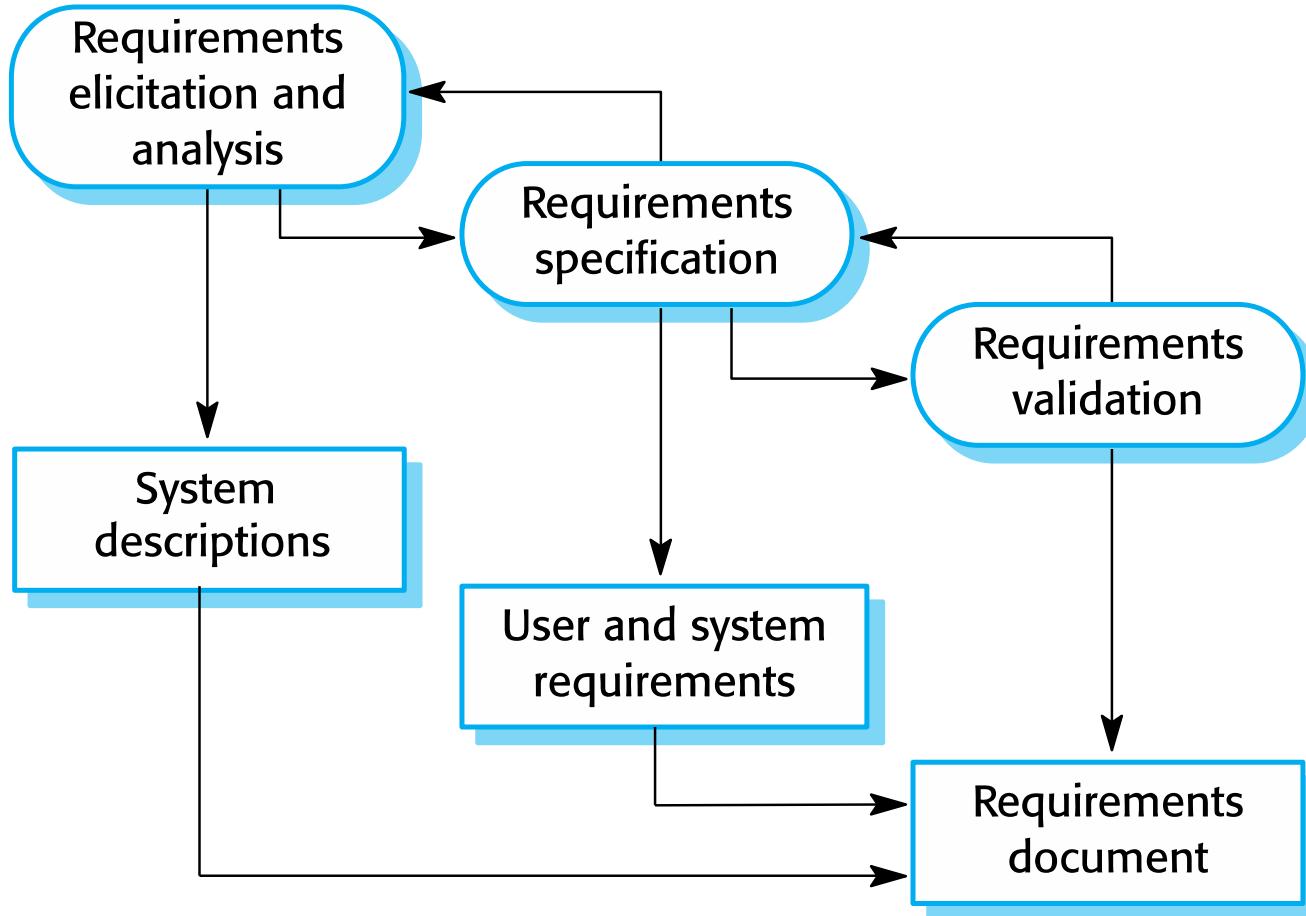
# **Process activities**

# Process activities

---

- ✧ Real software processes are inter-leaved sequences of technical, collaborative and managerial activities with the overall goal of specifying, designing, implementing and testing a software system.
- ✧ The four basic process activities of specification, development, validation and evolution are organized differently in different development processes.
- ✧ For example, in the waterfall model, they are organized in sequence, whereas in incremental development they are interleaved.

# The requirements engineering process



# Software specification

---

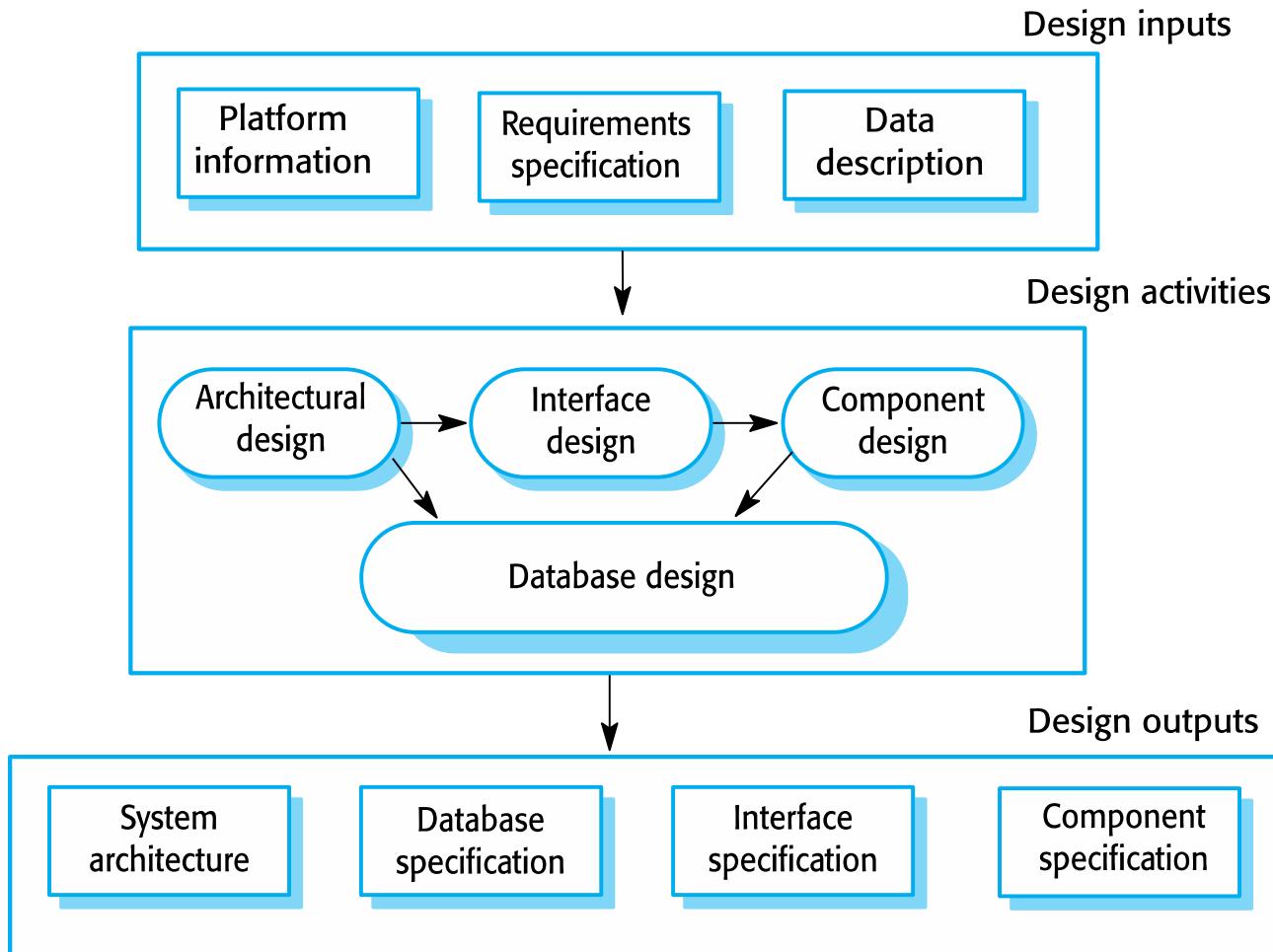
- ✧ The process of establishing what services are required and the constraints on the system's operation and development.
- ✧ Requirements engineering process
  - Requirements elicitation and analysis
    - What do the system stakeholders require or expect from the system?
  - Requirements specification
    - Defining the requirements in detail
  - Requirements validation
    - Checking the validity of the requirements

# Software design and implementation

---

- ✧ The process of converting the system specification into an executable system.
- ✧ Software design
  - Design a software structure that realises the specification;
- ✧ Implementation
  - Translate this structure into an executable program;
- ✧ The activities of design and implementation are closely related and may be inter-leaved.

# A general model of the design process



# Design activities

---

- ✧ *Architectural design*, where you identify the overall structure of the system, the principal components (subsystems or modules), their relationships and how they are distributed.
- ✧ *Database design*, where you design the system data structures and how these are to be represented in a database.
- ✧ *Interface design*, where you define the interfaces between system components.
- ✧ *Component selection and design*, where you search for reusable components. If unavailable, you design how it will operate.

# System implementation

---

- ✧ The software is implemented either by developing a program or programs or by configuring an application system.
- ✧ Design and implementation are interleaved activities for most types of software system.
- ✧ Programming is an individual activity with no standard process.
- ✧ Debugging is the activity of finding program faults and correcting these faults.

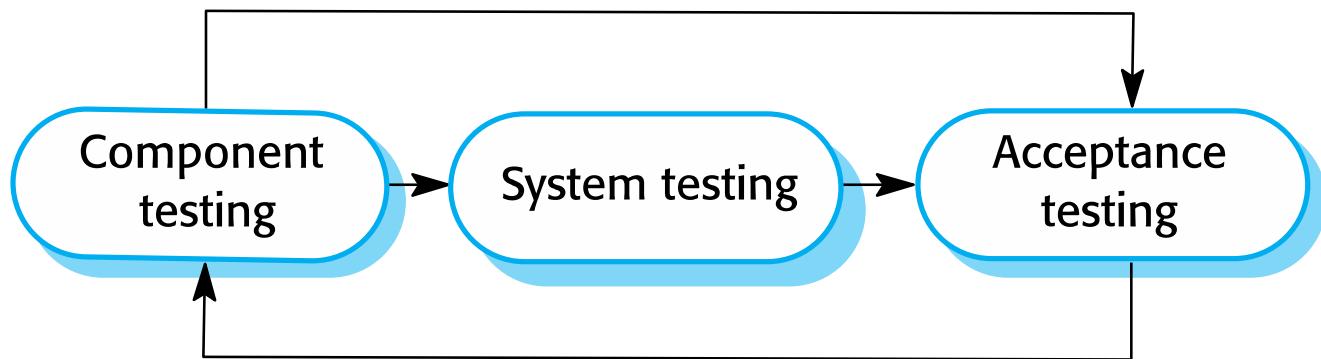
# Software validation

---

- ✧ Verification and validation (V & V) is intended to show that a system conforms to its specification and meets the requirements of the system customer.
- ✧ Involves checking and review processes and system testing.
- ✧ System testing involves executing the system with test cases that are derived from the specification of the real data to be processed by the system.
- ✧ Testing is the most commonly used V & V activity.

# Stages of testing

---



# Testing stages

---

## ✧ Component testing

- Individual components are tested independently;
- Components may be functions or objects or coherent groupings of these entities.

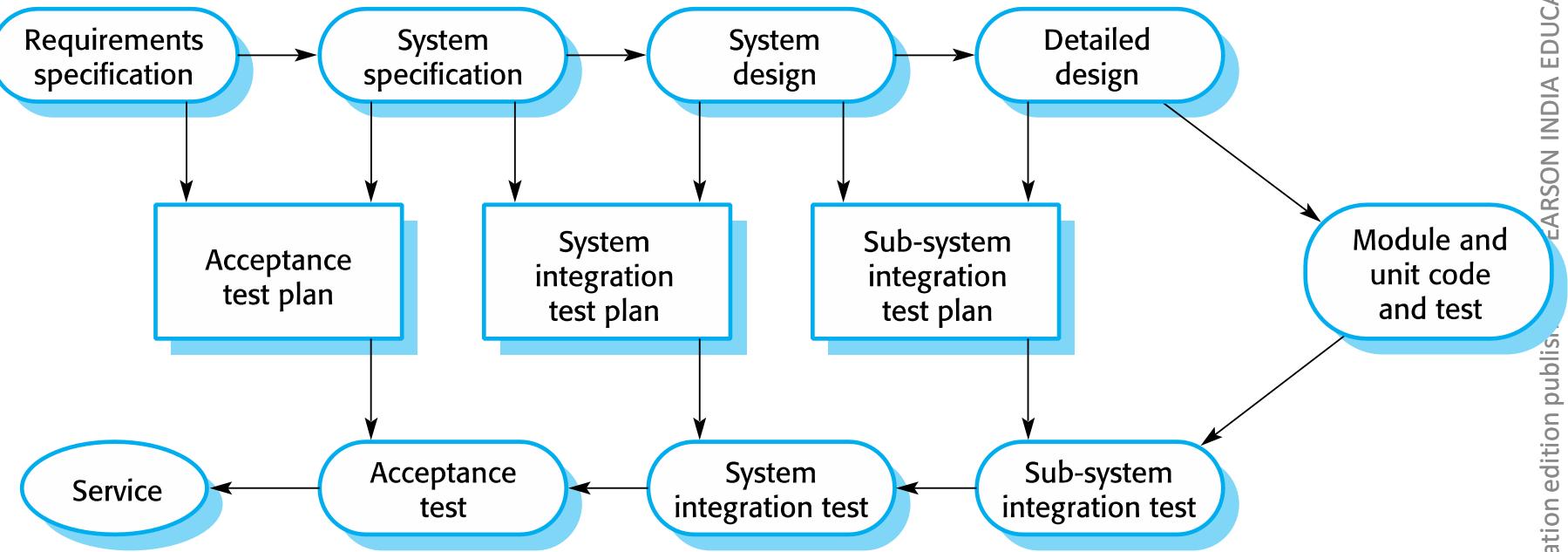
## ✧ System testing

- Testing of the system as a whole. Testing of emergent properties is particularly important.

## ✧ Customer testing

- Testing with customer data to check that the system meets the customer's needs.

# Testing phases in a plan-driven software process (V-model)



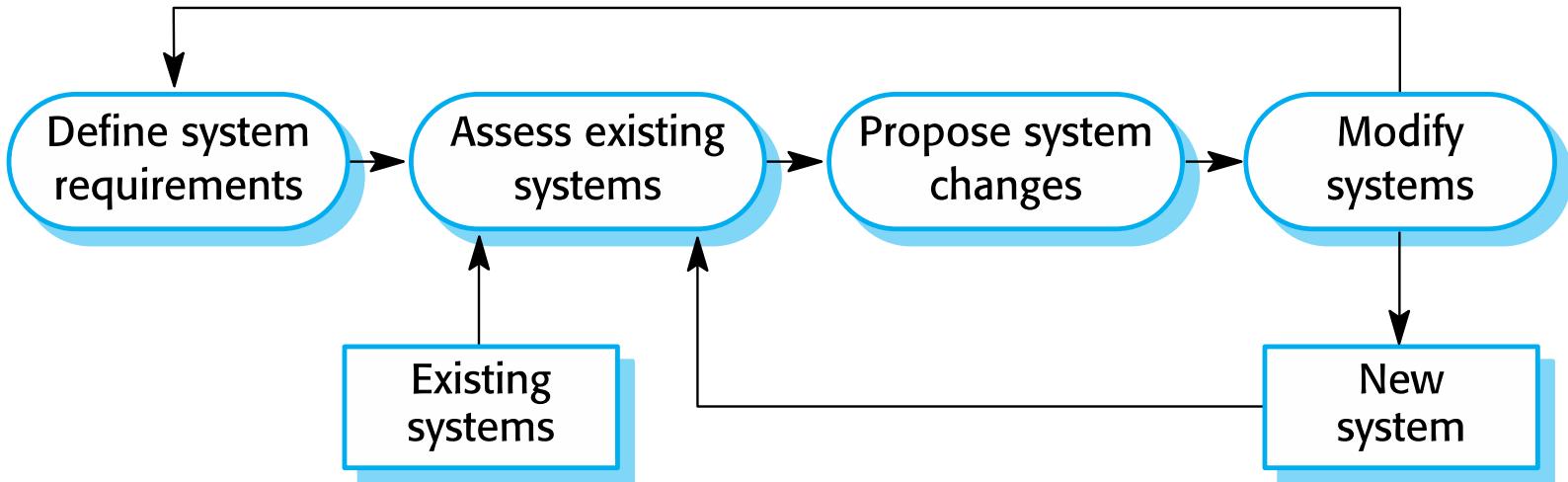
# Software evolution

---

- ✧ Software is inherently flexible and can change.
- ✧ As requirements change through changing business circumstances, the software that supports the business must also evolve and change.
- ✧ Although there has been a demarcation between development and evolution (maintenance) this is increasingly irrelevant as fewer and fewer systems are completely new.

# System evolution

---



# Coping with change

# Coping with change

---

- ✧ Change is inevitable in all large software projects.
  - Business changes lead to new and changed system requirements
  - New technologies open up new possibilities for improving implementations
  - Changing platforms require application changes
- ✧ Change leads to rework so the costs of change include both rework (e.g. re-analysing requirements) as well as the costs of implementing new functionality

# Reducing the costs of rework

---

- ✧ Change anticipation, where the software process includes activities that can anticipate possible changes before significant rework is required.
  - For example, a prototype system may be developed to show some key features of the system to customers.
- ✧ Change tolerance, where the process is designed so that changes can be accommodated at relatively low cost.
  - This normally involves some form of incremental development. Proposed changes may be implemented in increments that have not yet been developed. If this is impossible, then only a single increment (a small part of the system) may have been altered to incorporate the change.

# Coping with changing requirements

---

- ✧ System prototyping, where a version of the system or part of the system is developed quickly to check the customer's requirements and the feasibility of design decisions. This approach supports change anticipation.
- ✧ Incremental delivery, where system increments are delivered to the customer for comment and experimentation. This supports both change avoidance and change tolerance.

# Software prototyping

---

- ✧ A prototype is an initial version of a system used to demonstrate concepts and try out design options.
- ✧ A prototype can be used in:
  - The requirements engineering process to help with requirements elicitation and validation;
  - In design processes to explore options and develop a UI design;
  - In the testing process to run back-to-back tests.

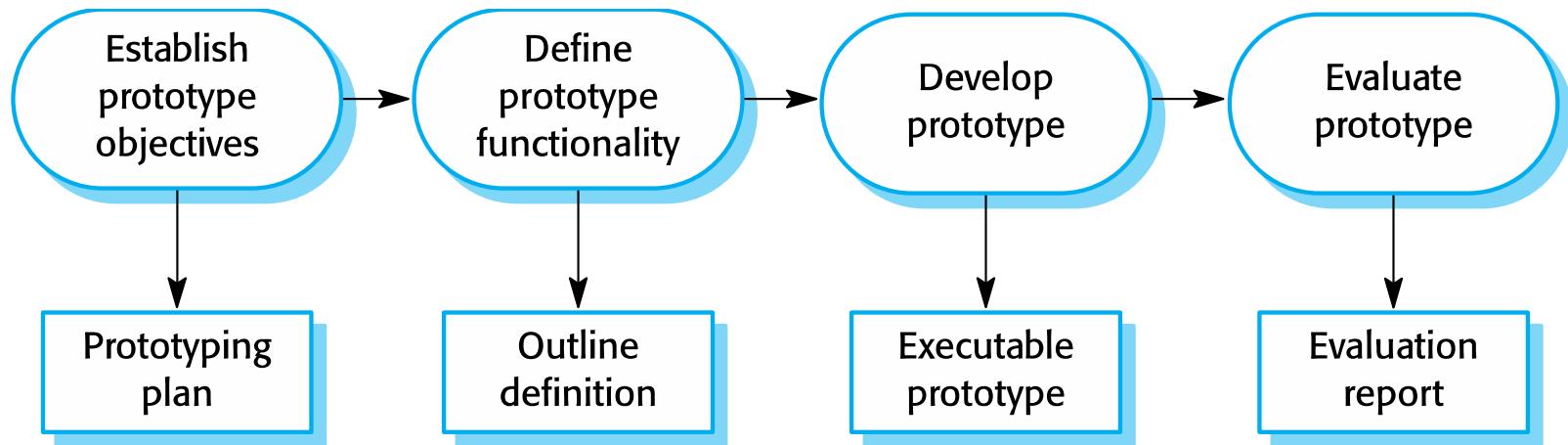
# Benefits of prototyping

---

- ✧ Improved system usability.
- ✧ A closer match to users' real needs.
- ✧ Improved design quality.
- ✧ Improved maintainability.
- ✧ Reduced development effort.

# The process of prototype development

---



# Prototype development

---

- ✧ May be based on rapid prototyping languages or tools
- ✧ May involve leaving out functionality
  - Prototype should focus on areas of the product that are not well-understood;
  - Error checking and recovery may not be included in the prototype;
  - Focus on functional rather than non-functional requirements such as reliability and security

# Throw-away prototypes

---

- ✧ Prototypes should be discarded after development as they are not a good basis for a production system:
  - It may be impossible to tune the system to meet non-functional requirements;
  - Prototypes are normally undocumented;
  - The prototype structure is usually degraded through rapid change;
  - The prototype probably will not meet normal organisational quality standards.

# Incremental delivery

---

- ✧ Rather than deliver the system as a single delivery, the development and delivery is broken down into increments with each increment delivering part of the required functionality.
- ✧ User requirements are prioritised and the highest priority requirements are included in early increments.
- ✧ Once the development of an increment is started, the requirements are frozen though requirements for later increments can continue to evolve.

# Incremental development and delivery

---

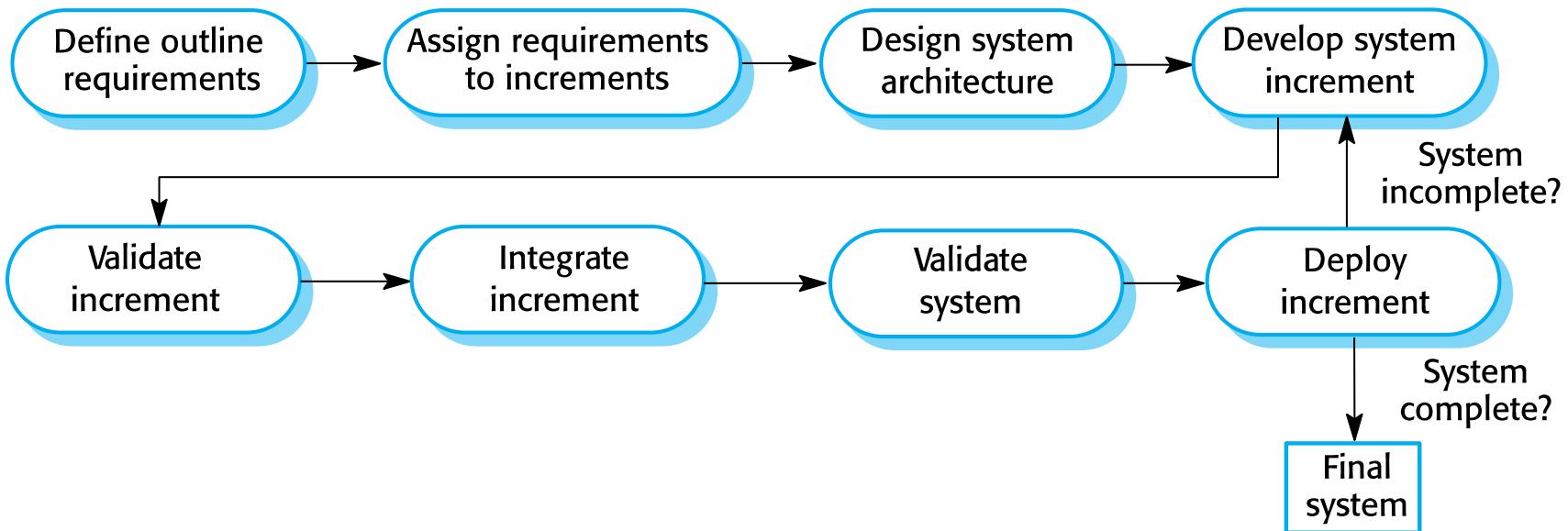
## ✧ Incremental development

- Develop the system in increments and evaluate each increment before proceeding to the development of the next increment;
- Normal approach used in agile methods;
- Evaluation done by user/customer proxy.

## ✧ Incremental delivery

- Deploy an increment for use by end-users;
- More realistic evaluation about practical use of software;
- Difficult to implement for replacement systems as increments have less functionality than the system being replaced.

# Incremental delivery



# Incremental delivery advantages

---

- ✧ Customer value can be delivered with each increment so system functionality is available earlier.
- ✧ Early increments act as a prototype to help elicit requirements for later increments.
- ✧ Lower risk of overall project failure.
- ✧ The highest priority system services tend to receive the most testing.

# Incremental delivery problems

---

- ✧ Most systems require a set of basic facilities that are used by different parts of the system.
  - As requirements are not defined in detail until an increment is to be implemented, it can be hard to identify common facilities that are needed by all increments.
- ✧ The essence of iterative processes is that the specification is developed in conjunction with the software.
  - However, this conflicts with the procurement model of many organizations, where the complete system specification is part of the system development contract.

# Process improvement

# Process improvement

---

- ✧ Many software companies have turned to software process improvement as a way of enhancing the quality of their software, reducing costs or accelerating their development processes.
- ✧ Process improvement means understanding existing processes and changing these processes to increase product quality and/or reduce costs and development time.

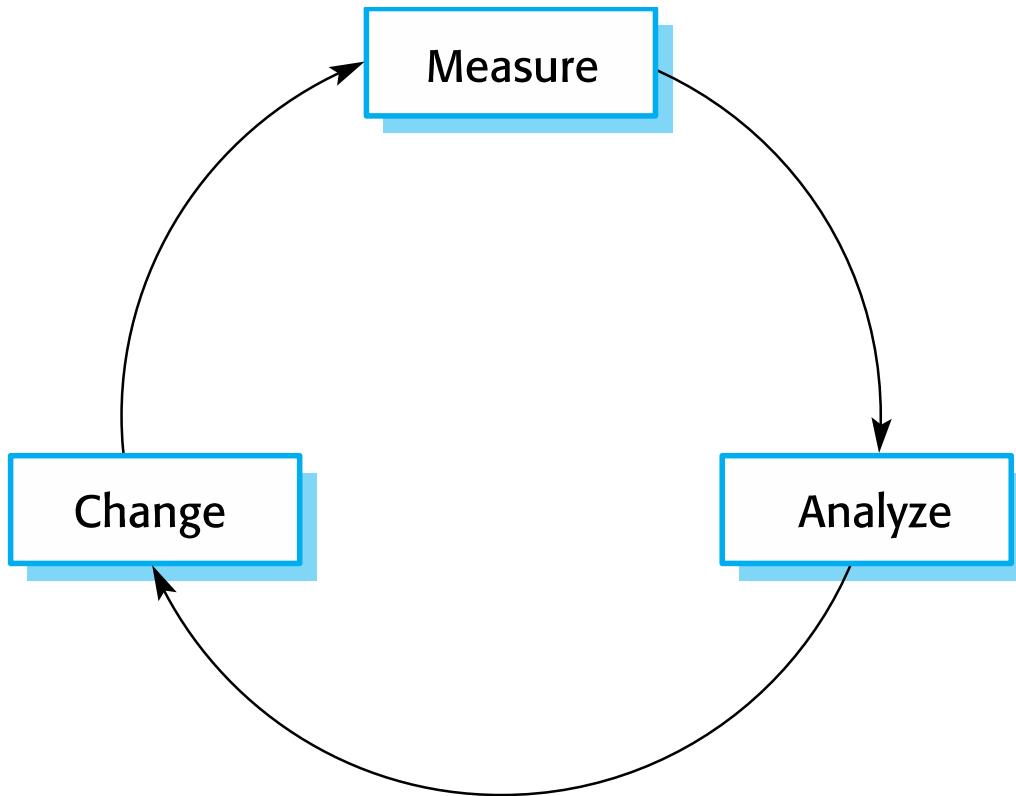
# Approaches to improvement

---

- ✧ The process maturity approach, which focuses on improving process and project management and introducing good software engineering practice.
  - The level of process maturity reflects the extent to which good technical and management practice has been adopted in organizational software development processes.
- ✧ The agile approach, which focuses on iterative development and the reduction of overheads in the software process.
  - The primary characteristics of agile methods are rapid delivery of functionality and responsiveness to changing customer requirements.

# The process improvement cycle

---



# Process improvement activities

---

## ✧ *Process measurement*

- You measure one or more attributes of the software process or product. These measurements forms a baseline that helps you decide if process improvements have been effective.

## ✧ *Process analysis*

- The current process is assessed, and process weaknesses and bottlenecks are identified. Process models (sometimes called process maps) that describe the process may be developed.

## ✧ *Process change*

- Process changes are proposed to address some of the identified process weaknesses. These are introduced and the cycle resumes to collect data about the effectiveness of the changes.

# Process measurement

---

- ✧ Wherever possible, quantitative process data should be collected
  - However, where organisations do not have clearly defined process standards this is very difficult as you don't know what to measure. A process may have to be defined before any measurement is possible.
- ✧ Process measurements should be used to assess process improvements
  - But this does not mean that measurements should drive the improvements. The improvement driver should be the organizational objectives.

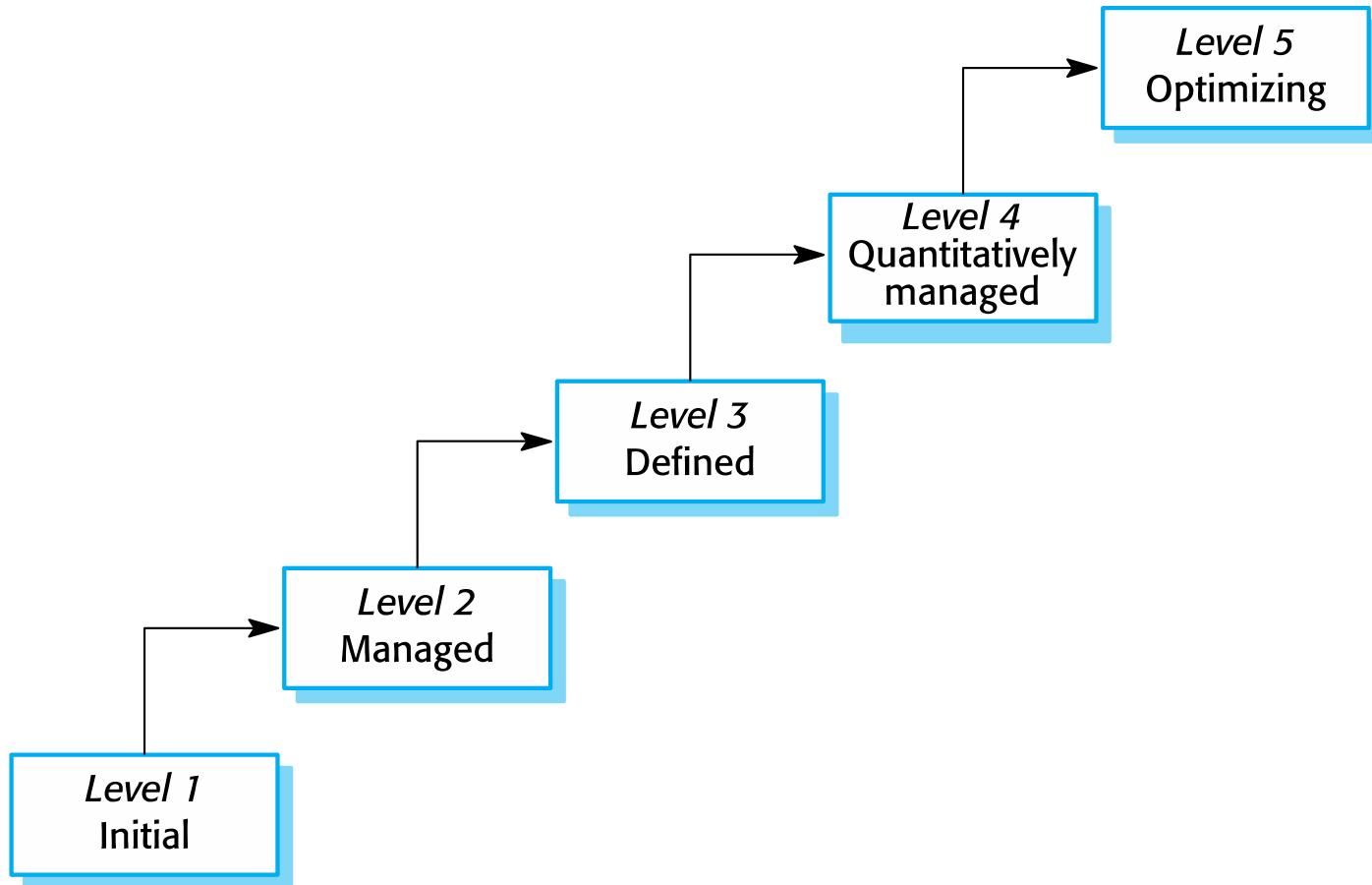
# Process metrics

---

- ✧ Time taken for process activities to be completed
  - E.g. Calendar time or effort to complete an activity or process.
- ✧ Resources required for processes or activities
  - E.g. Total effort in person-days.
- ✧ Number of occurrences of a particular event
  - E.g. Number of defects discovered.

# Capability maturity levels

---



# The SEI capability maturity model

---

- ✧ Initial
  - Essentially uncontrolled
- ✧ Repeatable
  - Product management procedures defined and used
- ✧ Defined
  - Process management procedures and strategies defined and used
- ✧ Managed
  - Quality management strategies defined and used
- ✧ Optimising
  - Process improvement strategies defined and used

# Key points

---

- ✧ Software processes are the activities involved in producing a software system. Software process models are abstract representations of these processes.
- ✧ General process models describe the organization of software processes.
  - Examples of these general models include the ‘waterfall’ model, incremental development, and reuse-oriented development.
- ✧ Requirements engineering is the process of developing a software specification.

## Key points

---

- ✧ Design and implementation processes are concerned with transforming a requirements specification into an executable software system.
- ✧ Software validation is the process of checking that the system conforms to its specification and that it meets the real needs of the users of the system.
- ✧ Software evolution takes place when you change existing software systems to meet new requirements. The software must evolve to remain useful.
- ✧ Processes should include activities such as prototyping and incremental delivery to cope with change.

# Key points

---

- ✧ Processes may be structured for iterative development and delivery so that changes may be made without disrupting the system as a whole.
- ✧ The principal approaches to process improvement are agile approaches, geared to reducing process overheads, and maturity-based approaches based on better process management and the use of good software engineering practice.
- ✧ The SEI process maturity framework identifies maturity levels that essentially correspond to the use of good software engineering practice.

# **Object-Oriented Software Engineering**

## **Practical Software Development using UML**

**Modelling with Classes**

# 5.1 What is UML?

**The Unified Modelling Language is a standard graphical language for modelling object oriented software**

- At the end of the 1980s and the beginning of 1990s, the first object-oriented development processes appeared
- The proliferation of methods and notations tended to cause considerable confusion
- Two important methodologists Rumbaugh and Booch decided to merge their approaches in 1994.
  - They worked together at the Rational Software Corporation
- In 1995, another methodologist, Jacobson, joined the team
  - His work focused on use cases
- In 1997 the Object Management Group (OMG) started the process of UML standardization

# UML diagrams

- Class diagrams
  - describe classes and their relationships
- Interaction diagrams
  - show the behaviour of systems in terms of how objects interact with each other
- State diagrams and activity diagrams
  - show how systems behave internally
- Component and deployment diagrams
  - show how the various components of systems are arranged logically and physically

# UML features

- It has detailed *semantics*
- It has *extension* mechanisms
- It has an associated textual language
  - Object Constraint Language* (OCL)

**The objective of UML is to assist in software development**

—It is not a *methodology*

# What constitutes a good model?

## A model should

- use a standard notation
- be understandable by clients and users
- lead software engineers to have insights about the system
- provide abstraction

Models are used:

- to help create designs
- to permit analysis and review of those designs.
- as the core documentation describing the system.

# 5.2 Essentials of UML Class Diagrams

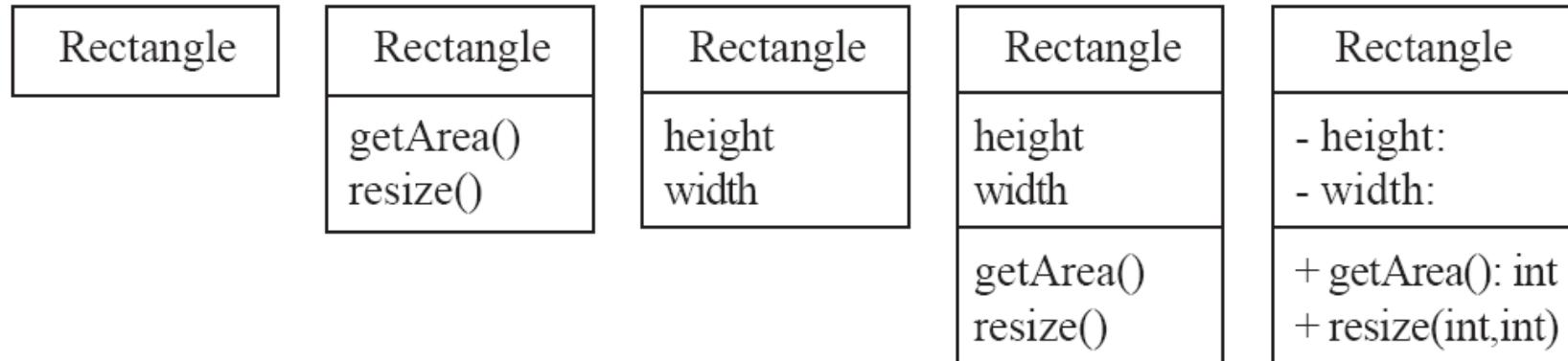
*The main symbols shown on class diagrams are:*

- *Classes*
  - represent the types of data themselves
- *Associations*
  - represent linkages between instances of classes
- *Attributes*
  - are simple data found in classes and their instances
- *Operations*
  - represent the functions performed by the classes and their instances
- *Generalizations*
  - group classes into inheritance hierarchies

# Classes

**A class is simply represented as a box with the name of the class inside**

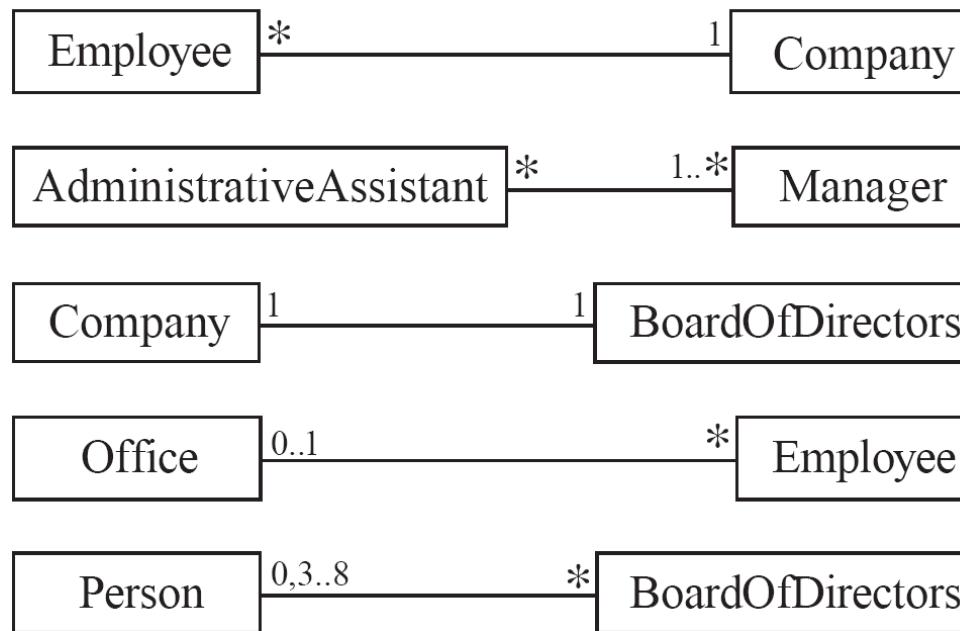
- The diagram may also show the attributes and operations
- The complete signature of an operation is:  
`operationName(parameterName: parameterType ...): returnType`



# 5.3 Associations and Multiplicity

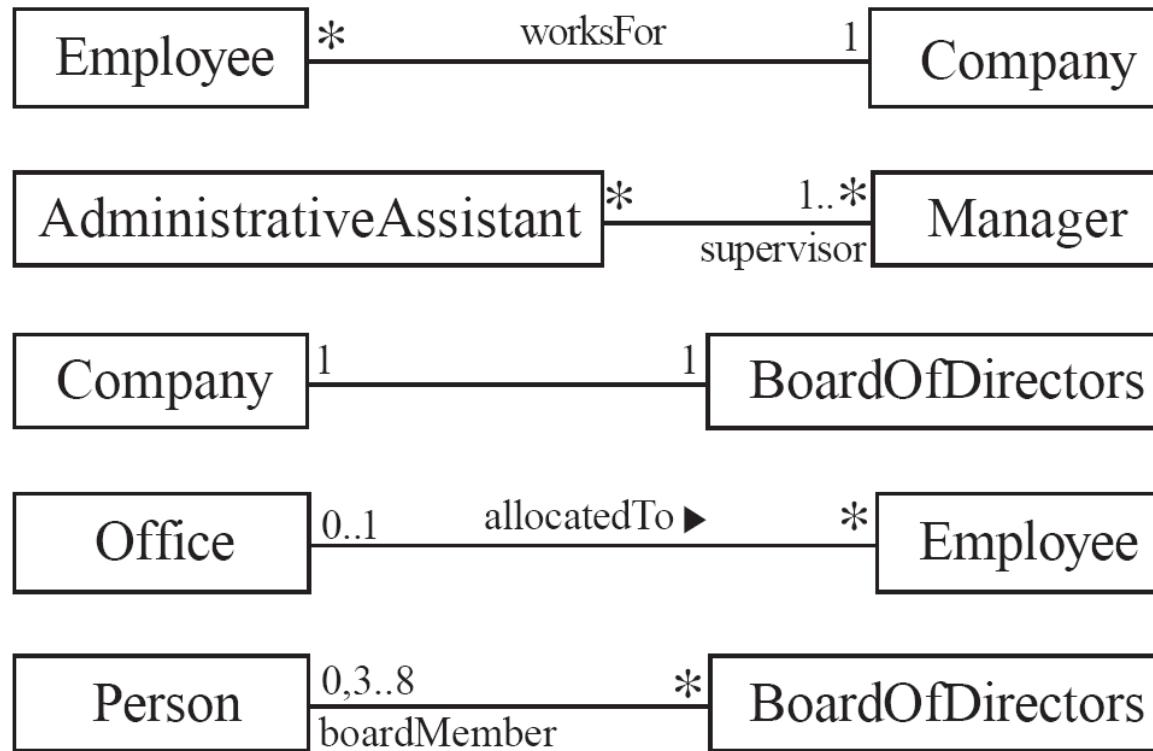
An **association** is used to show how two classes are related to each other

- Symbols indicating *multiplicity* are shown at each end of the association



# Labelling associations

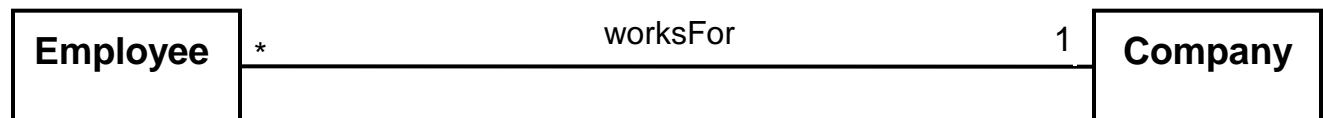
- Each association can be labelled, to make explicit the nature of the association



# Analyzing and validating associations

- **Many-to-one**

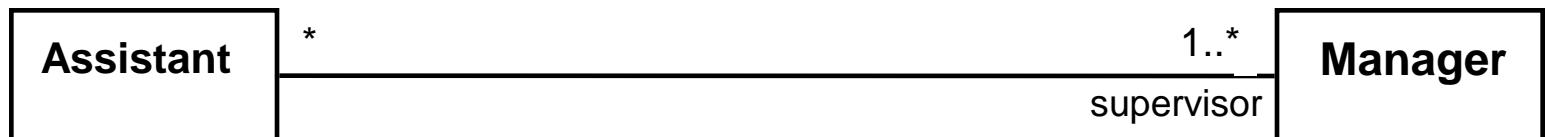
- A company has many employees,
- An employee can only work for one company.
  - This company will not store data about the moonlighting activities of employees!
- A company can have zero employees
  - E.g. a ‘shell’ company
- It is not possible to be an employee unless you work for a company



# Analyzing and validating associations

- **Many-to-many**

- An assistant can work for many managers
- A manager can have many assistants
- Assistants can work in pools
- Managers can have a group of assistants
- Some managers might have zero assistants.
- Is it possible for an assistant to have, perhaps temporarily, zero managers?

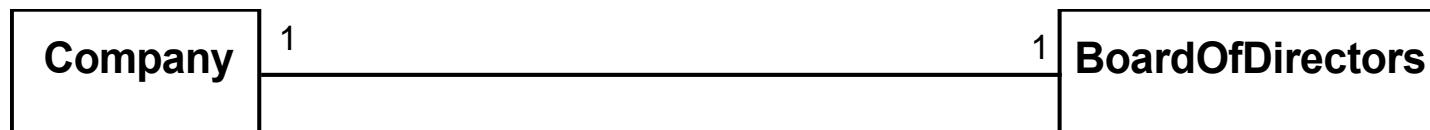


[Open in Umple](#)

# Analyzing and validating associations

- **One-to-one**

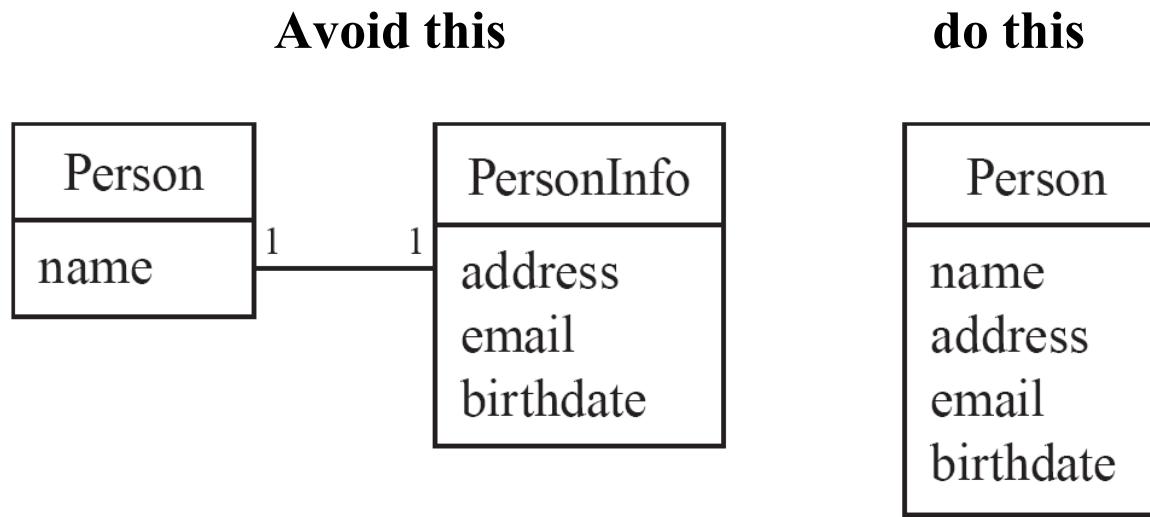
- For each company, there is exactly one board of directors
- A board is the board of only one company
- A company must always have a board
- A board must always be of some company



[Open in Umple](#)

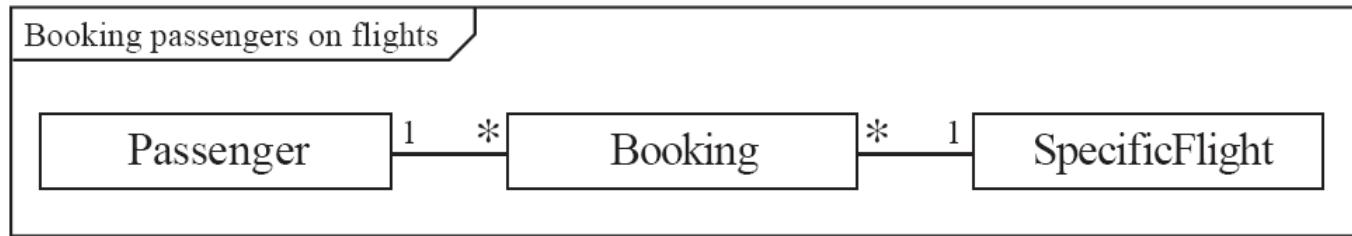
# Analyzing and validating associations

## Avoid unnecessary one-to-one associations



# A more complex example

- A booking is always for exactly one passenger
  - no booking with zero passengers
  - a booking could *never* involve more than one passenger.
- A Passenger can have any number of Bookings
  - a passenger could have no bookings at all
  - a passenger could have more than one booking



- The *frame* around this diagram is an optional feature that any UML 2.0 may possess.

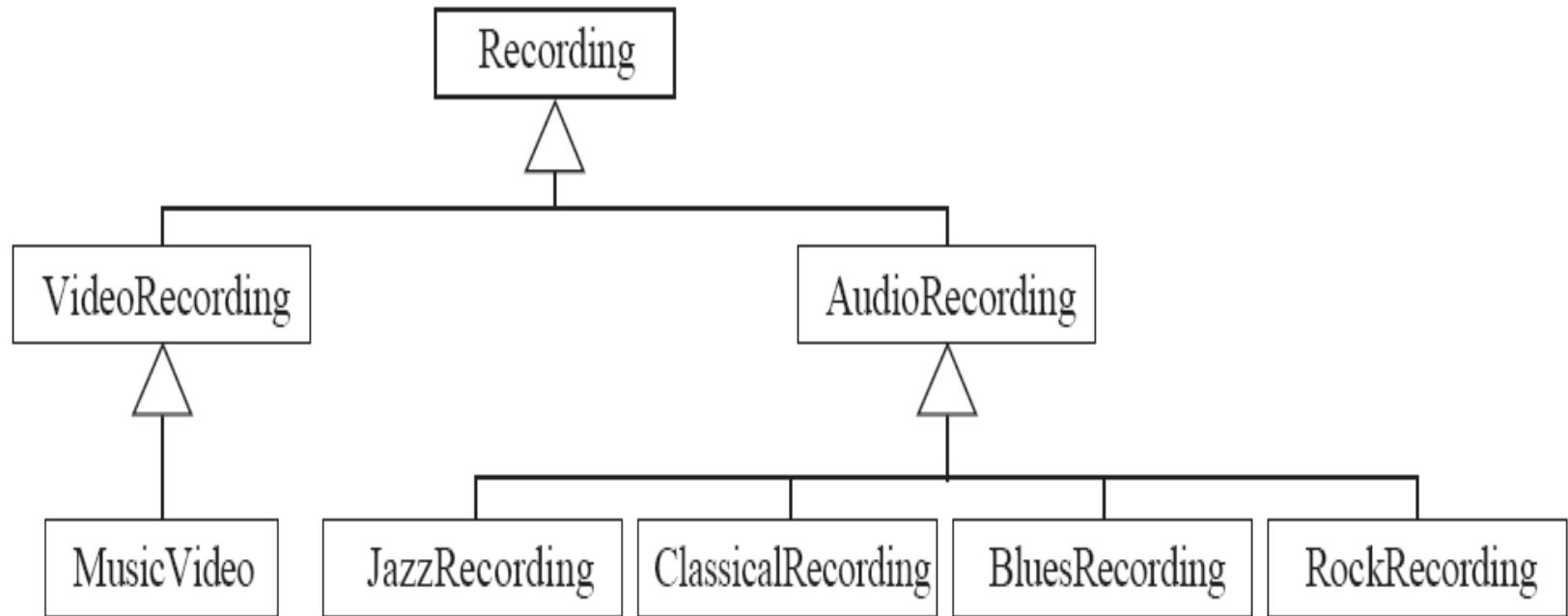
# 5.4 Generalization

## Specializing a superclass into two or more subclasses

- A *generalization set* is a labeled group of generalizations with a common superclass
- The label (sometimes called the *discriminator*) describes the criteria used in the specialization



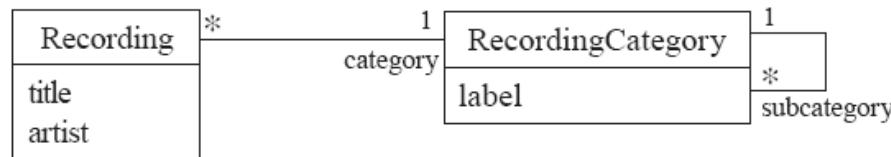
# Avoiding unnecessary generalizations



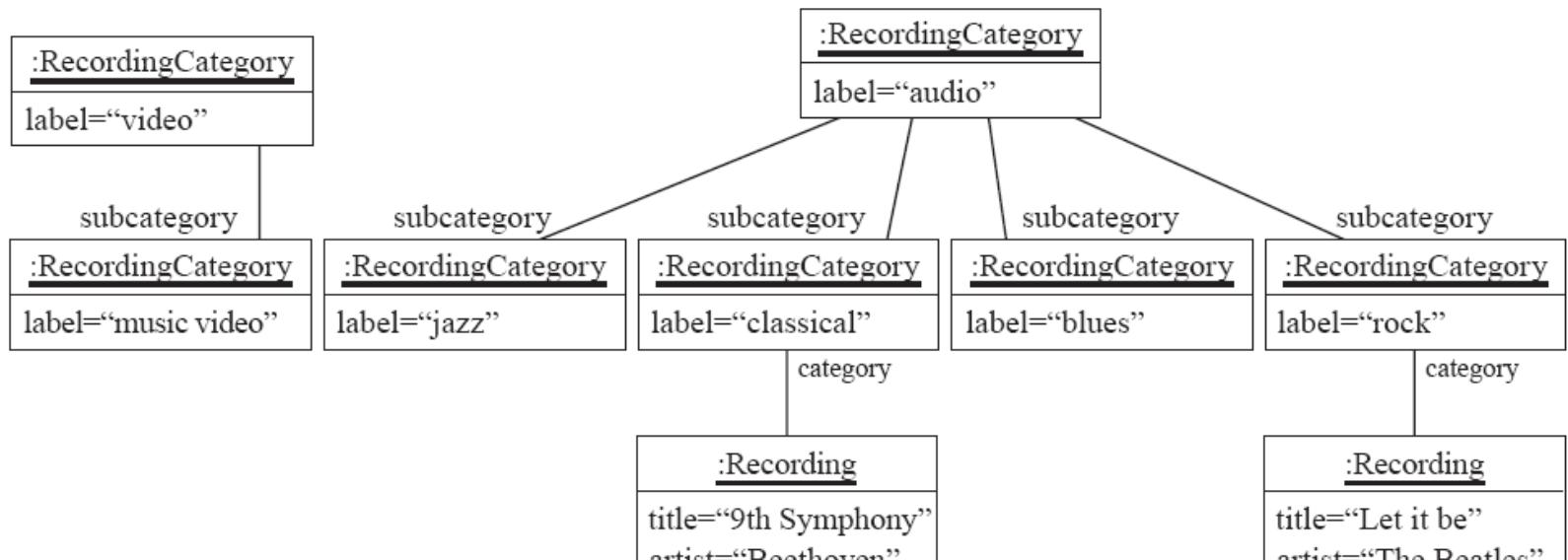
Inappropriate hierarchy of  
classes, which should be  
instances

# Avoiding unnecessary generalizations (cont)

[Open in Umple](#)



(a)



(b)

Improved class diagram, with its corresponding instance diagram

# Associations versus generalizations in object diagrams

- Associations describe the relationships that will exist between *instances* at run time.
  - When you show an instance diagram generated from a class diagram, there will be an instance of *both* classes joined by an association
- Generalizations describe relationships between *classes* in class diagrams.
  - They do not appear in instance diagrams at all.
  - An instance of any class should also be considered to be an instance of each of that class's superclasses

# When to use an aggregation

**As a general rule, you can mark an association as an aggregation if the following are true:**

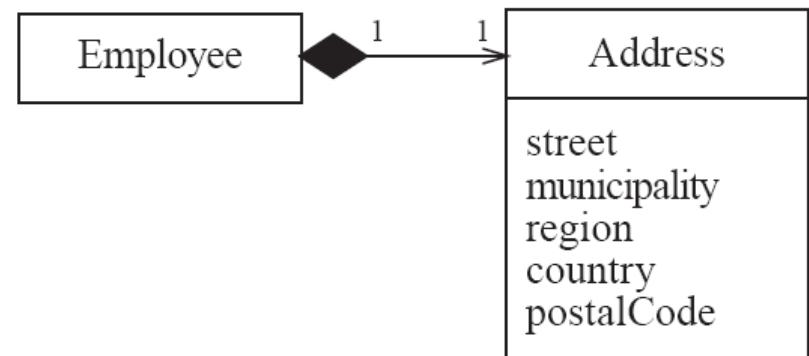
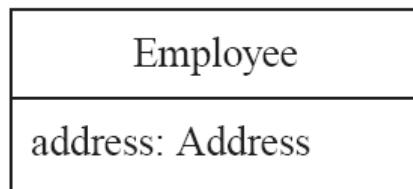
- You can state that
  - the parts ‘are part of’ the aggregate
  - or the aggregate ‘is composed of’ the parts
- When something owns or controls the aggregate, then they also own or control the parts

# Composition

- A *composition* is a strong kind of aggregation
  - if the aggregate is destroyed, then the parts are destroyed as well



- Two alternatives for addresses



# 5.7 Object Constraint Language (OCL)

**OCL is a *specification* language designed to formally specify constraints in software modules**

- An OCL expression simply specifies a logical fact (a constraint) about the system that must remain **true**
- A constraint cannot have any side-effects
  - it cannot compute a non-Boolean result nor modify any data.
- OCL statements in class diagrams can specify what the values of attributes and associations must be

# OCL statements

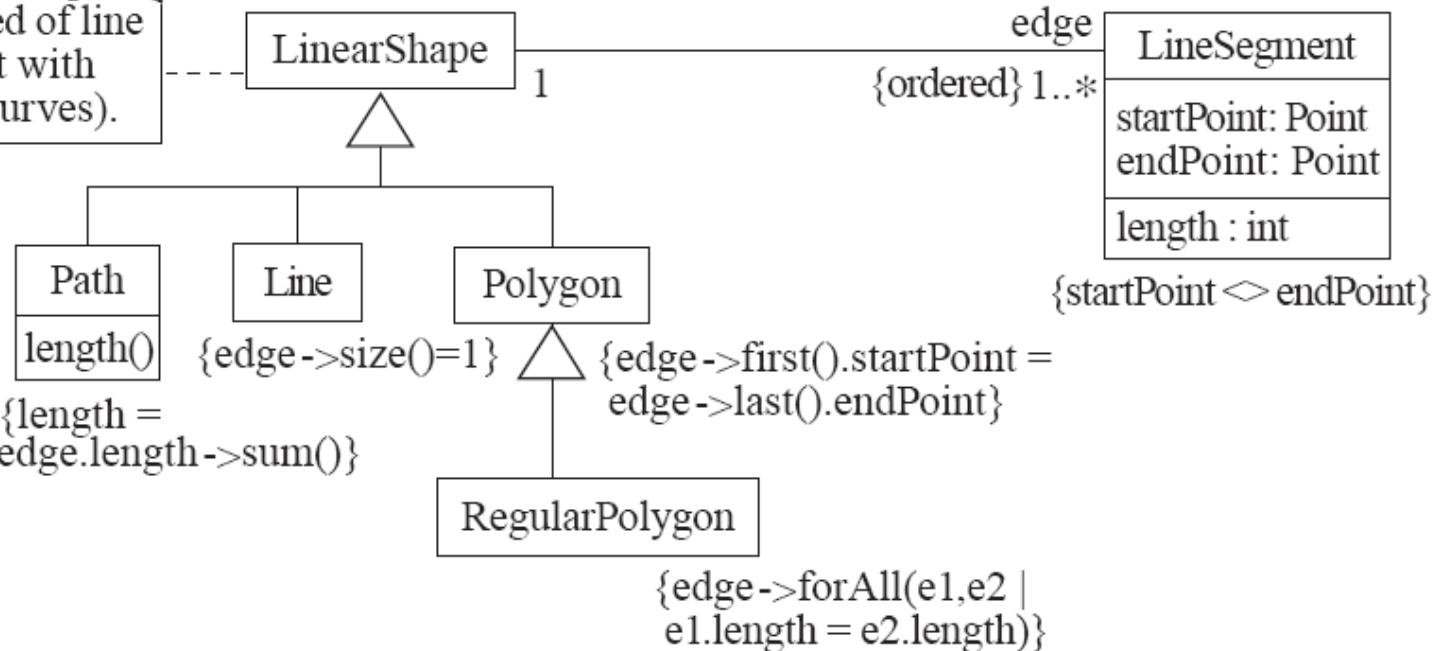
**OCL statements can be built from:**

- References to role names, association names, attributes and the results of operations
- The logical values **true** and **false**
- Logical operators such as **and**, **or**, **=**, **>**, **<** or **<>** (not equals)
- String values such as: '**a string**'
- Integers and real numbers
- Arithmetic operations **\***, **/**, **+**, **-**

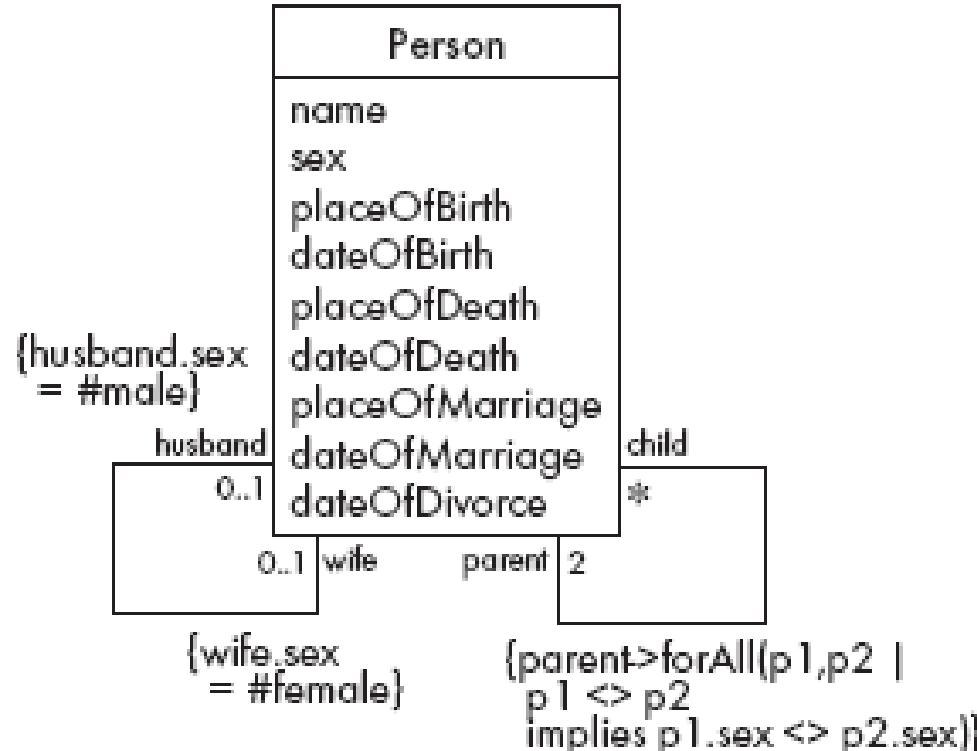
# An example: constraints on Polygons

a LinearShape is any shape that can be constructed of line segments (in contrast with shapes that contain curves).

{edge->forAll(e1,e2 |  
e1 <> e2  
implies e1.startPoint <> e2.startpoint  
and e1.endPoint <> e2.endpoint)}

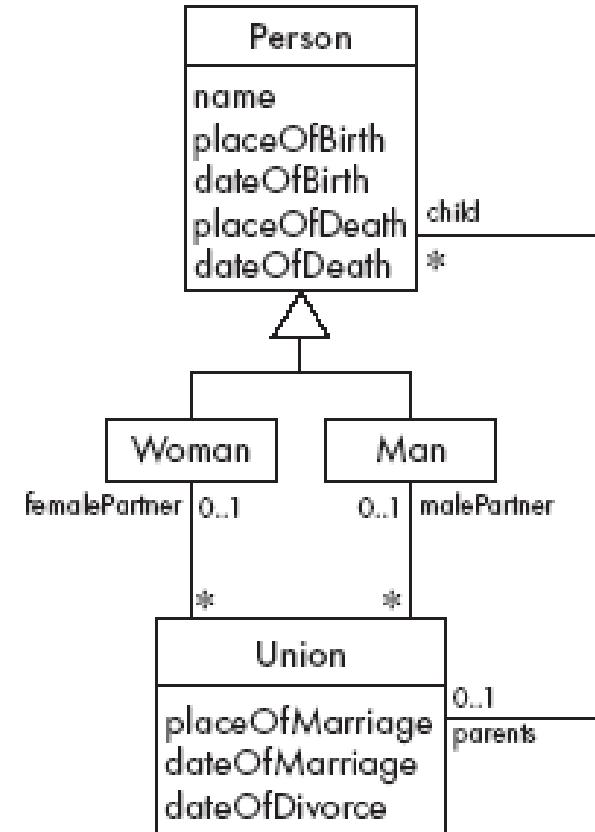
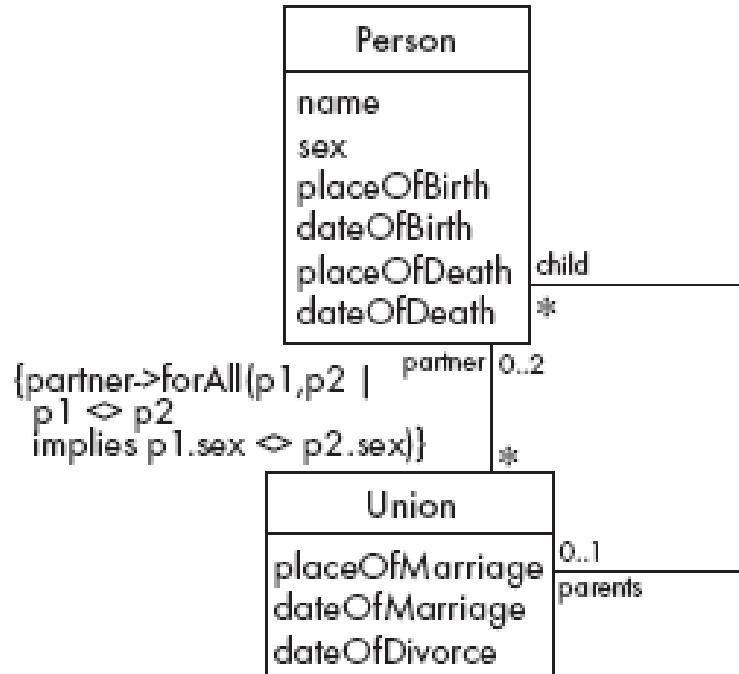


# 5.8 Detailed Example: A Class Diagram for Genealogy



- Problems
  - A person must have two parents
  - Marriages not properly accounted for

# Genealogy example: Possible solutions



[Open in Umple](#)

# 5.9 The Process of Developing Class Diagrams

**You can create UML models at different stages and with different purposes and levels of details**

- **Exploratory domain model:**
  - Developed in domain analysis to learn about the domain
- **System domain model:**
  - Models aspects of the domain represented by the system
- **System model:**
  - Includes also classes used to build the user interface and system architecture

# System domain model vs System model

Type of model	<i>Contains elements that represent things in the domain</i>	<i>Models only things that will actually be implemented</i>	<i>Contains elements that do not represent things in the domain, but are needed to build a complete system</i>
Exploratory domain model: developed in domain analysis to learn about the domain	Yes	No	No
System domain model: models those aspects of the domain represented by the system	Yes	Yes	No
System model: includes classes used to build the user interface and system architecture	Yes	Yes	Yes

# System domain model vs System model

- The *system domain model* omits many classes that are needed to build a complete system
  - Can contain less than half the classes of the system.
  - Should be developed to be used independently of particular sets of
    - user interface classes
    - architectural classes
- The complete *system model* includes
  - The system domain model
  - User interface classes
  - Architectural classes
  - Utility classes

# Suggested sequence of activities

- Identify a first set of candidate **classes**
  - Add **associations** and **attributes**
  - Find **generalizations**
  - List the main **responsibilities** of each class
  - Decide on specific **operations**
  - **Iterate** over the entire process until the model is satisfactory
    - Add or delete classes, associations, attributes, generalizations, responsibilities or operations
    - Identify interfaces
    - Apply design patterns (Chapter 6)
- Don't be too disorganized. Don't be too rigid either.*

# Identifying classes

- When developing a domain model you tend to *discover* classes
- When you work on the user interface or the system architecture, you tend to *invent* classes
  - Needed to solve a particular design problem
  - (Inventing may also occur when creating a domain model)
- Reuse should always be a concern
  - Frameworks
  - System extensions
  - Similar systems

# A simple technique for discovering domain classes

- Look at a source material such as a description of requirements
- Extract the *nouns* and *noun phrases*
- Eliminate nouns that:
  - are redundant
  - represent instances
  - are vague or highly general
  - not needed in the application
- Pay attention to classes in a domain model that represent *types of users* or other actors

# Identifying associations and attributes

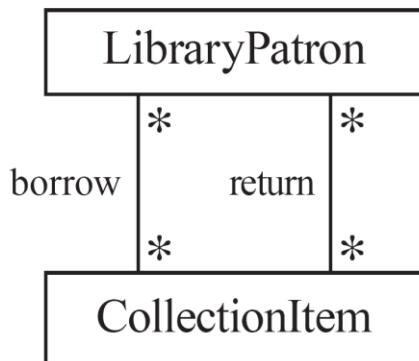
- Start with classes you think are most **central** and important
- Decide on the clear and obvious data it must contain and its relationships to other classes.
- Work outwards towards the classes that are less important.
- Avoid adding many associations and attributes to a class
  - A system is simpler if it manipulates less information

# Tips about identifying and specifying valid associations

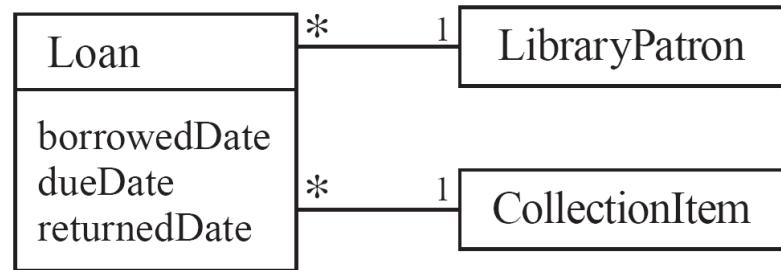
- An association should exist if a class
  - *possesses*
  - *controls*
  - *is connected to*
  - *is related to*
  - *is a part of*
  - *has as parts*
  - *is a member of*, or
  - *has as members*
- some other class in your model
- Specify the multiplicity at both ends
- Label it clearly.

# Actions versus associations

- A common mistake is to represent *actions* as if they were associations



Bad, due to the use of associations  
that are actions



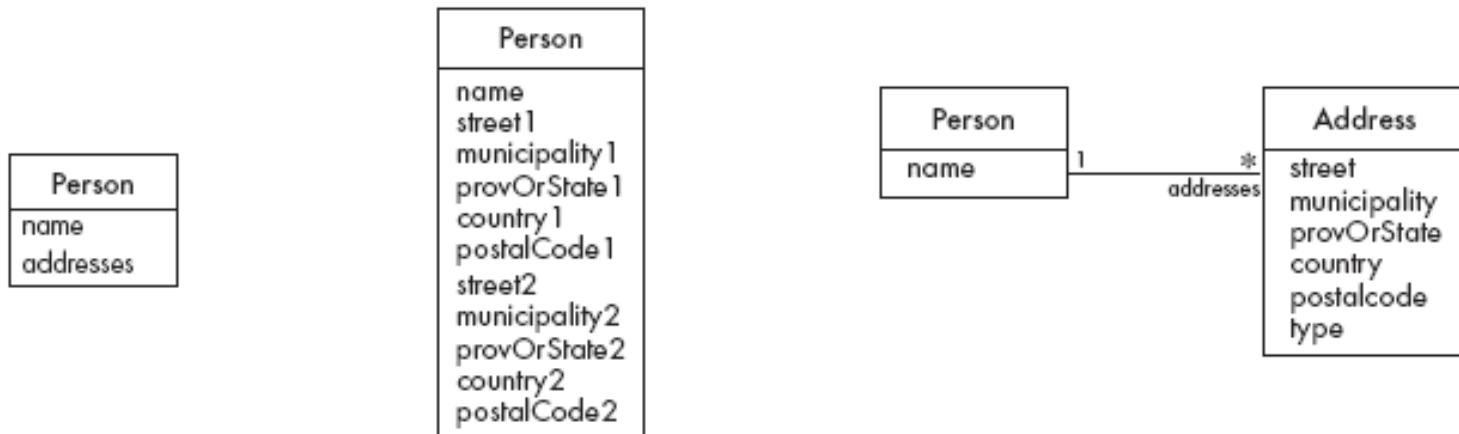
Better: The **borrow** operation creates a **Loan**, and  
the **return** operation sets the **returnedDate**  
attribute.

# Identifying attributes

- Look for information that must be maintained about each class
- Several nouns rejected as classes, may now become attributes
- An attribute should generally contain a simple value
  - E.g. string, number

# Tips about identifying and specifying valid attributes

- It is not good to have many duplicate attributes
- If a subset of a class' s attributes form a coherent group, then create a distinct class containing these attributes

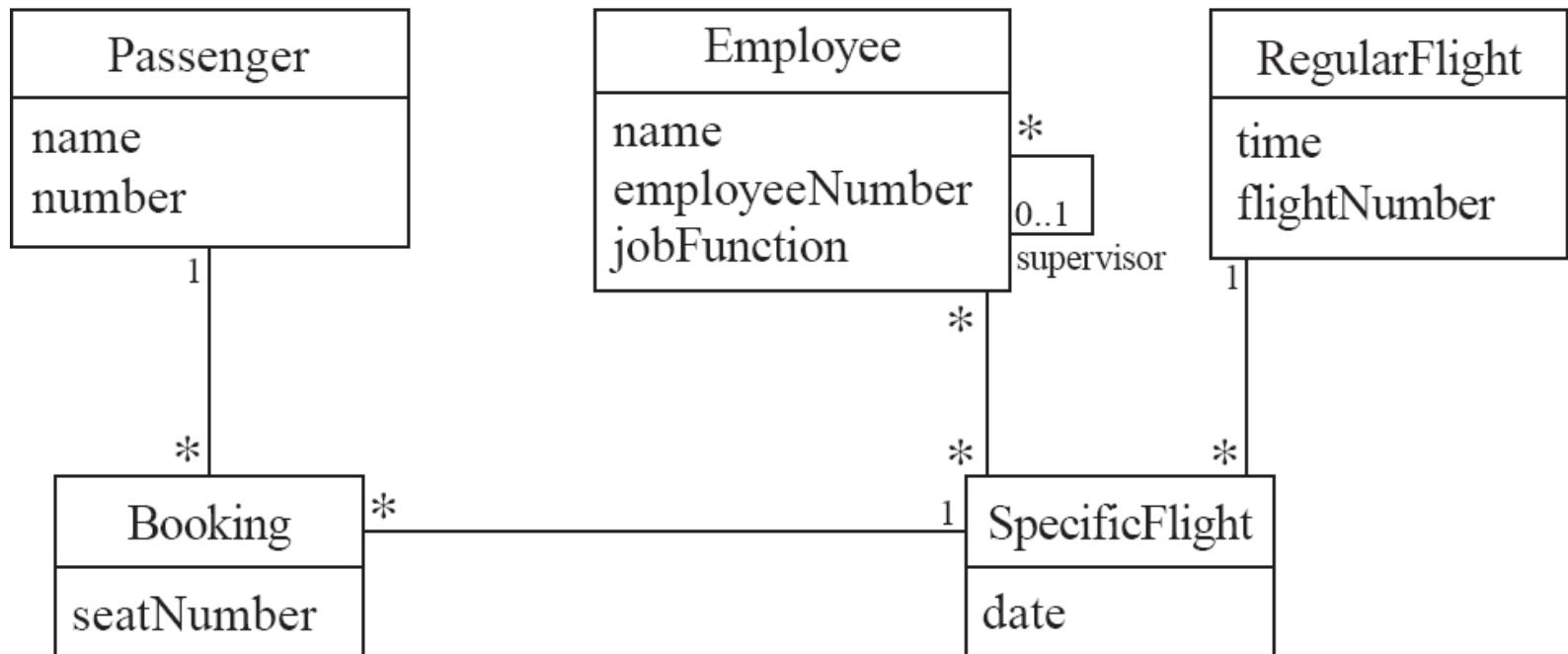


Bad, due to  
a plural attribute

Bad, due to too many  
attributes, and the  
inability to add more  
addresses

Good solution. The type indicates whether it  
is a home address, business address etc.

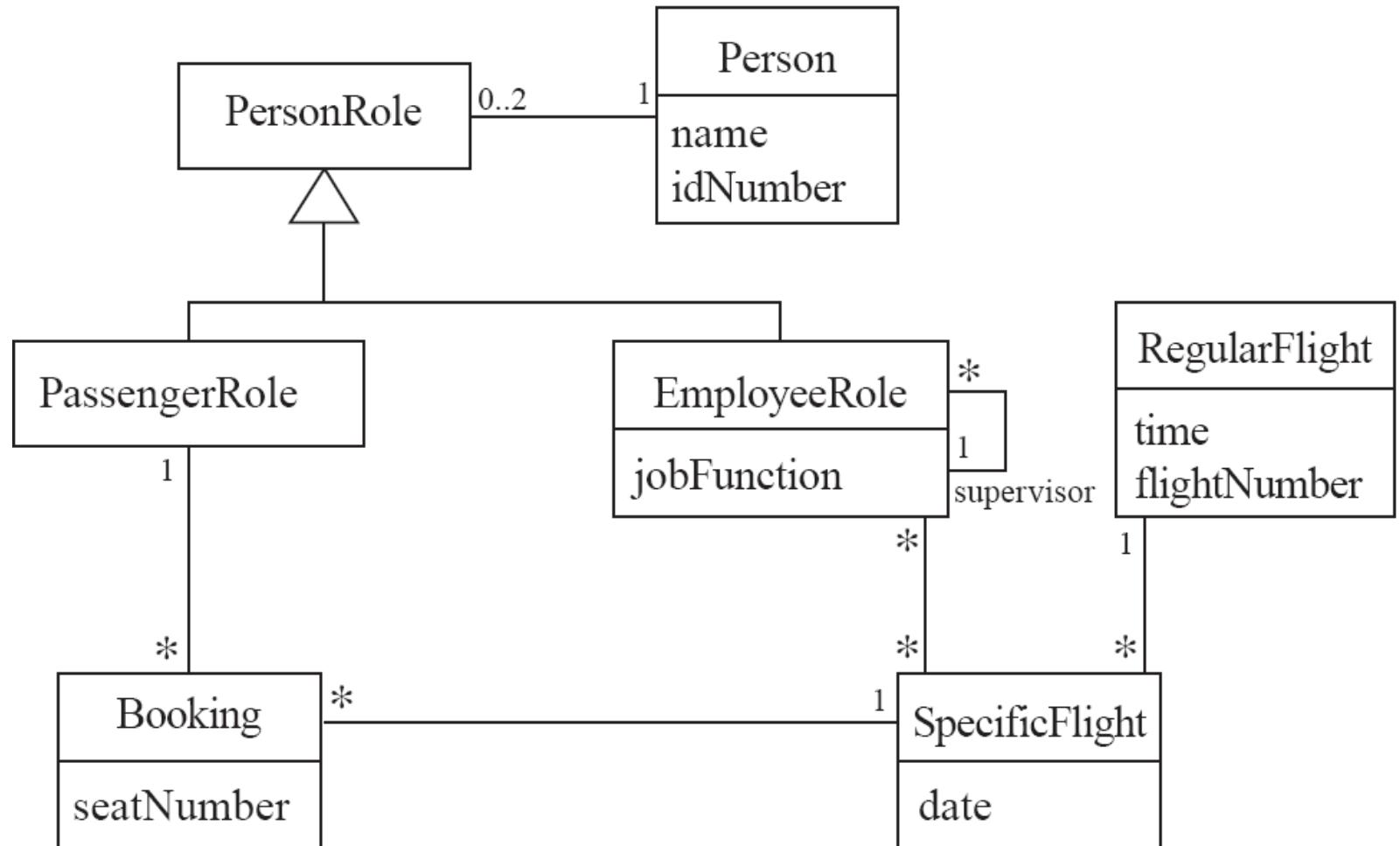
# An example (attributes and associations)



# Identifying generalizations and interfaces

- There are two ways to identify generalizations:
  - bottom-up
    - Group together similar classes creating a new superclass
  - top-down
    - Look for more general classes first, specialize them if needed
- Create an *interface*, instead of a superclass if
  - The classes are very dissimilar except for having a few operations in common
  - One or more of the classes already have their own superclasses
  - Different implementations of the same class might be available

# An example (generalization)



# Allocating responsibilities to classes

A **responsibility** is something that the system is required to do.

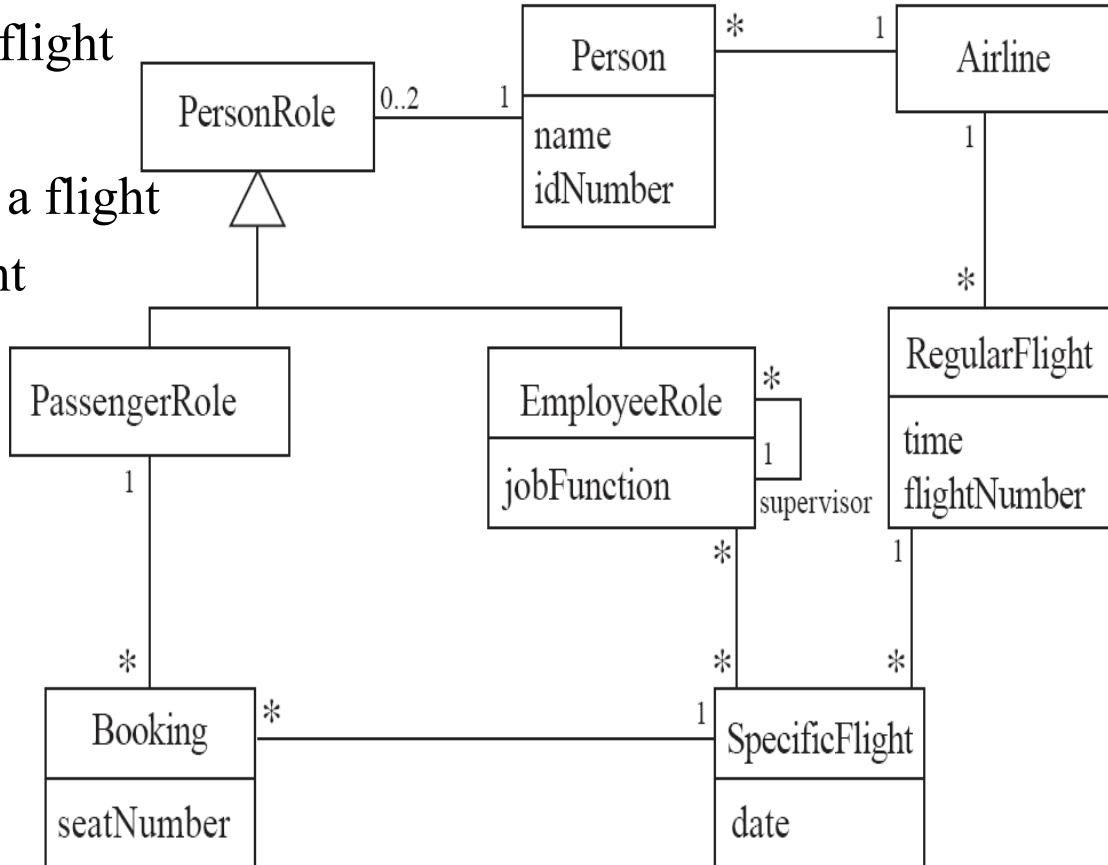
- Each functional requirement must be attributed to one of the classes
  - All the responsibilities of a given class should be *clearly related*.
  - If a class has too many responsibilities, consider *splitting* it into distinct classes
  - If a class has no responsibilities attached to it, then it is probably *useless*
  - When a responsibility cannot be attributed to any of the existing classes, then a *new class* should be created
- To determine responsibilities
  - Perform use case analysis
  - Look for verbs and nouns describing *actions* in the system description

# Categories of responsibilities

- Setting and getting the values of attributes
- Creating and initializing new instances
- Loading to and saving from persistent storage
- Destroying instances
- Adding and deleting links of associations
- Copying, converting, transforming, transmitting or outputting
- Computing numerical results
- Navigating and searching
- Other specialized work

# An example (responsibilities)

- Creating a new regular flight
- Searching for a flight
- Modifying attributes of a flight
- Creating a specific flight
- Booking a passenger
- Canceling a booking



# Prototyping a class diagram on paper

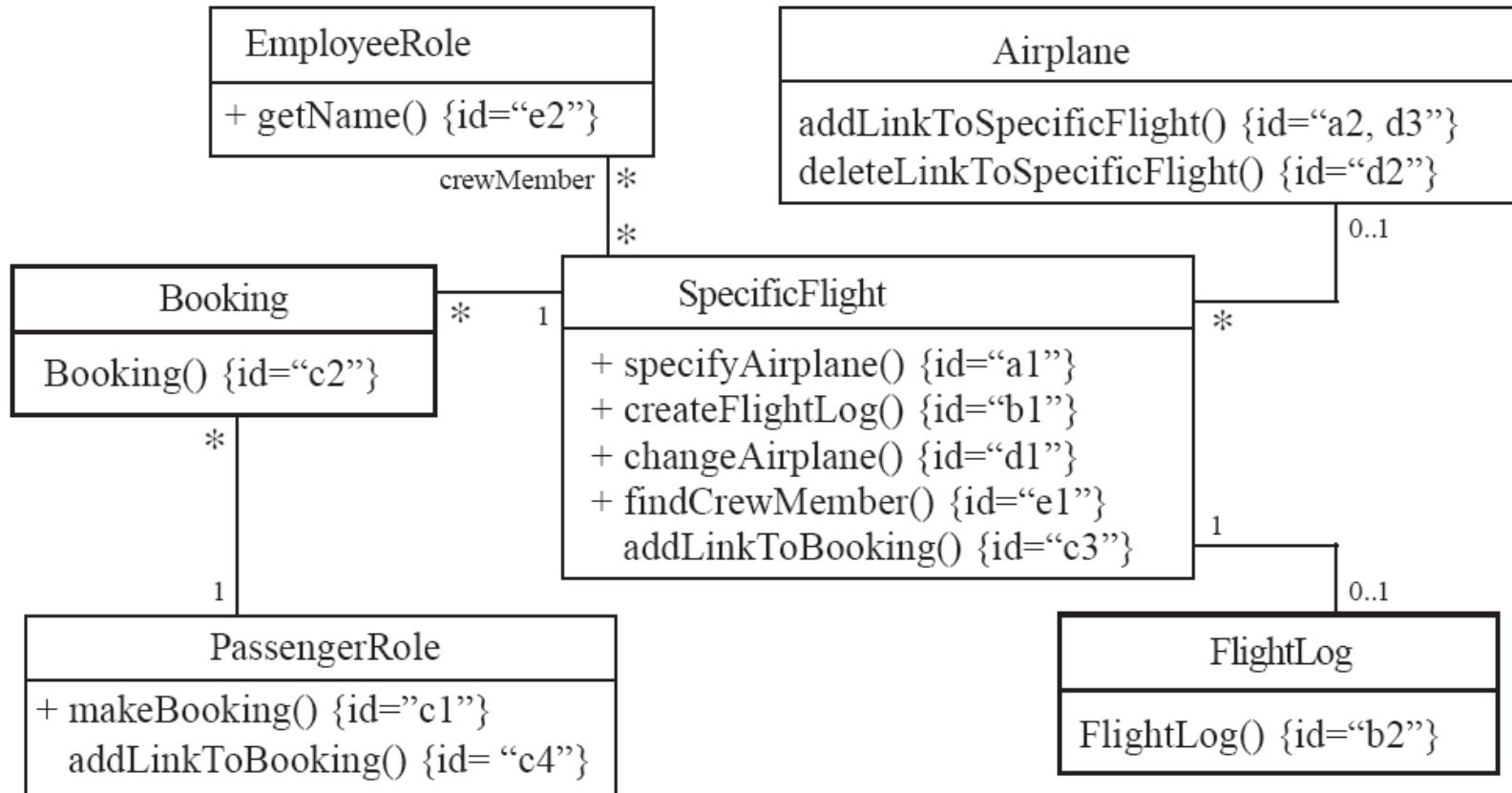
- As you identify classes, you write their names on small cards
- As you identify attributes and responsibilities, you list them on the cards
  - If you cannot fit all the responsibilities on one card:
    - this suggests you should split the class into two related classes.
- Move the cards around on a whiteboard to arrange them into a class diagram.
- Draw lines among the cards to represent associations and generalizations.

# Identifying operations

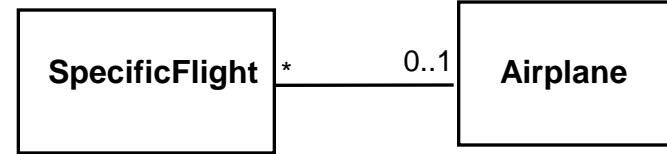
**Operations are needed to realize the responsibilities of each class**

- There may be several operations per responsibility
- The main operations that implement a responsibility are normally declared **public**
- Other methods that collaborate to perform the responsibility must be as private as possible

# An example (class collaboration)



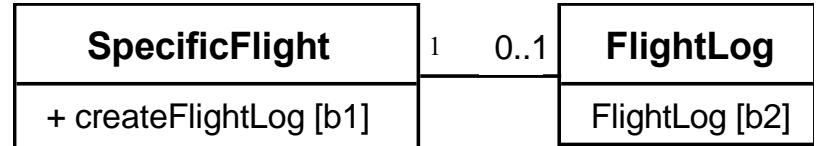
# Class collaboration ‘a’



***Making a bi-directional link between two existing objects;***  
e.g. adding a link between an instance of SpecificFlight and  
an instance of Airplane.

1. (public) The instance of SpecificFlight
  - **makes a one-directional link to the instance of Airplane**
  - **then calls operation 2.**
2. (non-public) The instance of Airplane
  - **makes a one-directional link back to the instance of SpecificFlight**

# Class collaboration ‘b’

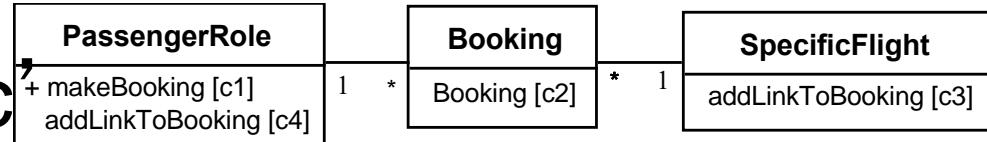


*Creating an object and linking it to an existing object*

e.g. creating a **FlightLog**, and linking it to a **SpecificFlight**.

1. (public) The instance of **SpecificFlight**
  - calls the constructor of FlightLog (operation 2)**
  - then makes a one-directional link to the new instance of FlightLog.**
2. (non-public) Class **FlightLog**'s constructor
  - makes a one-directional link back to the instance of SpecificFlight.**

# Class collaboration ‘c’



*Creating an association class, given two existing objects*

e.g. creating an instance of **Booking**, which will link a **SpecificFlight** to a **PassengerRole**.

1. (public) The instance of **PassengerRole**
  - calls the constructor of **Booking** (operation 2).
2. (non-public) Class **Booking**'s constructor, among its other actions
  - makes a one-directional link back to the instance of **PassengerRole**
  - makes a one-directional link to the instance of **SpecificFlight**
  - calls operations 3 and 4.
3. (non-public) The instance of **SpecificFlight**
  - makes a one-directional link to the instance of **Booking**.
4. (non-public) The instance of **PassengerRole**
  - makes a one-directional link to the instance of **Booking**.

# Class collaboration ‘d’



## *Changing the destination of a link*

e.g. changing the **Airplane** of to a **SpecificFlight**, from **airplane1** to **airplane2**

1. (public) The instance of **SpecificFlight**

- deletes the link to **airplane1**
- makes a one-directional link to **airplane2**
- calls operation 2
- then calls operation 3.

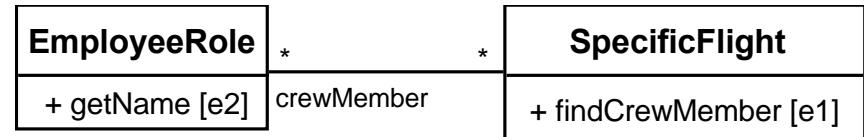
2. (non-public) **airplane1**

- deletes its one-directional link to the instance of **SpecificFlight**.

3. (non-public) **airplane2**

- makes a one-directional link to the instance of **SpecificFlight**.

# Class collaboration ‘e’



## *Searching for an associated instance*

e.g. searching for a crew member associated with a **SpecificFlight** that has a certain name.

1. (public) The instance of **SpecificFlight**
  - creates an Iterator over all the **crewMember** links of the **SpecificFlight**\
  - for each of them call operation 2, until it finds a match.
2. (may be public) The instance of **EmployeeRole** returns its name.

## 5.11 Difficulties and Risks when creating class diagrams

- Modeling is particularly difficult skill
  - Even excellent programmers have difficulty thinking at the appropriate level of abstraction*
  - Education traditionally focus more on design and programming than modeling*
- Resolution:
  - Ensure that team members have adequate training*
  - Have experienced modeler as part of the team*
  - Review all models thoroughly*

# Chapter 2

---

## ■ Software Engineering

*Slide Set to accompany*

*Software Engineering: A Practitioner's Approach, 8/e*  
**by Roger S. Pressman and Bruce R. Maxim**

Slides copyright © 1996, 2001, 2005, 2009, 2014 by Roger S. Pressman

***For non-profit educational use only***

May be reproduced ONLY for student use at the university level when used in conjunction with *Software Engineering: A Practitioner's Approach, 8/e*. Any other reproduction or use is prohibited without the express written permission of the author.

All copyright information MUST appear if these slides are posted on a website for student use.

# Software Engineering

---

- Some realities:
  - *a concerted effort should be made to understand the problem before a software solution is developed*
  - *design becomes a pivotal activity*
  - *software should exhibit high quality*
  - *software should be maintainable*
- The seminal definition:
  - *[Software engineering is] the establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines.*

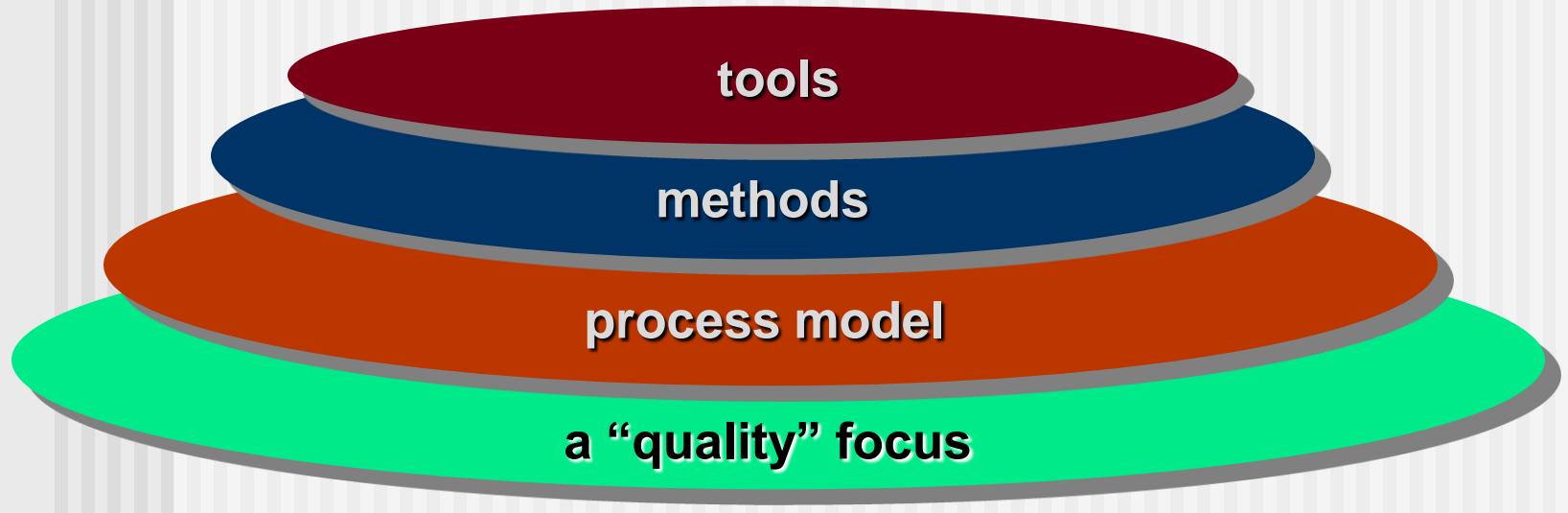
# Software Engineering

---

- The IEEE definition:
  - *Software Engineering:*
  - (1) *The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software.*
  - (2) *The study of approaches as in (1).*

# A Layered Technology

---



***Software Engineering***

# A Process Framework

---

## Process framework

### Framework activities

work tasks

work products

milestones & deliverables

QA checkpoints

### Umbrella Activities

# Framework Activities

---

- Communication
- Planning
- Modeling
  - Analysis of requirements
  - Design
- Construction
  - Code generation
  - Testing
- Deployment

# Umbrella Activities

---

- Software project tracking and control
- Risk management
- Software quality assurance
- Technical reviews
- Measurement
- Software configuration management
- Reusability management
- Work product preparation and production

# Adapting a Process Model

---

- the overall flow of activities, actions, and tasks and the interdependencies among them
- the degree to which actions and tasks are defined within each framework activity
- the degree to which work products are identified and required
- the manner which quality assurance activities are applied
- the manner in which project tracking and control activities are applied
- the overall degree of detail and rigor with which the process is described
- the degree to which the customer and other stakeholders are involved with the project
- the level of autonomy given to the software team
- the degree to which team organization and roles are prescribed

# The Essence of Practice

---

- Polya suggests:

1. *Understand the problem* (communication and analysis).
2. *Plan a solution* (modeling and software design).
3. *Carry out the plan* (code generation).
4. *Examine the result for accuracy* (testing and quality assurance).

# Understand the Problem

---

- *Who has a stake in the solution to the problem?* That is, who are the stakeholders?
- *What are the unknowns?* What data, functions, and features are required to properly solve the problem?
- *Can the problem be compartmentalized?* Is it possible to represent smaller problems that may be easier to understand?
- *Can the problem be represented graphically?* Can an analysis model be created?

# Plan the Solution

---

- *Have you seen similar problems before?* Are there patterns that are recognizable in a potential solution? Is there existing software that implements the data, functions, and features that are required?
- *Has a similar problem been solved?* If so, are elements of the solution reusable?
- *Can subproblems be defined?* If so, are solutions readily apparent for the subproblems?
- *Can you represent a solution in a manner that leads to effective implementation?* Can a design model be created?

# Carry Out the Plan

---

- *Does the solution conform to the plan?* Is source code traceable to the design model?
- *Is each component part of the solution provably correct?* Has the design and code been reviewed, or better, have correctness proofs been applied to algorithm?

# Examine the Result

---

- *Is it possible to test each component part of the solution?* Has a reasonable testing strategy been implemented?
- *Does the solution produce results that conform to the data, functions, and features that are required?* Has the software been validated against all stakeholder requirements?

# Hooker's General Principles

---

- 1: *The Reason It All Exists*
- 2: *KISS (Keep It Simple, Stupid!)*
- 3: *Maintain the Vision*
- 4: *What You Produce, Others Will Consume*
- 5: *Be Open to the Future*
- 6: *Plan Ahead for Reuse*
- 7: *Think!*

# Software Myths

---

- Affect managers, customers (and other non-technical stakeholders) and practitioners
- Are believable because they often have elements of truth,  
*but ...*
- Invariably lead to bad decisions,  
*therefore ...*
- Insist on reality as you navigate your way through software engineering

# How It all Starts

---

## ■ *SafeHome*:

- Every software project is precipitated by some business need—
  - the need to correct a defect in an existing application;
  - the need to adapt a ‘legacy system’ to a changing business environment;
  - the need to extend the functions and features of an existing application, or
  - the need to create a new product, service, or system.

# Chapter 3

---

## ■ Software Process Structure

*Slide Set to accompany*

*Software Engineering: A Practitioner's Approach, 8/e*  
by Roger S. Pressman and Bruce R. Maxim

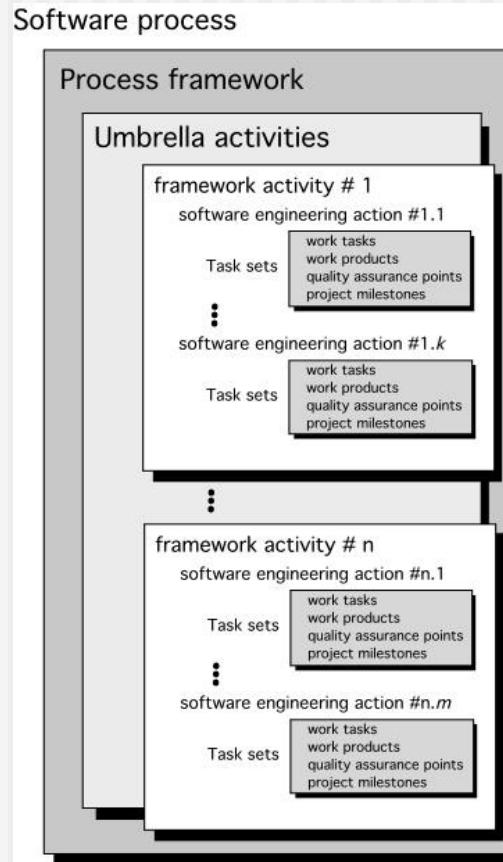
Slides copyright © 1996, 2001, 2005, 2009, 2014 by Roger S. Pressman

***For non-profit educational use only***

May be reproduced ONLY for student use at the university level when used in conjunction with *Software Engineering: A Practitioner's Approach, 8/e*. Any other reproduction or use is prohibited without the express written permission of the author.

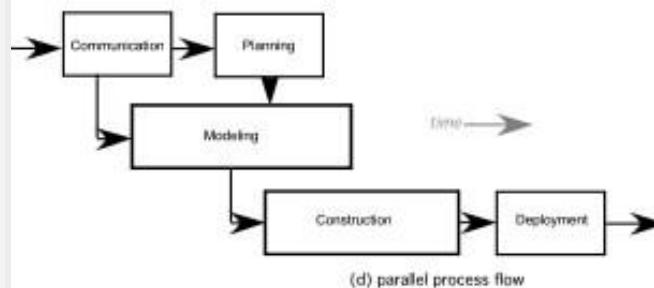
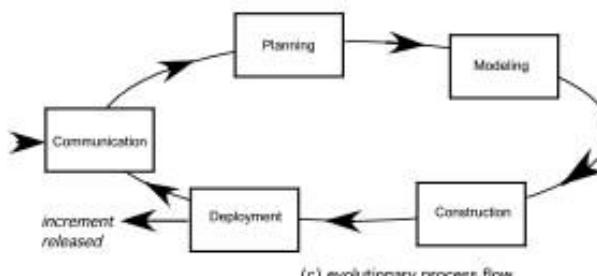
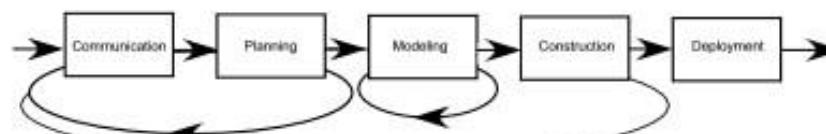
All copyright information MUST appear if these slides are posted on a website for student use.

# A Generic Process Model



These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 8/e* (McGraw-Hill, 2014). Slides copyright 2014 by Roger Pressman.

# Process Flow



These slides are designed to accompany *Software Engineering: A Practitioner's Approach*, 8/e (McGraw-Hill, 2014). Slides copyright 2014 by Roger Pressman.

# Identifying a Task Set

---

- A task set defines the actual work to be done to accomplish the objectives of a software engineering action.
  - A list of the tasks to be accomplished
  - A list of the work products to be produced
  - A list of the quality assurance filters to be applied

# Process Patterns

---

- A *process pattern*
  - describes a process-related problem that is encountered during software engineering work,
  - identifies the environment in which the problem has been encountered, and
  - suggests one or more proven solutions to the problem.
- Stated in more general terms, a process pattern provides you with a *template* [Amb98]—a consistent method for describing problem solutions within the context of the software process.

# Process Pattern Types

---

- *Stage patterns*—defines a problem associated with a framework activity for the process.
- *Task patterns*—defines a problem associated with a software engineering action or work task and relevant to successful software engineering practice
- *Phase patterns*—define the sequence of framework activities that occur with the process, even when the overall flow of activities is iterative in nature.

# Process Assessment and Improvement

---

- **Standard CMMI Assessment Method for Process Improvement (SCAMPI)** — provides a five step process assessment model that incorporates five phases: initiating, diagnosing, establishing, acting and learning.
- **CMM-Based Appraisal for Internal Process Improvement (CBA IPI)**—provides a diagnostic technique for assessing the relative maturity of a software organization; uses the SEI CMM as the basis for the assessment [Dun01]
- **SPICE—The SPICE (ISO/IEC15504)** standard defines a set of requirements for software process assessment. The intent of the standard is to assist organizations in developing an objective evaluation of the efficacy of any defined software process. [ISO08]
- **ISO 9001:2000 for Software**—a generic standard that applies to any organization that wants to improve the overall quality of the products, systems, or services that it provides. Therefore, the standard is directly applicable to software organizations and companies. [Ant06]

# Object-oriented design patterns in UML

Software modeling (401016) – 2016/2017

Ivano Malavolta  
[i.malavolta@vu.nl](mailto:i.malavolta@vu.nl)



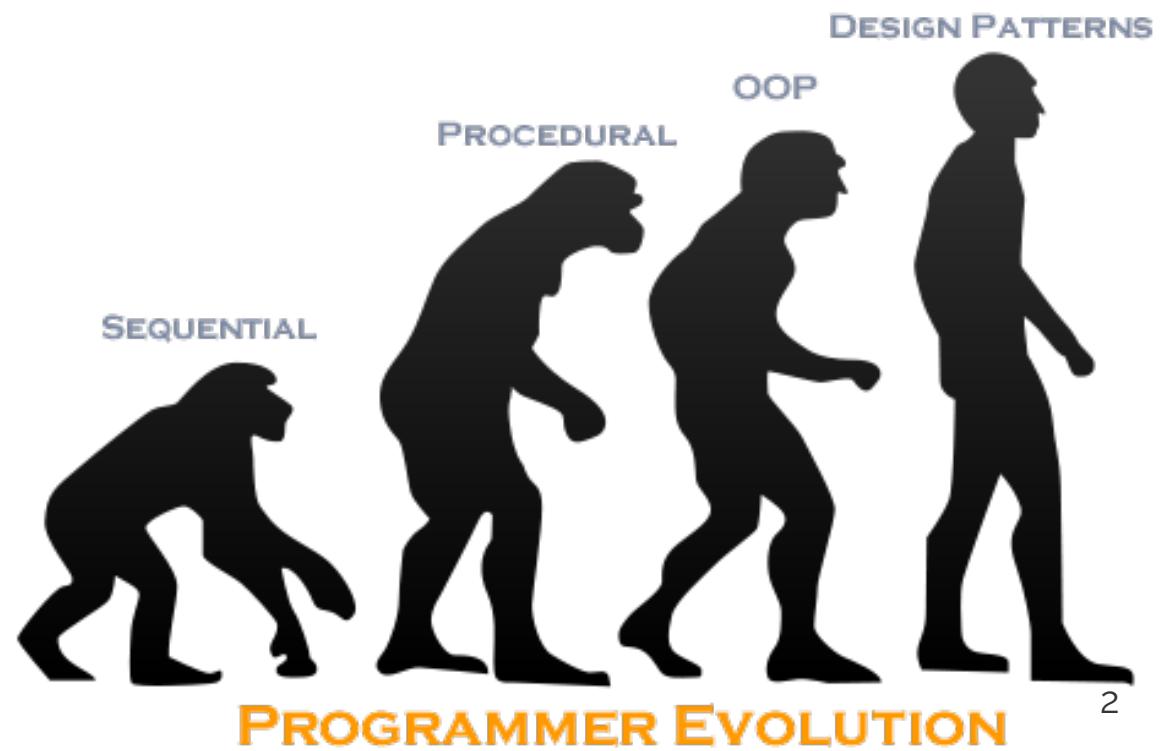
VRIJE  
UNIVERSITEIT  
AMSTERDAM

Software and Services research group (S2)  
Department of Computer Science, Faculty of Sciences  
Vrije Universiteit Amsterdam

# Roadmap

---

- Object-oriented design patterns
  - Creational design patterns
  - Structural design patterns
  - Behavioural design patterns



# What is a design pattern?

---

A reusable form of a solution  
to a common design problem

- a "template" for how to solve a problem that can be used in many different situations
- not a finished design
  - it cannot be transformed directly into source code
- helps the designer in getting to the right design faster

# The “gang of four”

- Erich Gamma
- Richard Helm
- Ralph Johnson
- John Vlissides

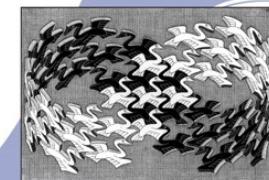


1994: 4 IBM programmers observed and documented 23 common problems and their best accepted solutions

## Design Patterns

Elements of Reusable Object-Oriented Software

Erich Gamma  
Richard Helm  
Ralph Johnson  
John Vlissides



Cover art © 1994 M.C. Escher / Cordon Art - Baarn - Holland. All rights reserved.

Foreword by Grady Booch

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES



VRIJE  
UNIVERSITEIT  
AMSTERDAM

# Types of design patterns

---

CREATIONAL

- how objects can be created
  - maintainability
  - control
  - extensibility

STRUCTURAL

- how to form larger structures
  - management of complexity
  - efficiency

BEHAVIOURAL

- how responsibilities can be assigned to objects
  - objects decoupling
  - flexibility
  - better communication



VRIJE  
UNIVERSITEIT  
AMSTERDAM

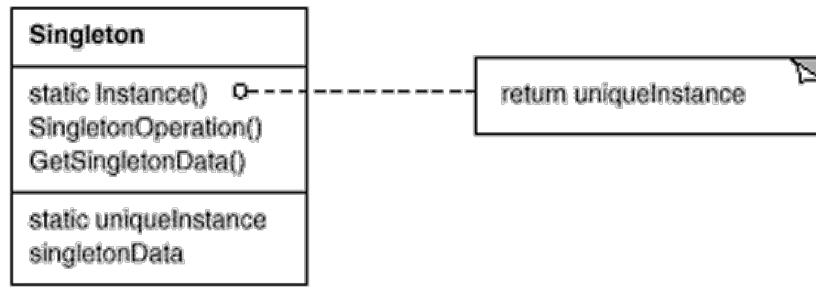
# Essential parts of a design patterns

---

Pattern name	Provides a common vocabulary for software designers
Intent	What does the design pattern do? What is its rationale and intent? What particular design issue or problem does it address?
Solution	The basic elements providing the solution to the problem in terms of: structure, participants, collaborations
Consequences	What are the results and trade offs by applying the design pattern

# Example: the Singleton design pattern

---

Name	Singleton
Intent	<ul style="list-style-type: none"><li>To ensure that only one instance of a class is allowed within a system</li><li>Controlled access to a single object is necessary</li></ul>
Solution	 <pre>classDiagram     class Singleton {         static Instance()         SingletonOperation()         GetSingletonData()     }     Singleton "0..1" --&gt; "1" return uniqueInstance     note over return uniqueInstance: return uniqueInstance</pre>
Consequences	<ul style="list-style-type: none"><li>Controlled access to sole instance</li><li>Reduced name space</li><li>Permits a variable number of instances</li><li>...</li></ul>
Examples	<ul style="list-style-type: none"><li>Logging utility</li><li>An entity representing the whole system</li><li>Central station within a robotic system</li><li>...</li></ul>

---

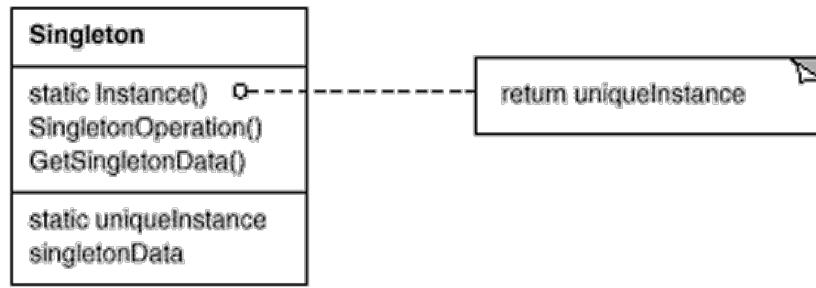
# Creational design patterns

# Creational design patterns

- Singleton studied in this course
  - A class for which only a one single instance can exist in the system
- Factory Method
  - Creates an object without hardcoding its type in the code
- Abstract Factory
  - Creates groups of related objects without specifying the exact concrete classes
- Object Pool
  - Allows to recycle objects that are no longer in use
- Prototype
  - Allows to instantiate a class by copying or cloning the properties of an existing object



# Singleton

Name	Singleton
Intent	<ul style="list-style-type: none"><li>To ensure that only one instance of a class is allowed within a system</li><li>Controlled access to a single object is necessary</li></ul>
Solution	 <p>The diagram illustrates the Singleton pattern. It features a class box labeled 'Singleton' containing the following elements:<ul style="list-style-type: none"><li>Operations: static Instance(), SingletonOperation(), GetSingletonData()</li><li>Attributes: static uniqueInstance, singletonData</li></ul>A dashed arrow originates from the 'uniqueInstance' attribute and points to a separate box containing the code 'return uniqueInstance'. A small icon of a hand holding a flag is positioned next to the return box.</p>
Consequences	<ul style="list-style-type: none"><li>Controlled access to sole instance</li><li>Reduced name space</li><li>Permits a variable number of instances</li><li>...</li></ul>
Examples	<ul style="list-style-type: none"><li>Logging utility</li><li>An entity representing the whole system</li><li>Central station within a robotic system</li><li>...</li></ul>

# Singleton – implementation (1/2)

```
public class SingleObject {  
  
    //create an object of SingleObject  
    private static SingleObject instance = new SingleObject();  
  
    //make the constructor private so that this class cannot be  
    //instantiated  
    private SingleObject(){}
  
  
    //Get the only object available  
    public static SingleObject getInstance(){  
        return instance;  
    }  
  
    public void showMessage(){  
        System.out.println("Hello World!");  
    }  
}
```



## Singleton – implementation (2/2)

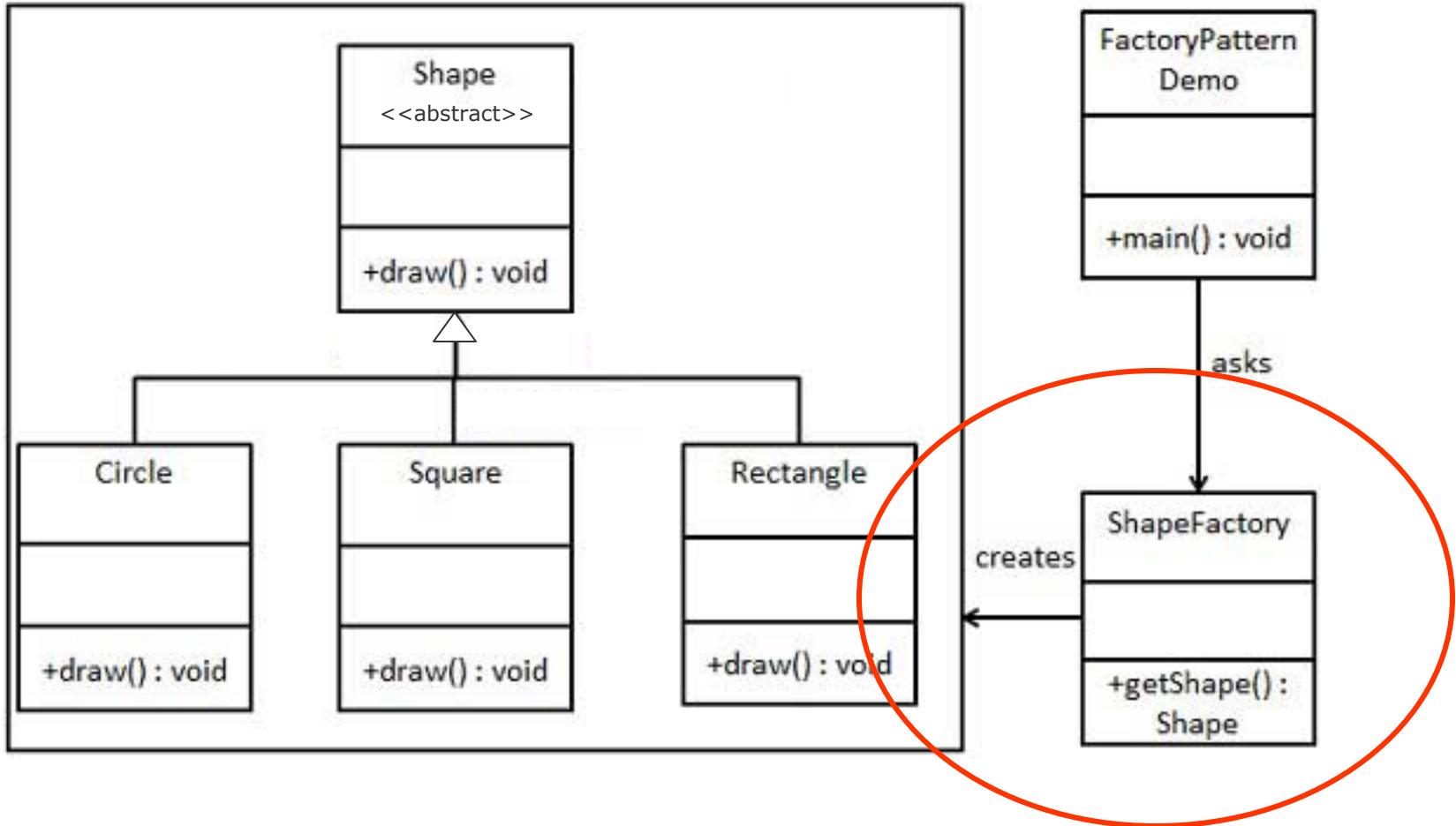
```
public class SingleObject {  
  
    //create an object of SingleObject  
    private static SingleObject instance = new SingleObject();  
  
    public class SingletonPatternDemo {  
        public static void main(String[] args) {  
  
            //illegal construct  
            //Compile Time Error: The constructor SingleObject() is not visible  
            //SingleObject object = new SingleObject();  
  
            //Get the only object available  
            SingleObject object = SingleObject.getInstance();  
  
            //show the message  
            object.showMessage();  
        }  
    }  
}
```



# Factory method

Name	Factory method
Intent	<ul style="list-style-type: none"><li>• to abstract the process of object creation so that the type of the created object can be determined at run-time</li><li>• to make a design more <u>customizable</u> in terms of which objects can be created</li><li>• you want to avoid the new operator because you do not want to hard code which class you want to instantiate</li></ul>
Solution	[see next slide]
Consequences	<ul style="list-style-type: none"><li>• You have a dedicated class for creating instances of objects</li><li>• You can pass arguments to that class for controlling the features of the objects you want to create</li></ul>
Examples	<ul style="list-style-type: none"><li>• A central entity for creating rovers, but you really do not want to know exactly how a rover is created</li><li>• All the cases in which an object does not know what concrete classes it will be required to create at runtime, but just wants to get a class that will do the job</li></ul>

# Factory method – solution by example



# Factory method - implementation

---

```
1 public abstract class Shape {  
2     public abstract void draw();  
3 }  
4  
5 public class Rectangle extends Shape {  
6     public void draw() {  
7         // ... draw rectangle  
8     }  
9 }  
10  
11 public class Square extends Shape {  
12     public void draw() {  
13         // ... draw square  
14     }  
15 }  
16  
17 public class Circle extends Shape {  
18     public void draw() {  
19         // ... draw circle  
20     }  
21 }
```

# Factory method - implementation

```
public class ShapeFactory {  
  
    //use getShape method to get object of type shape  
    public Shape getShape(String shapeType){  
        if(shapeType == null){  
            return null;  
        }  
        if(shapeType.equalsIgnoreCase("CIRCLE")){  
            return new Circle();  
  
        } else if(shapeType.equalsIgnoreCase("RECTANGLE")){  
            return new Rectangle();  
  
        } else if(shapeType.equalsIgnoreCase("SQUARE")){  
            return new Square();  
        }  
  
        return null;  
    }  
}
```



# Factory method - implementation

```
public class ShapeFactory {  
  
    //use getShape method to get object of type shape  
    public Shape getShape(String shapeType){  
        if(shapeType == null)  
            return null;  
        }  
        if(shapeType.equals("CIRCLE"))  
            return new Circle();  
        } else if(shapeType.equals("RECTANGLE"))  
            return new Rectangle();  
        } else if(shapeType.equals("SQUARE"))  
            return new Square();  
        }  
        return null;  
    }
```

```
public class FactoryPatternDemo {  
  
    public static void main(String[] args) {  
        ShapeFactory shapeFactory = new ShapeFactory();  
  
        //get an object of Circle and call its draw method.  
        Shape shape1 = shapeFactory.getShape("CIRCLE");  
  
        //call draw method of Circle  
        shape1.draw();  
  
        //get an object of Rectangle and call its draw method.  
        Shape shape2 = shapeFactory.getShape("RECTANGLE");  
  
        //call draw method of Rectangle  
        shape2.draw();  
  
        //get an object of Square and call its draw method.  
        Shape shape3 = shapeFactory.getShape("SQUARE");  
  
        //call draw method of circle  
        shape3.draw();  
    }
```



---

# Structural design patterns

# Structural design patterns

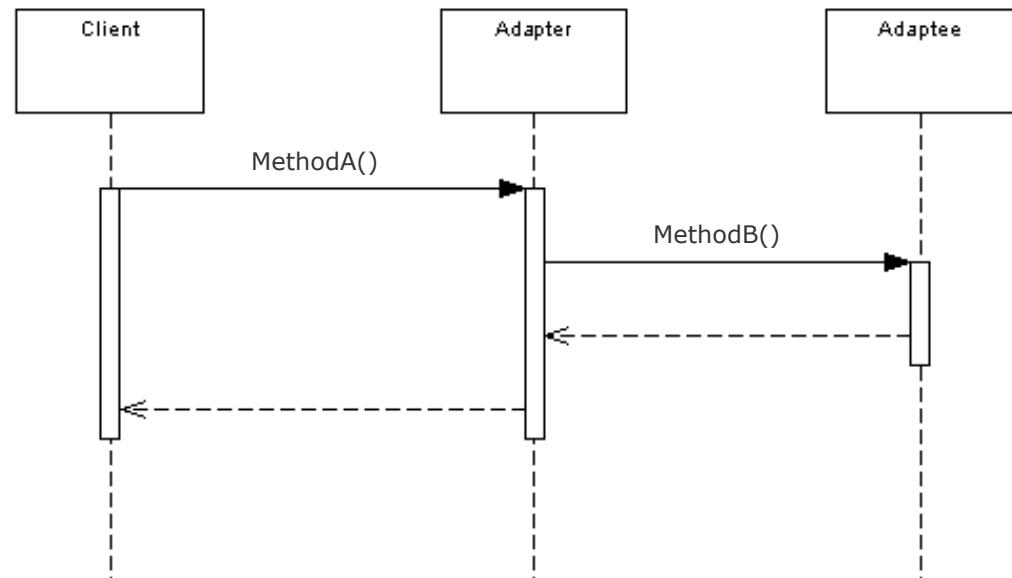
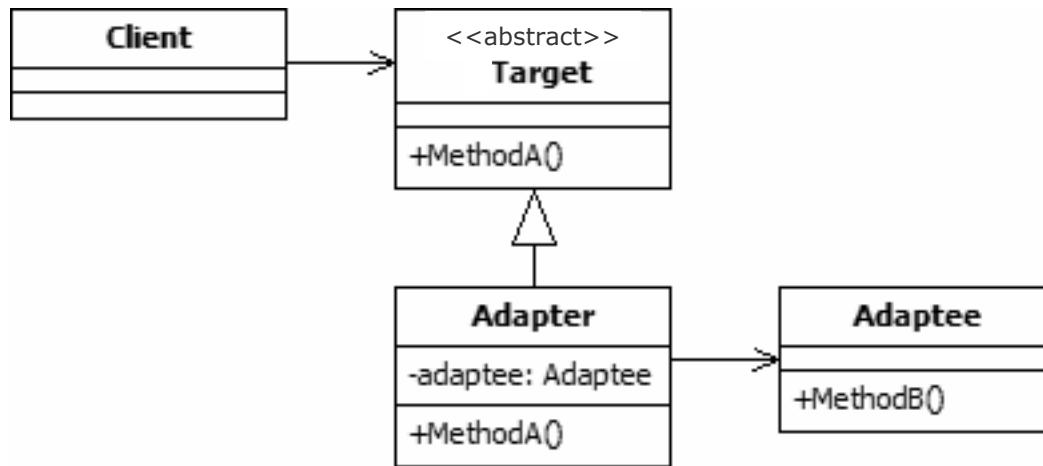
- Adapter studied in this course
  - For gluing together incompatible classes
- Proxy
  - An object representing another object
- Bridge
  - Separates an object's interface from its implementation
- Decorator
  - Add responsibilities to objects dynamically
- Façade
  - A single class that represents an entire subsystem/library
- Flyweight, Composite , Private Class Data
  - ...



# Adapter

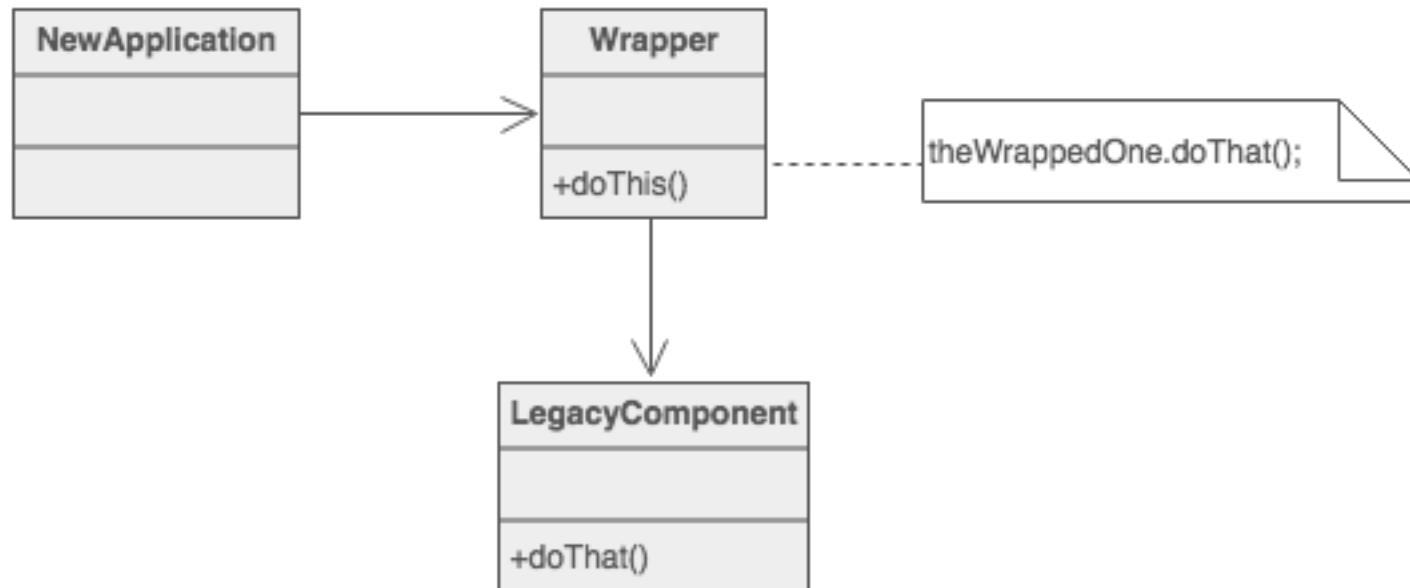
Name	Adapter
Intent	<ul style="list-style-type: none"><li>• To <u>convert</u> the interface of a class into another interface</li><li>• To let two or more classes with incompatible interfaces <u>work together</u></li><li>• To <u>wrap</u> an existing class with a new one</li><li>• To have a sort of <u>homogenous interface</u> that masks the diversity of some set of various objects</li></ul>
Solution	[see next slide]
Consequences	<ul style="list-style-type: none"><li>• You have a <u>single class</u> which is responsible to join functionalities of independent or incompatible classes</li></ul>
Examples	<ul style="list-style-type: none"><li>• A wrapper of a complex Java library in order to expose only the APIs that you need in your system</li><li>• A class uses a naïve coordinate reference system, but you want to mask it and show a more straightforward one to the rest of your system</li></ul>

# Adapter - solution



# Adapter - example

---



# Adapter - implementation

```
public class Wrapper {  
  
    // this is the wrapped object  
    private LegacyComponent legacyComponent;  
  
    // constructor  
    public Wrapper (LegacyComponent instance) {  
        this.legacyComponent = instance;  
    }  
  
    // call to the wrapped method  
    public int doThis() {  
        int result = 0;  
        float value = this.legacyComponent.doThat();  
        // ...here you transform value into an integer...  
        return result;  
    }  
}
```



---

# Behavioural design patterns

# Behavioural design patterns (1/2)

- Observer studied in this course
  - A way of notifying change to a number of classes
- Chain of responsibility
  - A way of passing a request between a chain of objects
- Command
  - Encapsulate a command request as an object
- Interpreter
  - A way to include language elements in a program
- Iterator
  - Sequentially access the elements of a collection
- Mediator
  - Defines simplified communication between classes



# Behavioural design patterns (2/2)

---

- Memento
  - Capture and restore an object's internal state
- Null Object
  - Designed to act as a default value of an object
- State
  - Alter an object's behavior when its state changes
- Strategy
  - Encapsulates an algorithm inside a class
- Template method
  - Defer the exact steps of an algorithm to a subclass
- Visitor
  - Defines a new operation to a class without change

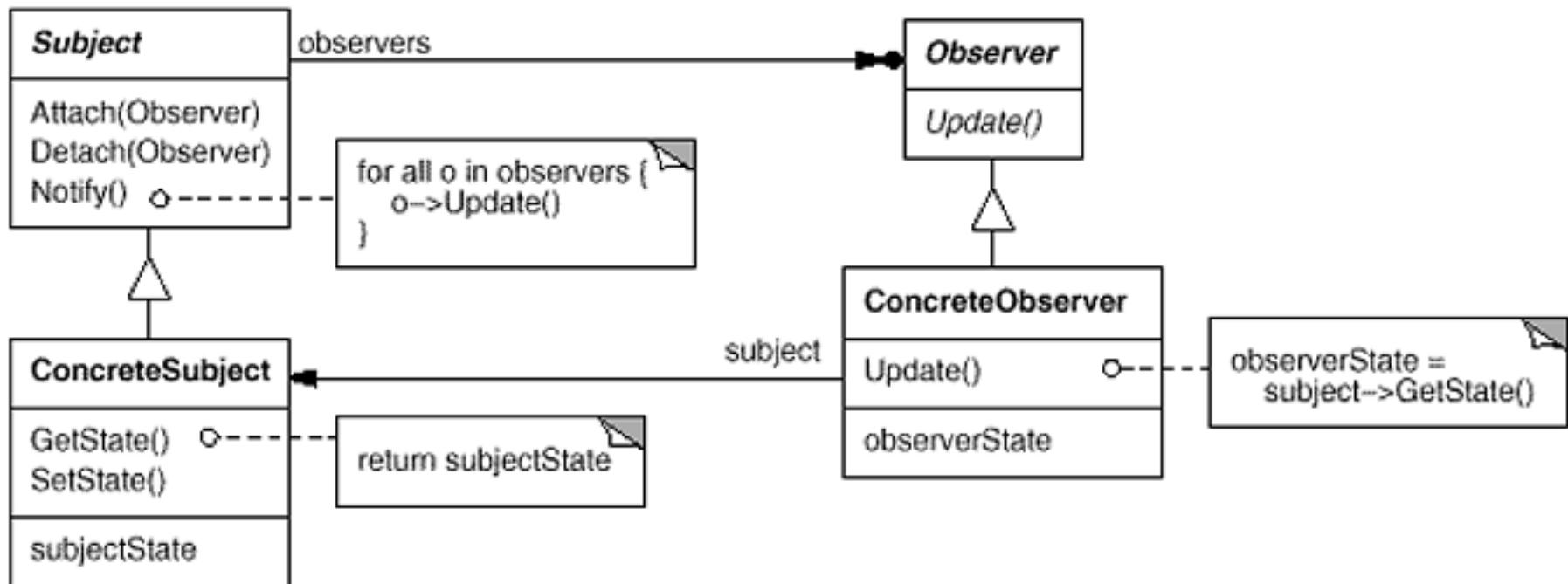


# Observer

Name	Observer
Intent	<ul style="list-style-type: none"><li>To let one or more objects be notified of state changes in other objects within the system</li><li>When one object changes state, all its dependents are <u>notified and updated automatically</u></li></ul>
Solution	[see next slide]
Consequences	<ul style="list-style-type: none"><li>Support for <u>broadcast communication</u></li><li><u>State changes</u> in one or more objects should <u>trigger behavior</u> in other objects</li><li>You can <u>reuse</u> subjects without reusing their observers, and vice versa</li><li>You can add or remove observers without modifying the subjects</li></ul>
Examples	<ul style="list-style-type: none"><li>A central station can issue commands to a subset of rovers moving in the environment</li><li>When a rover finishes its tasks it can notify the central station</li><li>etc.</li></ul>



# Observer - solution



# Observer – implementation (1/5)

---

```
1 public abstract class Observer {  
2     protected Subject subject;  
3     public abstract void update();  
4 }  
5  
6 }
```

## Observer – implementation (2/5)

---

```
1 public abstract class Observer {  
2     protected Subject subject;  
3     public abstract void update();  
4 }  
5  
6 }
```

```
9 ▼ class MappingRover extends Observer {  
10  
11     // in the constructor you specify the subject it observes  
12 ▼     public MappingRover(Subject subject) {  
13         this.subject = subject;  
14         subject.attach(this);  
15     }  
16  
17     // Observers "pull" information  
18 ▼     public void update() {  
19         if(this.subject.getState() == 0) {  
20             // .. let's go to map the environment!  
21         } else {  
22             // .. ohhh, I have to come back home.  
23         }  
24     }  
25 }
```

## Observer – implementation (3/5)

```
1 public abstract class Observer {  
2  
3     protected Subject subject;  
4  
5     public abstract void update();  
6  
7 }
```

```
9 ▼ class MappingRover extends Observer {  
10  
11     // in the constructor you specify the subject it observes  
12     public MappingRover(Subject subject) {  
13         this.subject = subject;  
14         subject.attach(this);  
15     }  
16  
17     // Observers "push" information  
18     public void update() {  
19         if(this.subject.getState() == 0) {  
20             // .. let's go to take some pictures!  
21         } else {  
22             // .. ohhh, it's not my turn yet  
23         }  
24     }  
25 }  
  
27 ▼ class CameraRover extends Observer {  
28  
29     // in the constructor you specify the subject it observes  
30     public CameraRover(Subject subject) {  
31         this.subject = subject;  
32         subject.attach(this);  
33     }  
34  
35     // Observers "pull" information  
36     public void update() {  
37         if(this.subject.getState() == 0) {  
38             // .. it's not my turn yet  
39         } else {  
40             // .. let's go to take some pictures!  
41         }  
42     }  
43 }
```

## Observer – implementation (4/5)

```
37 ▼ public abstract class Subject {  
38  
39     private List<Observer> observers = new ArrayList<Observer>();  
40     private int state;  
41  
42     public int getState() {  
43         return this.state;  
44     }  
45  
46 ▼ public void setState(int state) {  
47     this.state = state;  
48     this.notifyAllObservers();  
49 }  
50  
51 public void attach(Observer observer){  
52     this.observers.add(observer);  
53 }  
54  
55 public void detach(Observer observer){  
56     this.observers.remove(test.indexOf(observer));  
57 }  
58  
59 ▼ public void notifyAllObservers(){  
60     for (Observer observer : this.observers) {  
61         observer.update();  
62     }  
63 }  
64 }
```

```
66     public class CentralStation extends Subject {  
67  
68 }
```

# Observer – implementation (5/5)

---

```
78▼ public static void main(String[] args) {  
79    CentralStation cs = new CentralStation();  
80    cs.setState(0);  
81  
82    MappingRover rover1 = new MappingRover(cs);  
83    CameraRover rover2 = new CameraRover(cs);  
84    CameraRover rover3 = new CameraRover(cs);  
85  
86    cs.setState(1);  
87}
```

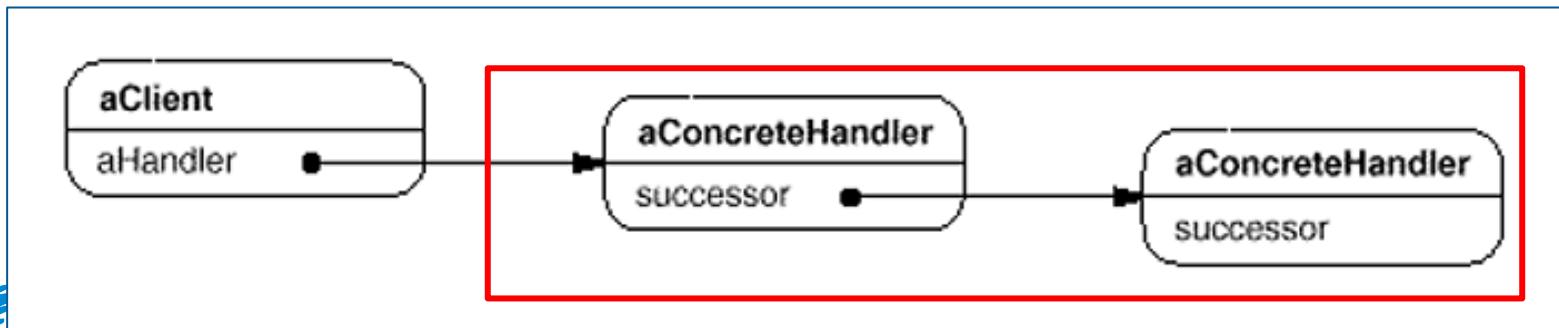
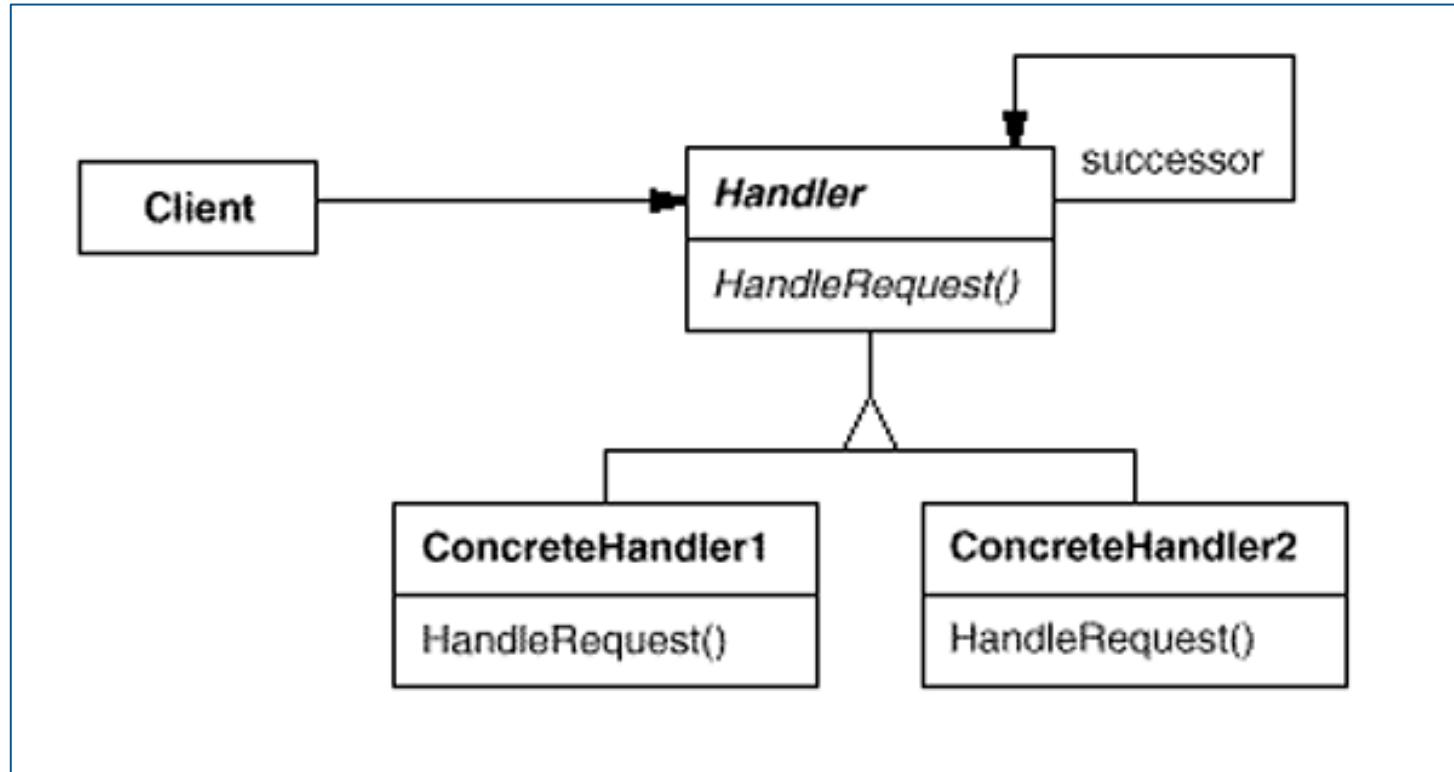


# Chain of responsibility

Name	Chain of responsibility
Intent	<ul style="list-style-type: none"><li>• Avoid coupling the <u>sender</u> of a request to its <u>receiver</u> by giving <u>more than one object</u> a chance to handle the request</li><li>• Chain the receiving objects and pass the request along the chain <u>until an object handles it</u><ul style="list-style-type: none"><li>• → it is sequential (Observer is in parallel)</li></ul></li></ul>
Solution	[see next slide]
Consequences	<ul style="list-style-type: none"><li>• <u>Reduced coupling</u> between objects<ul style="list-style-type: none"><li>• every objects just needs to know its successor</li></ul></li><li>• More than one object may handle a request, and the <u>handler is not known a priori</u></li><li>• You can <u>change responsibilities</u> by changing the chain at run-time</li><li>• A request can also go <u>unhandled</u></li></ul>
Examples	<ul style="list-style-type: none"><li>• A central station keeps a pool of idle robots and assign tasks without knowing the exact number of available robots</li></ul>



# Chain of responsibility - solution



this chain goes on until  
the request is handled



# Chain of responsibility – implementation (1/4)

```
1 ▼ public class Task {  
2  
3     private Coordinate coords;  
4     private RequestEnum request;  
5  
6     public Coordinate getcoords() {  
7         return this.coords;  
8     }  
9  
10    public setCoords(Coordinate coords) {  
11        this.coords = coords;  
12    }  
13  
14    public RequestEnum getRequest() {  
15        return this.request;  
16    }  
17  
18    public setRequest(RequestEnum request) {  
19        this.request = request;  
20    }  
21 }
```

```
23    public enum RequestEnum {  
24        PICTURE, MAP;  
25    }  
26  
27 ▼ public class Coordinate {  
28  
29     private float lat;  
30     private float lon;  
31  
32     public float getLat() {  
33         return this.lat;  
34     }  
35  
36     public setLat(float lat) {  
37         this.lat = lat;  
38     }  
39  
40     public float getLon() {  
41         return this.lon;  
42     }  
43  
44     public setLon(float lon) {  
45         this.lon = lon;  
46     }  
47 }
```



## Chain of responsibility – implementation (2/4)

---

```
49 ▼ public abstract class TaskHandler {  
50  
51     TaskHandler successor;  
52  
53     public void setSuccessor(TaskHandler successor) {  
54         this.successor = successor;  
55     }  
56  
57     public abstract void handleRequest(Task task);  
58  
59 }
```



## Chain of responsibility – implementation (3/4)

```
49▼ public abstract class TaskHandler {  
50  
51▼ public class CameraRover extends TaskHandler {  
52  
53▼     public void handleRequest(Task task) {  
54▼         if (task.request == RequestEnum.PICTURE) {  
55             System.out.println("Smile!");  
56             // take a picture  
57▼         } else {  
58             System.out.println("Sorry, I do not know how to do it.");  
59             if (successor != null) {  
60                 successor.handleRequest(request);  
61             }  
62         }  
63     }  
64 }  
65 }
```



## Chain of responsibility – implementation (3/4)

```
49▼ public abstract class TaskHandler {  
50  
51▼ public class CameraRover extends TaskHandler {  
52  
53▼     public void handleRequest(Task task) {  
54▼         if (task.request == RequestEnum.PICTURE) {  
55             System.out.println("Smile!");  
56             // take a picture  
57▼     } else {  
58         System.out.println("Sorry, I do not know how to do it.");  
59         if (successor != null) {  
60             successor.handleRequest(request);  
61         } } } }  
62  
63▼     public class MapRover extends TaskHandler {  
64  
65▼         public void handleRequest(Task task) {  
66▼             if (task.request == RequestEnum.MAP) {  
67                 System.out.println("Let's go!");  
68                 // start mapping  
69▼             } else {  
70                 System.out.println("Sorry, I do not know how to do it.");  
71                 if (successor != null) {  
72                     successor.handleRequest(request);  
73                 } } } }  
74 } }
```



## Chain of responsibility – implementation (4/4)

```
91 ▼ public static TaskHandler setUpChain() {  
92     // we create the hanlders  
93     MapRover mapper = new MapRover();  
94     CameraRover photographer1 = new CameraRover();  
95     CameraRover photographer2 = new CameraRover();  
96     CameraRover photographer3 = new CameraRover();  
97  
98     // let's build the chain  
99     mapper.setSuccessor(photographer1);  
100    photographer1.setSuccessor(photographer2);  
101    photographer2.setSuccessor(photographer3);  
102  
103    return mapper;  
104 }
```

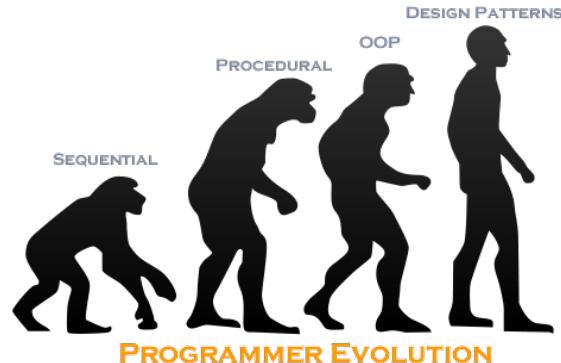
```
106 ▼ public static void main(String[] args) {  
107     TaskHandler chain = setUpChain();  
108  
109     // here I don't care about who is able to do what, they know.  
110     chain.handleRequest(...);  
111  
112  
113  
114  
115  
116 }
```



# What this lecture means to you?

---

- Design patterns = solutions to common software design problems
- They are not prescriptive specifications for software
  - Do not memorize them → it is more important that you understand when they are needed and why
- They are not always needed
  - Do not apply design patterns to a trivial solution → you will overcomplicate your code and lead to maintainability issues



---

# Take-home exercise

# Exercise

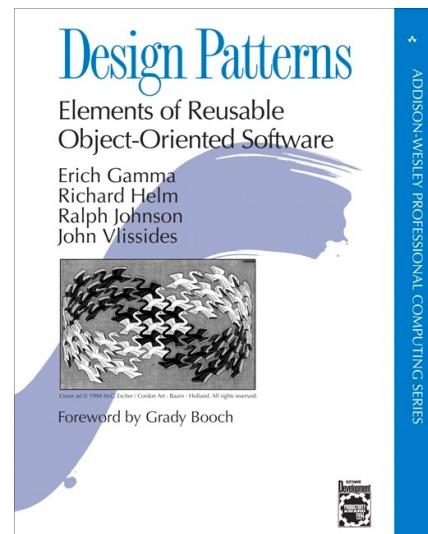
---

1. Reconsider the FB class diagram
2. Identify a potential issue
3. Apply a design pattern for solving it

# Readings

---

- [https://en.wikipedia.org/wiki/Design\\_Patterns](https://en.wikipedia.org/wiki/Design_Patterns)
- [https://sourcemaking.com/design\\_patterns](https://sourcemaking.com/design_patterns)
- <http://www.blackwasp.co.uk/gofpatterns.aspx>
- [optional] Detailed info on the GoF book



# Chapter 16

---

## ■ Pattern-Based Design

*Slide Set to accompany*

*Software Engineering: A Practitioner's Approach, 8/e*  
**by Roger S. Pressman and Bruce R. Maxim**

Slides copyright © 1996, 2001, 2005, 2009, 2014 by Roger S. Pressman

***For non-profit educational use only***

May be reproduced ONLY for student use at the university level when used in conjunction with *Software Engineering: A Practitioner's Approach, 8/e*. Any other reproduction or use is prohibited without the express written permission of the author.

All copyright information MUST appear if these slides are posted on a website for student use.

# Design Patterns

---

- Each of us has encountered a design problem and silently thought: *I wonder if anyone has developed a solution to for this?*
  - What if there was a standard way of describing a problem (so you could look it up), and an organized method for representing the solution to the problem?
- *Design patterns* are a codified method for describing problems and their solution allows the software engineering community to capture design knowledge in a way that enables it to be reused.

# Design Patterns

---

- *Each pattern describes a problem that occurs over and over again in our environment and then describes the core of the solution to that problem in such a way that you can use the solution a million times over without ever doing it the same way twice.*
  - Christopher Alexander, 1977
- “a three-part rule which expresses a relation between a certain context, a problem, and a solution.”

# Basic Concepts

---

- *Context* allows the reader to understand the environment in which the problem resides and what solution might be appropriate within that environment.
- A set of requirements, including limitations and constraints, acts as a *system of forces* that influences how
  - the problem can be interpreted within its context and
  - how the solution can be effectively applied.

# Effective Patterns

---

- Coplien [Cop05] characterizes an effective design pattern in the following way:
  - *It solves a problem:* Patterns capture solutions, not just abstract principles or strategies.
  - *It is a proven concept:* Patterns capture solutions with a track record, not theories or speculation.
  - *The solution isn't obvious:* Many problem-solving techniques (such as software design paradigms or methods) try to derive solutions from first principles. The best patterns *generate* a solution to a problem indirectly--a necessary approach for the most difficult problems of design.
  - *It describes a relationship:* Patterns don't just describe modules, but describe deeper system structures and mechanisms.
  - *The pattern has a significant human component (minimize human intervention):* All software serves human comfort or quality of life; the best patterns explicitly appeal to aesthetics and utility.

# Generative Patterns

---

- *Generative patterns* describe an important and repeatable aspect of a system and then provide us with a way to build that aspect within a system of forces that are unique to a given context.
- A collection of generative design patterns could be used to “generate” an application or computer-based system whose architecture enables it to adapt to change.

# Kinds of Patterns

---

- *Architectural patterns* describe broad-based design problems that are solved using a structural approach.
- *Data patterns* describe recurring data-oriented problems and the data modeling solutions that can be used to solve them.
- *Component patterns* (also referred to as *design patterns*) address problems associated with the development of subsystems and components, the manner in which they communicate with one another, and their placement within a larger architecture
- *Interface design patterns* describe common user interface problems and their solution with a system of forces that includes the specific characteristics of end-users.
- *WebApp patterns* address a problem set that is encountered when building WebApps and often incorporates many of the other patterns categories just mentioned.

# Kinds of Patterns

---

- ***Creational patterns*** focus on the “creation, composition, and representation of objects, e.g.,
  - [Abstract factory pattern](#): centralize decision of what [factory](#) to instantiate
  - [Factory method pattern](#): centralize creation of an object of a specific type choosing one of several implementations
- ***Structural patterns*** focus on problems and solutions associated with how classes and objects are organized and integrated to build a larger structure, e.g.,
  - [Adapter pattern](#): 'adapts' one interface for a class into one that a client expects
  - [Aggregate pattern](#): a version of the [Composite pattern](#) with methods for aggregation of children
- ***Behavioral patterns*** address problems associated with the assignment of responsibility between objects and the manner in which communication is effected between objects, e.g.,
  - [Chain of responsibility pattern](#): Command objects are handled or passed on to other objects by logic-containing processing objects
  - [Command pattern](#): Command objects encapsulate an action and its parameters

# Frameworks

---

- Patterns themselves may not be sufficient to develop a complete design.
  - In some cases it may be necessary to provide an implementation-specific skeletal infrastructure, called a *framework*, for design work.
  - That is, you can select a “*reusable mini-architecture* that provides the generic structure and behavior for a family of software abstractions, along with a context ... which specifies their collaboration and use within a given domain.” [Amb98]
- A **framework is not an architectural pattern**, but rather a skeleton with a collection of “**plug points**” (also called *hooks* and *slots*) that enable it to be adapted to a specific problem domain.
  - The plug points enable you to integrate problem specific classes or functionality within the skeleton.

# Describing a Pattern

---

- **Pattern name**—describes the essence of the pattern in a short but expressive name
- **Problem**—describes the problem that the pattern addresses
- **Motivation**—provides an example of the problem
- **Context**—describes the environment in which the problem resides including application domain
- **Forces**—lists the system of forces that affect the manner in which the problem must be solved; includes a discussion of limitation and constraints that must be considered
- **Solution**—provides a detailed description of the solution proposed for the problem
- **Intent**—describes the pattern and what it does
- **Collaborations**—describes how other patterns contribute to the solution
- **Consequences**—describes the potential trade-offs that must be considered when the pattern is implemented and the consequences of using the pattern
- **Implementation**—identifies special issues that should be considered when implementing the pattern
- **Known uses**—provides examples of actual uses of the design pattern in real applications
- **Related patterns**—cross-references related design patterns

# Pattern Languages

---

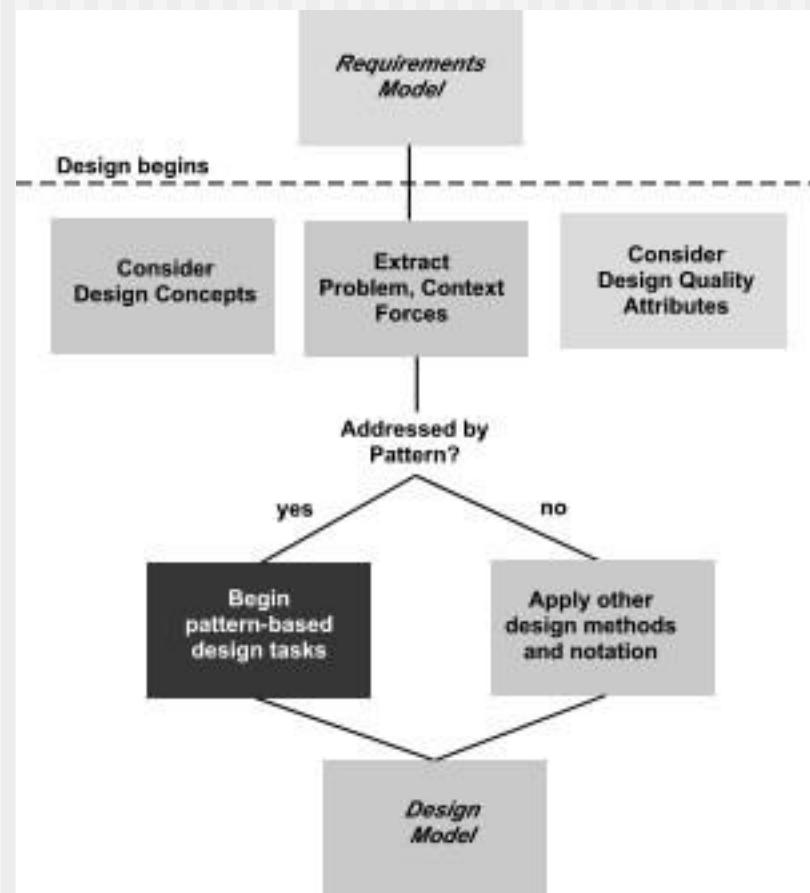
- A *pattern language* encompasses a collection of patterns
  - each described using a standardized template (Section 12.1.3) and
  - interrelated to show how these patterns collaborate to solve problems across an application domain.
- a pattern language is analogous to a hypertext instruction manual for problem solving in a specific application domain.
  - The problem domain under consideration is first described hierarchically, beginning with broad design problems associated with the domain and then refining each of the broad problems into lower levels of abstraction.

# Pattern-Based Design

---

- A software designer begins with a requirements model (either explicit or implied) that presents an abstract representation of the system.
- The requirements model describes the problem set, establishes the context, and identifies the system of forces that hold sway.
- Then ...

# Pattern-Based Design



# Thinking in Patterns

---

- Shalloway and Trott [Sha05] suggest the following approach that enables a designer to think in patterns:
  - 1. Be sure you understand the big picture—the context in which the software to be built resides. The requirements model should communicate this to you.
  - 2. Examining the big picture, extract the patterns that are present at that level of abstraction.
  - 3. Begin your design with ‘big picture’ patterns that establish a context or skeleton for further design work.
  - 4. “Work inward from the context” [Sha05] looking for patterns at lower levels of abstraction that contribute to the design solution.
  - 5. Repeat steps 1 to 4 until the complete design is fleshed out.
  - 6. Refine the design by adapting each pattern to the specifics of the software you’re trying to build.

# Design Tasks—I

---

- Examine the requirements model and develop a problem hierarchy.
- Determine if a reliable pattern language has been developed for the problem domain.
- Beginning with a broad problem, determine whether one or more architectural patterns are available for it.
- Using the collaborations provided for the architectural pattern, examine subsystem or component level problems and search for appropriate patterns to address them.
- Repeat steps 2 through 5 until all broad problems have been addressed.

# Design Tasks—II

---

- If user interface design problems have been isolated (this is almost always the case), search the many user interface design pattern repositories for appropriate patterns.
- Regardless of its level of abstraction, if a pattern language and/or patterns repository or individual pattern shows promise, compare the problem to be solved against the existing pattern(s) presented.
- Be certain to refine the design as it is derived from patterns using design quality criteria as a guide.

# Pattern Organizing Table

	Database	Application	Implementation	Infrastructure
<i>Data/Content</i>				
Problem statement ...	PatternName(s)		PatternName(s)	
Problem statement ...		PatternName(s)		PatternName(s)
Problem statement ...	PatternName(s)			PatternName(s)
<i>Architecture</i>				
Problem statement ...		PatternName(s)		
Problem statement ...		PatternName(s)		PatternName(s)
Problem statement ...				
<i>Component-level</i>				
Problem statement ...		PatternName(s)	PatternName(s)	
Problem statement ...				PatternName(s)
Problem statement ...	PatternName(s)		PatternName(s)	
<i>User Interface</i>				
Problem statement ...		PatternName(s)	PatternName(s)	
Problem statement ...		PatternName(s)	PatternName(s)	
Problem statement ...		PatternName(s)	PatternName(s)	

# Common Design Mistakes

---

- Not enough time has been spent to understand the underlying problem, its context and forces, and as a consequence, you select a pattern that looks right, but is inappropriate for the solution required.
- Once the wrong pattern is selected, you refuse to see your error and force fit the pattern.
- In other cases, the problem has forces that are not considered by the pattern you've chosen, resulting in a poor or erroneous fit.
- Sometimes a pattern is applied too literally and the required adaptations for your problem space are not implemented.

# Architectural Patterns

---

- Example: every house (and every architectural style for houses) employs a **Kitchen** pattern.
- The **Kitchen** pattern and patterns it collaborates with address problems associated with the storage and preparation of food, the tools required to accomplish these tasks, and rules for placement of these tools relative to workflow in the room.
- In addition, the pattern might address problems associated with counter tops, lighting, wall switches, a central island, flooring, and so on.
- Obviously, there is more than a single design for a kitchen, often dictated by the context and system of forces. But every design can be conceived within the context of the ‘solution’ suggested by the **Kitchen** pattern.

# Patterns Repositories

---

- There are many sources for design patterns available on the Web. Some patterns can be obtained from individually published pattern languages, while others are available as part of a patterns portal or patterns repository.
- A list of patterns repositories is presented in the sidebar near Section 12.3

# Component-Level Patterns

---

- Component-level design patterns provide a proven solution that addresses one or more sub-problems extracted from the requirement model.
- In many cases, design patterns of this type focus on some functional element of a system.
- For example, the **SafeHomeAssured.com** application must address the following design sub-problem: *How can we get product specifications and related information for any SafeHome device?*

# Component-Level Patterns

---

- Having enunciated the sub-problem that must be solved, consider context and the system of forces that affect the solution.
- Examining the appropriate requirements model use case, the specification for a *SafeHome* device (e.g., a security sensor or camera) is used for informational purposes by the consumer.
  - However, other information that is related to the specification (e.g., pricing) may be used when e-commerce functionality is selected.
- The solution to the sub-problem involves a **search**. Since searching is a very common problem, it should come as no surprise that there are many search-related patterns.
- See Section 12.4

# User Interface (UI) Patterns

---

- **Whole UI.** Provide design guidance for top-level structure and navigation throughout the entire interface.
- **Page layout.** Address the general organization of pages (for Websites) or distinct screen displays (for interactive applications)
- **Forms and input.** Consider a variety of design techniques for completing form-level input.
- **Tables.** Provide design guidance for creating and manipulating tabular data of all kinds.
- **Direct data manipulation.** Address data editing, modification, and transformation.
- **Navigation.** Assist the user in navigating through hierarchical menus, Web pages, and interactive display screens.
- **Searching.** Enable content-specific searches through information maintained within a Web site or contained by persistent data stores that are accessible via an interactive application.
- **Page elements.** Implement specific elements of a Web page or display screen.
- **E-commerce.** Specific to Web sites, these patterns implement recurring elements of e-commerce applications.

# WebApp Patterns

---

- **Information architecture patterns** relate to the overall structure of the information space, and the ways in which users will interact with the information.
- **Navigation patterns** define navigation link structures, such as hierarchies, rings, tours, and so on.
- **Interaction patterns** contribute to the design of the user interface. Patterns in this category address how the interface informs the user of the consequences of a specific action; how a user expands content based on usage context and user desires; how to best describe the destination that is implied by a link; how to inform the user about the status of an on-going interaction, and interface related issues.
- **Presentation patterns** assist in the presentation of content as it is presented to the user via the interface. Patterns in this category address how to organize user interface control functions for better usability; how to show the relationship between an interface action and the content objects it affects, and how to establish effective content hierarchies.
- **Functional patterns** define the workflows, behaviors, processing, communications, and other algorithmic elements within a WebApp.

# Design Granularity

---

- When a problem involves “big picture” issues, attempt to develop solutions (and use relevant patterns) that focus on the big picture.
- Conversely, when the focus is very narrow (e.g., uniquely selecting one item from a small set of five or fewer items), the solution (and the corresponding pattern) is targeted quite narrowly.
- In terms of the level of granularity, patterns can be described at the following levels:

# Design Granularity

---

- **Architectural patterns.** This level of abstraction will typically relate to patterns that define the overall structure of the WebApp, indicate the relationships among different components or increments, and define the rules for specifying relationships among the elements (pages, packages, components, subsystems) of the architecture.
- **Design patterns.** These address a specific element of the design such as an aggregation of components to solve some design problem, relationships among elements on a page, or the mechanisms for effecting component to component communication. An example might be the *Broadsheet* pattern for the layout of a WebApp homepage.
- **Component patterns.** This level of abstraction relates to individual small-scale elements of a WebApp. Examples include individual interaction elements (e.g. radio buttons, text books), navigation items (e.g. how might you format links?) or functional elements (e.g. specific algorithms).

# Mobile User Interface Patterns

---

- Check-in screens
- Maps
- Popovers
- Sign-up flows
- Custom Tab Navigation
- Invitations

# Mobile App Design Patterns

---

- Active Objects
- Applications Controller
- Communicator
- Data Transfer Object
- Domain Model
- Entity Translator
- Lazy Acquisition
- Model-View-Comtroller
- Pagination
- Reliable Sessions
- Synchronization
- Transaction Script

# Chapter 4

---

## ■ Process Models

*Slide Set to accompany*

*Software Engineering: A Practitioner's Approach, 8/e*  
**by Roger S. Pressman and Bruce R. Maxim**

Slides copyright © 1996, 2001, 2005, 2009, 2014 by Roger S. Pressman

***For non-profit educational use only***

May be reproduced ONLY for student use at the university level when used in conjunction with *Software Engineering: A Practitioner's Approach, 8/e*. Any other reproduction or use is prohibited without the express written permission of the author.

All copyright information MUST appear if these slides are posted on a website for student use.

# Prescriptive Models

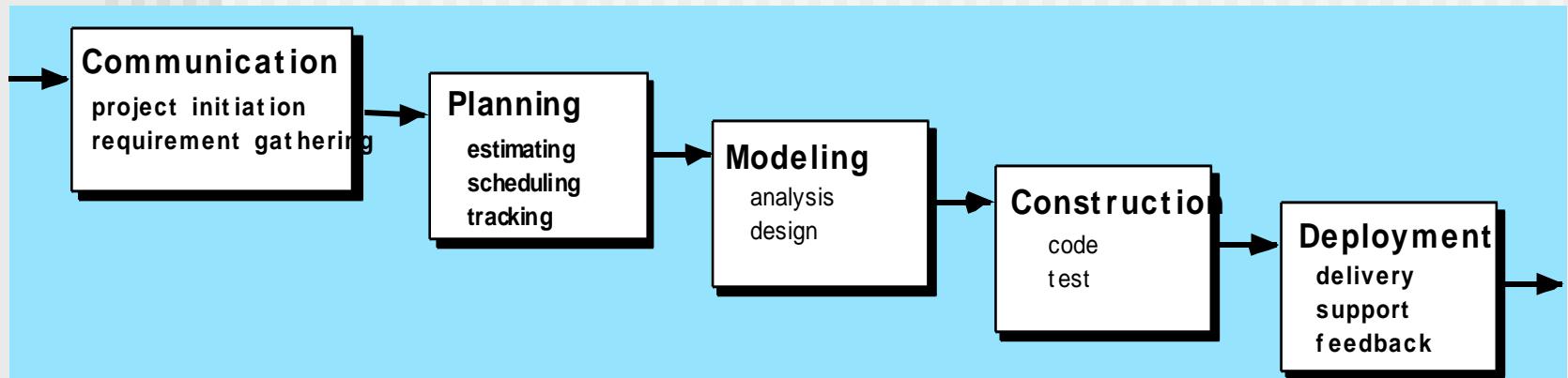
---

- Prescriptive process models advocate an orderly approach to software engineering

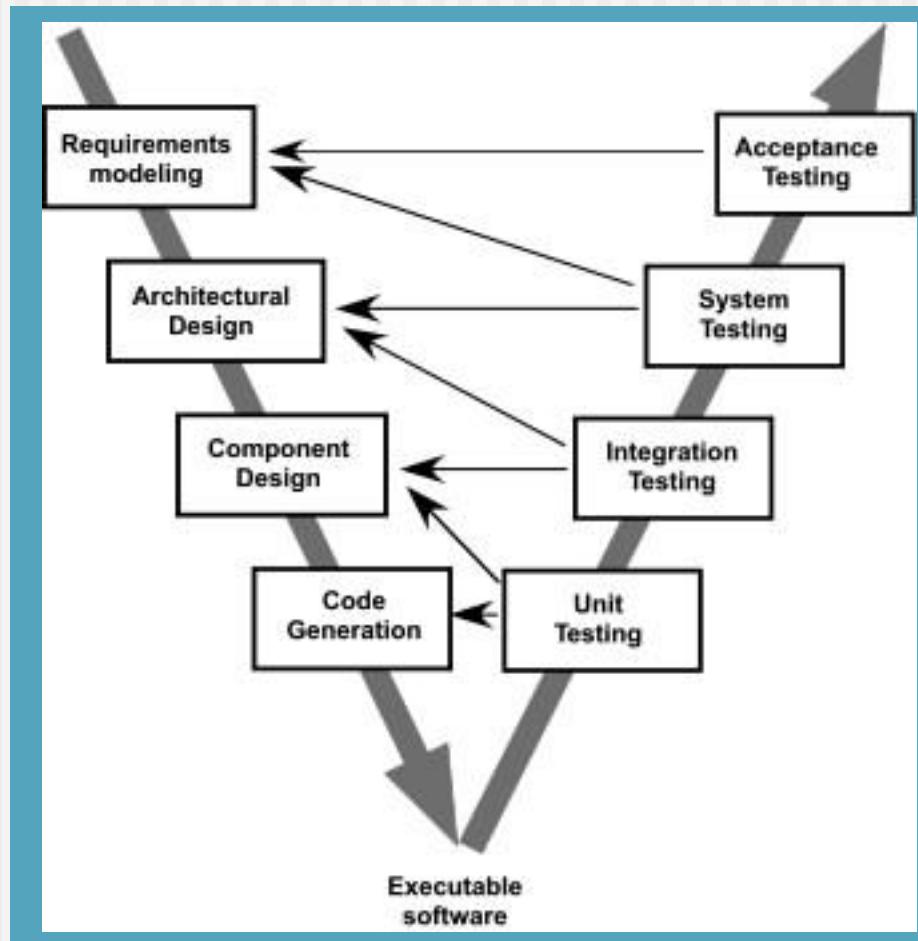
*That leads to a few questions ...*

- If prescriptive process models strive for structure and order, **are they inappropriate for a software world that thrives on change?**
- Yet, if we reject traditional process models (and the order they imply) and replace them with something less structured, **do we make it impossible to achieve coordination and coherence in software work?**

# The Waterfall Model

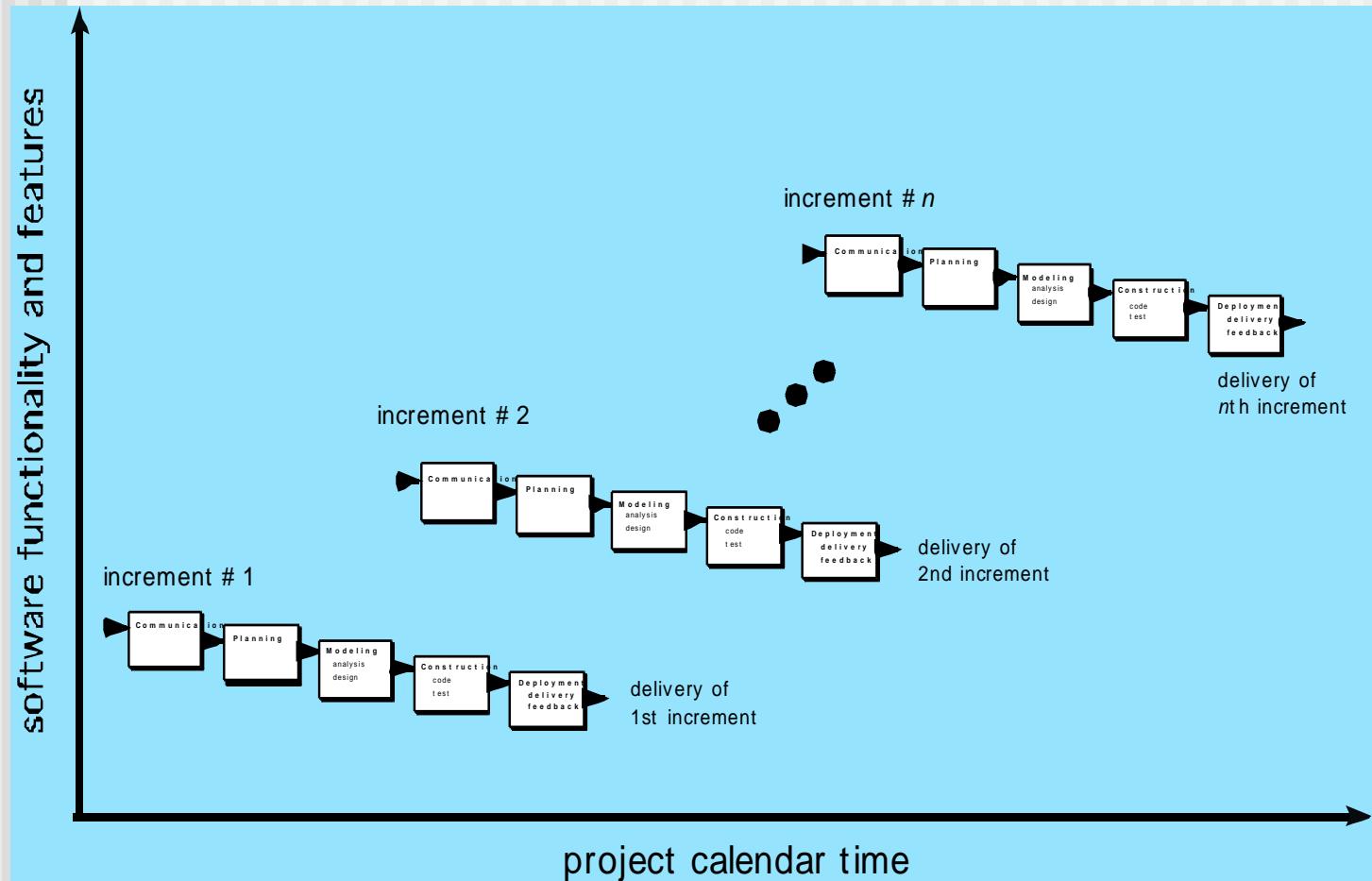


# The V-Model

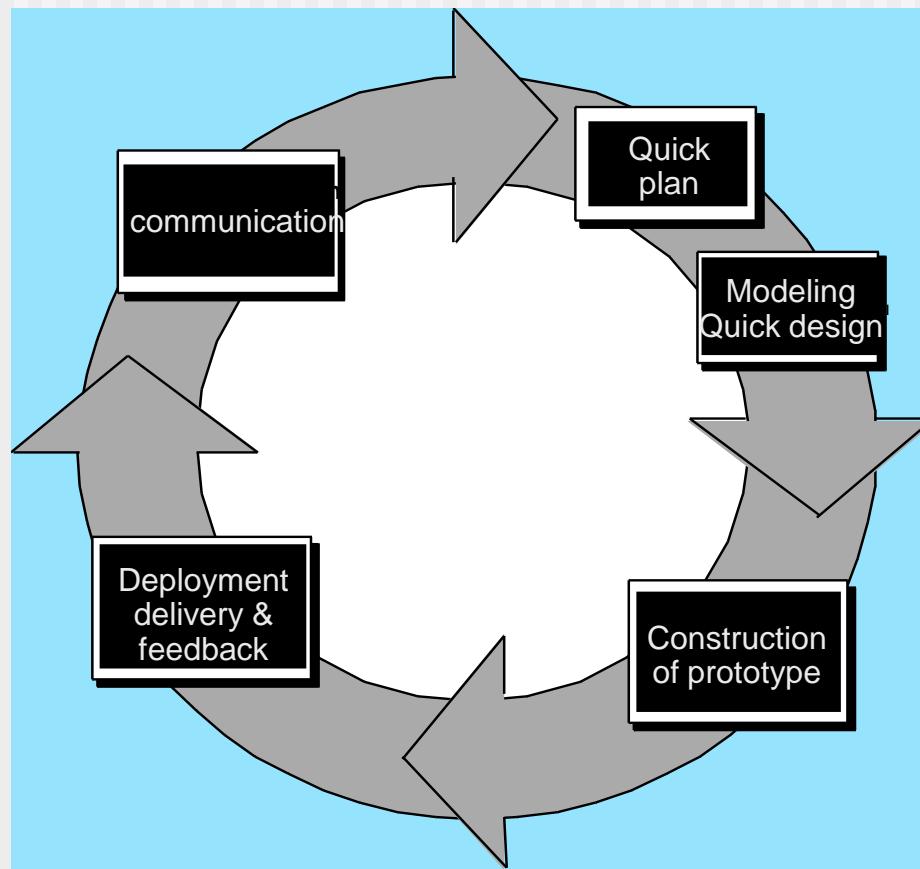


These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 8/e* (McGraw-Hill, 2014). Slides copyright 2014 by Roger Pressman.

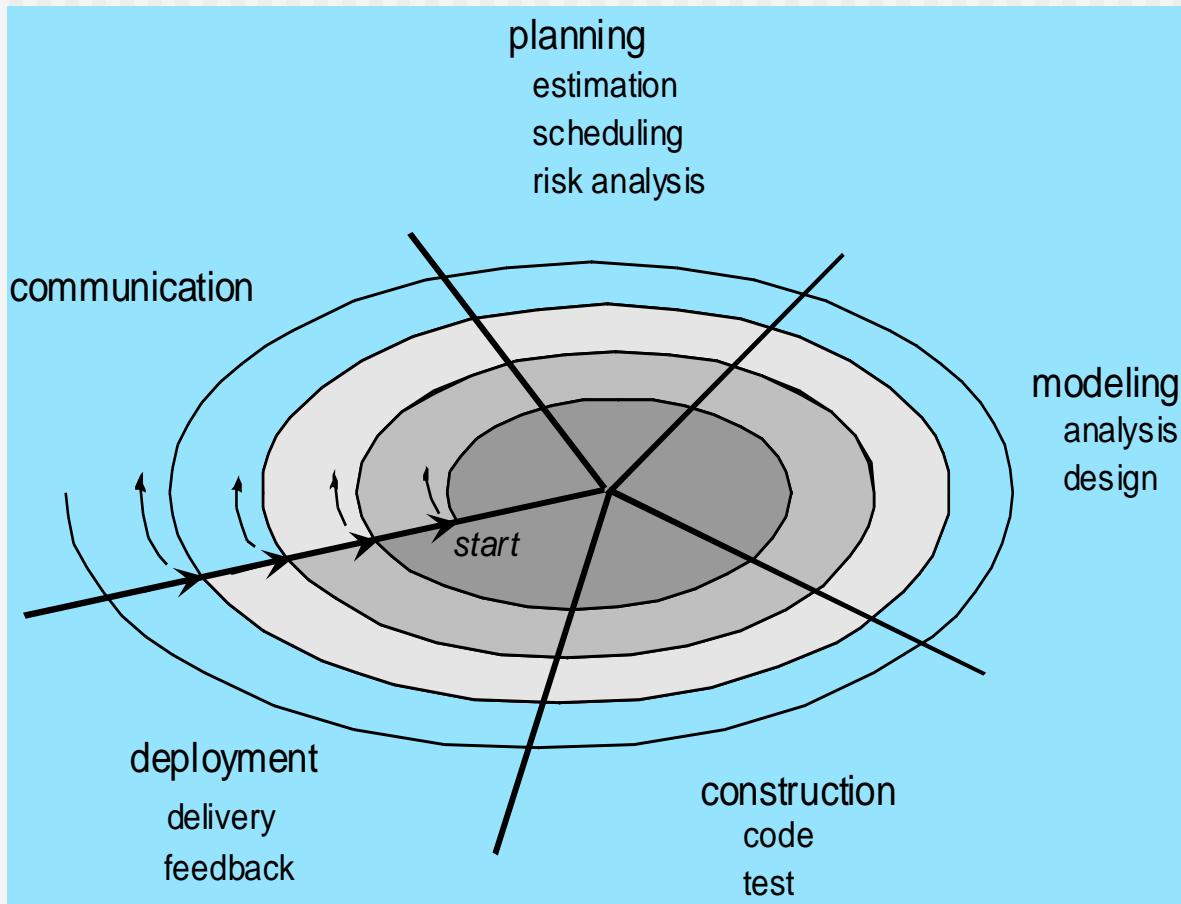
# The Incremental Model



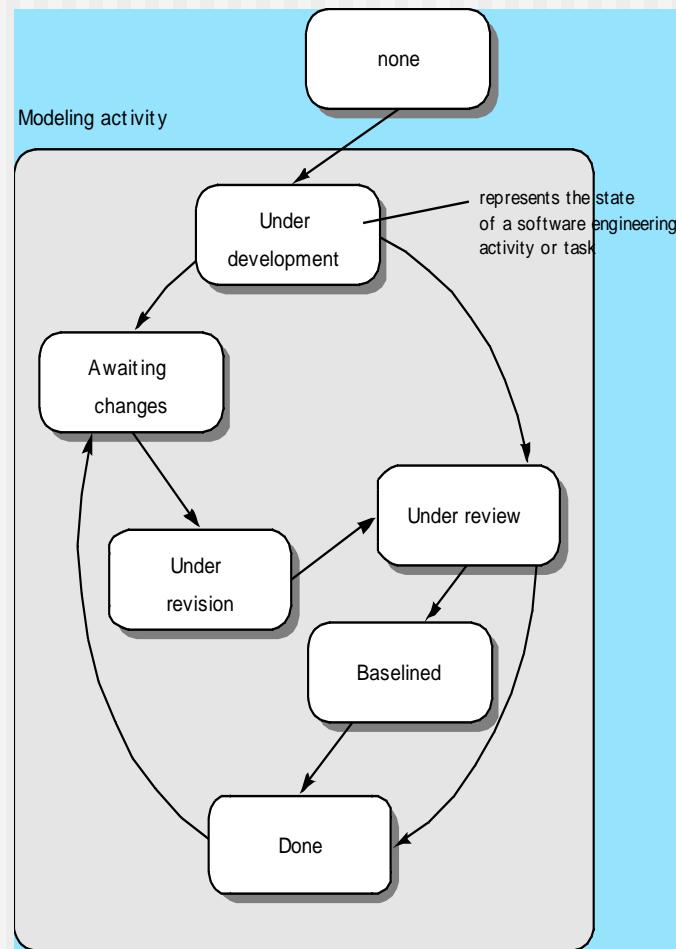
# Evolutionary Models: Prototyping



# Evolutionary Models: The Spiral



# Evolutionary Models: Concurrent



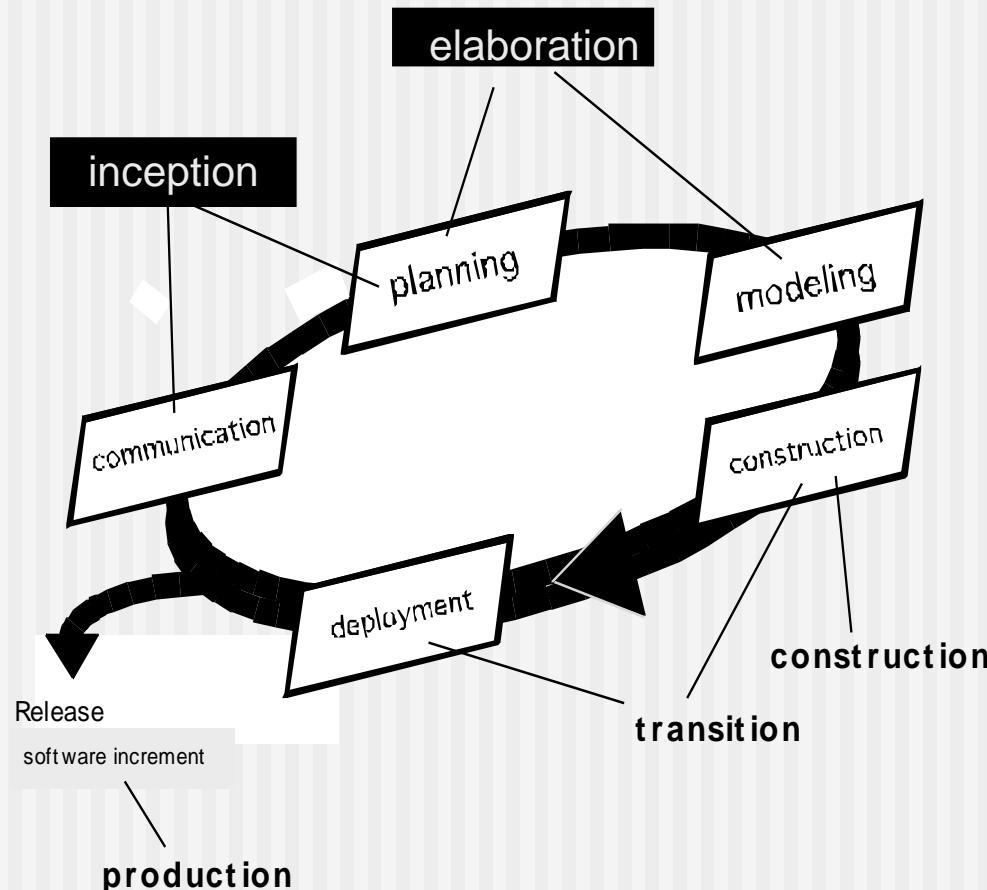
These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 8/e* (McGraw-Hill, 2014). Slides copyright 2014 by Roger Pressman.

# Still Other Process Models

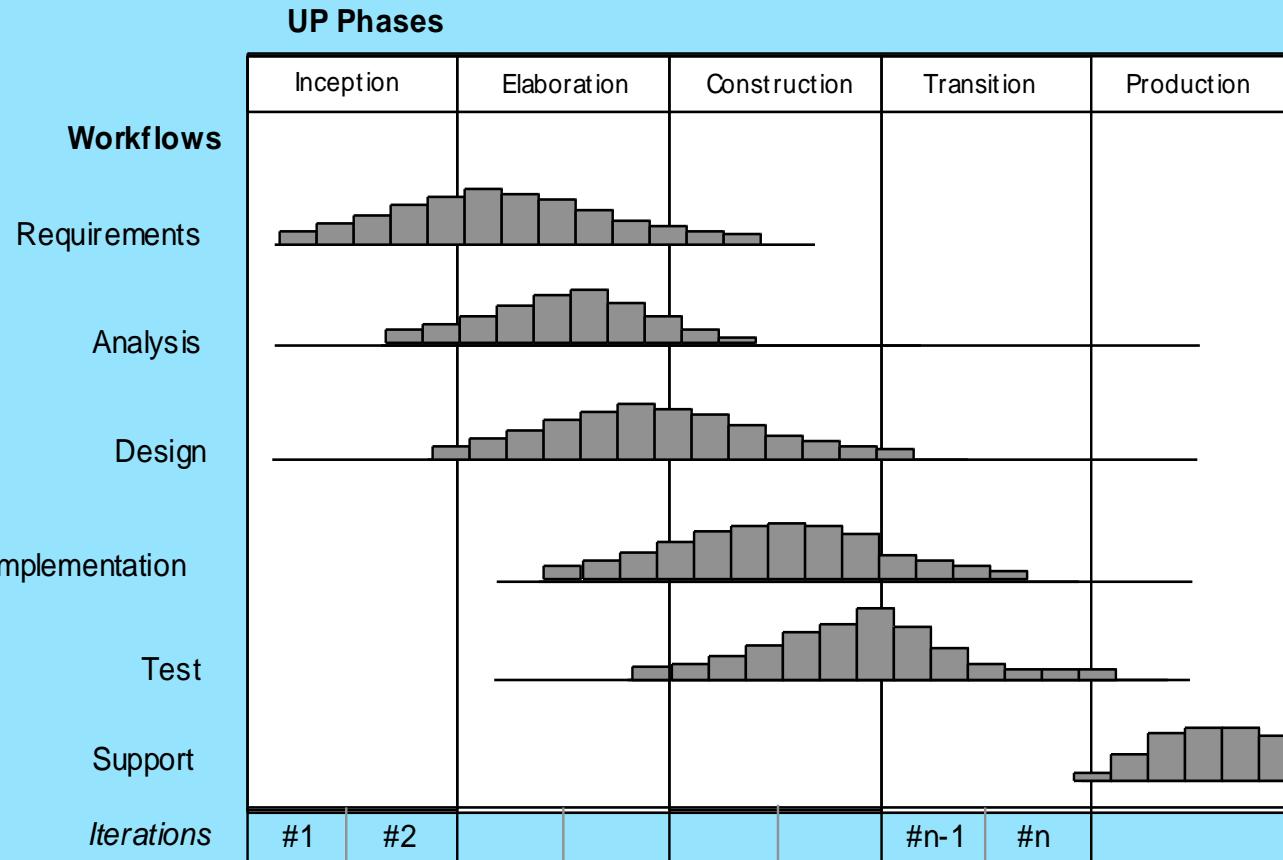
---

- **Component based development**—the process to apply when reuse is a development objective
- **Formal methods**—emphasizes the mathematical specification of requirements
- **AOSD**—provides a process and methodological approach for defining, specifying, designing, and constructing aspects
- **Unified Process**—a “use-case driven, architecture-centric, iterative and incremental” software process closely aligned with the Unified Modeling Language (UML)

# The Unified Process (UP)



# UP Phases



# UP Work Products

## Inception phase

Vision document  
Initial use-case model  
Initial project glossary  
Initial business case  
Initial risk assessment.  
Project plan,  
phases and iterations.  
Business model,  
if necessary.  
One or more prototypes

## Elaboration phase

Use-case model  
Supplementary requirements  
including non-functional  
Analysis model  
Software architecture  
Description.  
Executable architectural  
prototype.  
Preliminary design model  
Revised risk list  
Project plan including  
iteration plan  
adapted workflows  
milestones  
technical work products  
Preliminary user manual

## Construction phase

Design model  
Software components  
Integrated software  
increment  
Test plan and procedure  
Test cases  
Support documentation  
user manuals  
installation manuals  
description of current  
increment

## Transition phase

Delivered software increment  
Beta test reports  
General user feedback

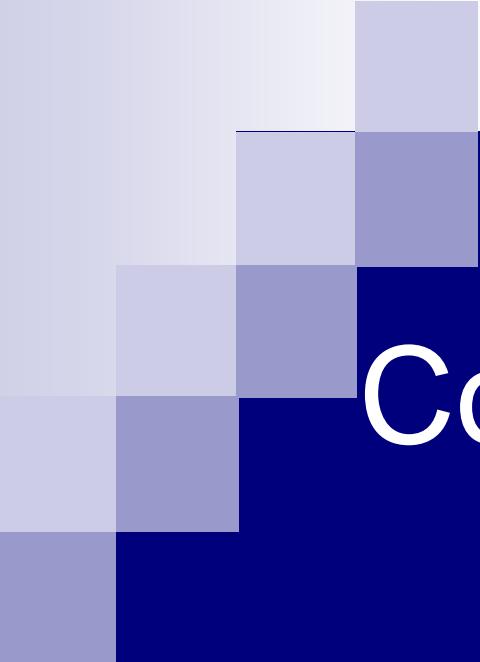
# Personal Software Process (PSP)

- **Planning.** This activity isolates requirements and develops both size and resource estimates. In addition, a defect estimate (the number of defects projected for the work) is made. All metrics are recorded on worksheets or templates. Finally, development tasks are identified and a project schedule is created.
- **High-level design.** An external specification is created for each component and a component design is created. Prototypes are built when uncertainty exists. All issues are recorded and tracked.
- **High-level design review.** Formal verification methods (Chapter 21) are applied to uncover errors in the design. Metrics are maintained for all important tasks and work results.
- **Development.** The component level design is refined and reviewed. Code is generated, reviewed, compiled, and tested. Metrics are maintained for all important tasks and work results.
- **Postmortem.** Using measures and metrics collected the effectiveness of the process is determined. If this is a large amount of data it should be analyzed statistically), Measures and metrics should provide guidance for modifying the process to improve its effectiveness.

# Team Software Process (TSP)

---

- Build self-directed teams that plan and track their work, establish goals, and own their processes and plans. These can be pure software teams or integrated product teams (IPT) of three to about 20 engineers.
- Show managers how to coach and motivate their teams and how to help them sustain peak performance.
- Accelerate software process improvement by making CMM Level 5 behavior normal and expected.
  - The Capability Maturity Model (CMM), a measure of the effectiveness of a software process, is discussed in Chapter 30.
- Provide improvement guidance to high-maturity organizations.
- Facilitate university teaching of industrial-grade team skills.



# Cohesion and Coupling

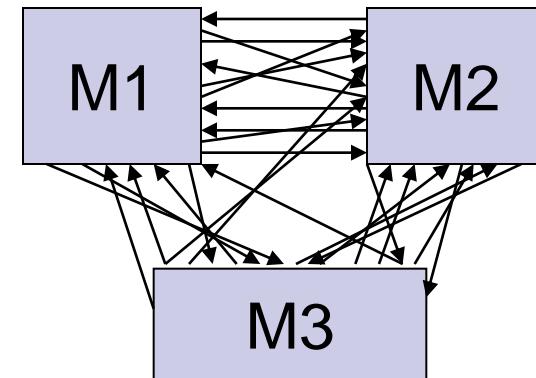
Chandravadan Prajapati

# Outline

- Cohesion
- Coupling

# Characteristics of Good Design

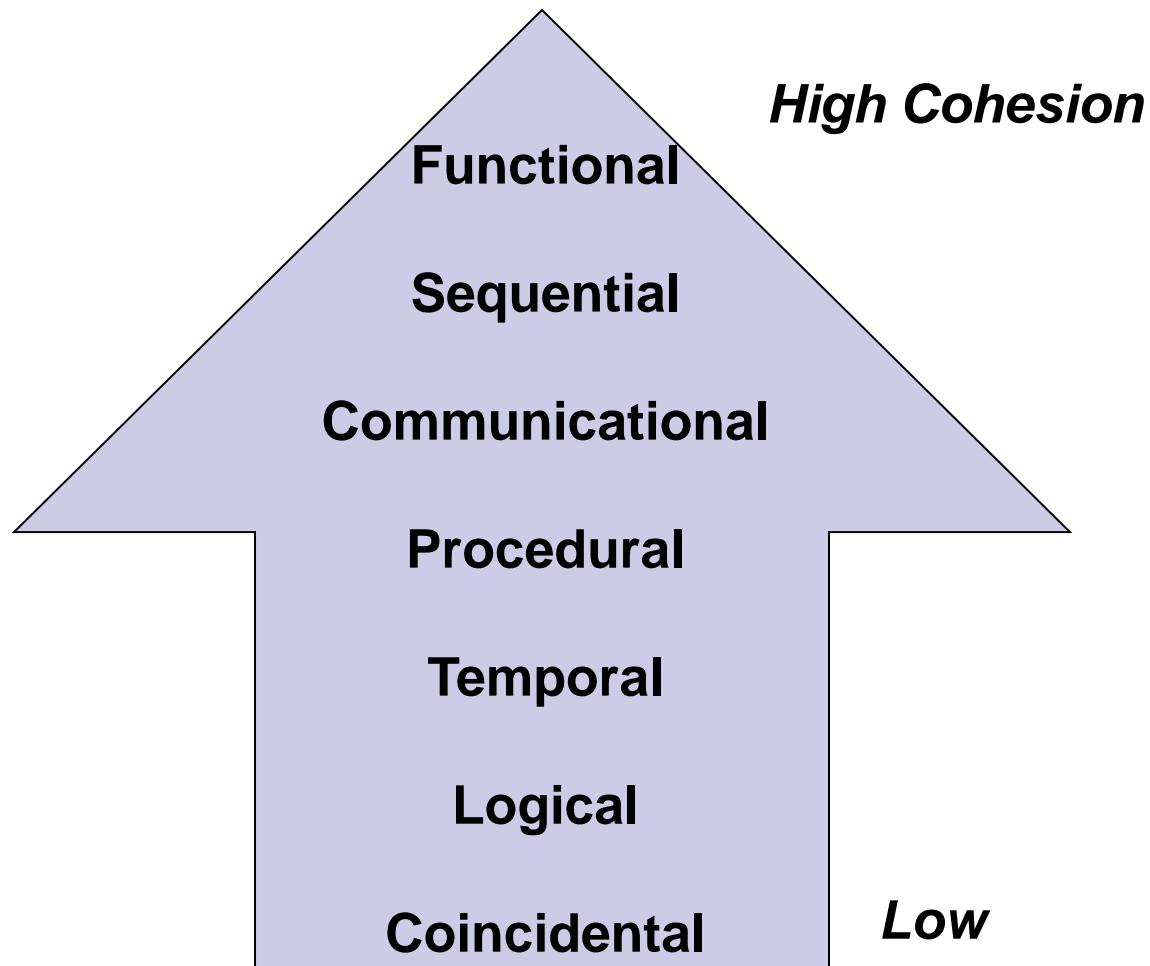
- Component independence
  - High cohesion
  - Low coupling
- Exception identification and handling
- Fault prevention and fault tolerance
- Design for change

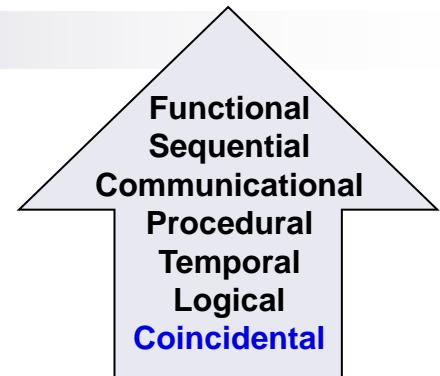


# Cohesion

- Definition
  - The degree to which all elements of a component are directed towards a single task.
  - The degree to which all elements directed towards a task are contained in a single component.
  - The degree to which all responsibilities of a single class are related.
- Internal glue with which component is constructed
- All elements of component are directed toward and essential for performing the same task.

# Type of Cohesion



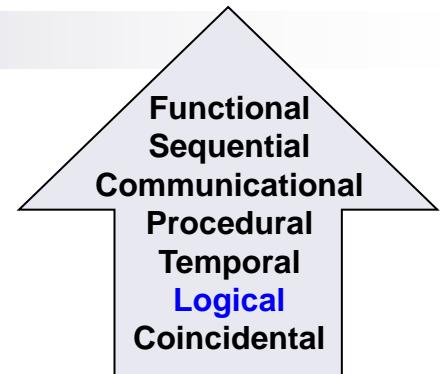


# Coincidental Cohesion

- Def: Parts of the component are unrelated (unrelated functions, processes, or data)
- Parts of the component are only related by their location in source code.
- Elements needed to achieve some functionality are scattered throughout the system.
- Accidental
- Worst form

# Example

1. Print next line
2. Reverse string of characters in second argument
3. Add 7 to 5<sup>th</sup> argument
4. Convert 4<sup>th</sup> argument to float



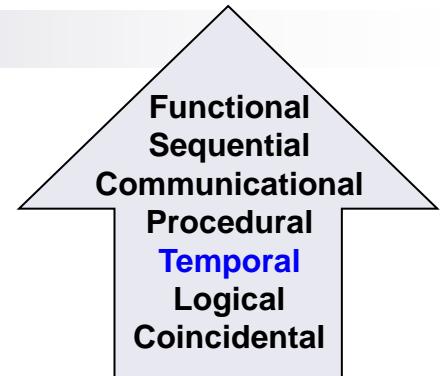
# Logical Cohesion

- Def: Elements of component are related logically and not functionally.
- Several logically related elements are in the same component and one of the elements is selected by the client component.

# Example

- A component reads inputs from tape, disk, and network.
- All the code for these functions are in the same component.
- Operations are related, but the functions are significantly different.

Improvement?



# Temporal Cohesion

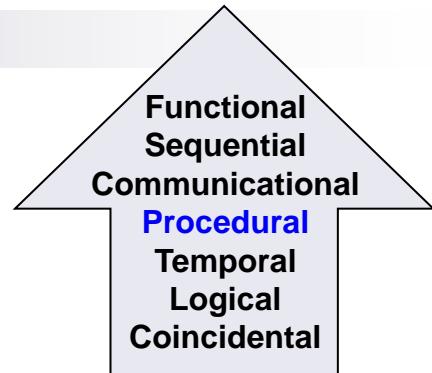
- Def: Elements are related by timing involved
- Elements are grouped by when they are processed.
- Example: An exception handler that
  - Closes all open files
  - Creates an error log
  - Notifies user
  - Lots of different activities occur, all at same time

# Example

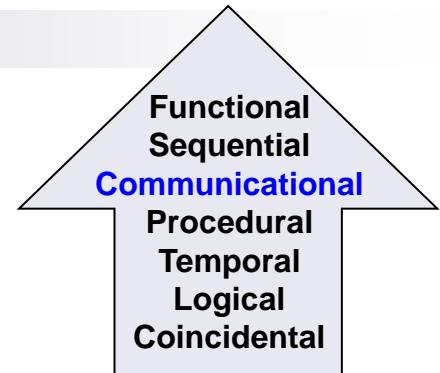
- A system initialization routine: this routine contains all of the code for initializing all of the parts of the system. Lots of different activities occur, all at init time.

Improvement?

# Procedural Cohesion

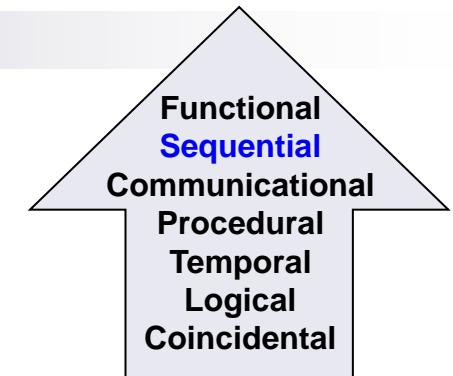


- Def: Elements of a component are related only to ensure a particular order of execution.
- Actions are still weakly connected and unlikely to be reusable.
- Example:
  - ...
  - Write output record
  - Read new input record
  - Pad input with spaces
  - Return new record
  - ...



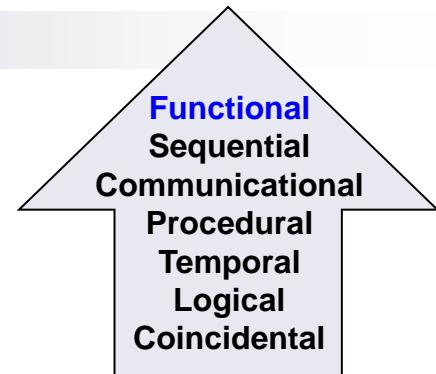
# Communicational Cohesion

- Def: Functions performed on the same data or to produce the same data.
- Examples:
  - Update record in data base and send it to the printer
    - Update a record on a database
    - Print the record
  - Fetch unrelated data at the same time.
    - To minimize disk access



# Sequential Cohesion

- Def: The output of one part is the input to another.
- *Data flows* between parts (different from procedural cohesion)
- Occurs naturally in functional programming languages
- Good situation



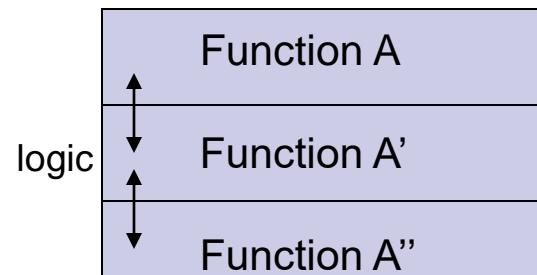
# Functional Cohesion

- Def: Every essential element to a single computation is contained in the component.
- Every element in the component is essential to the computation.
- Ideal situation
- What is a functionally cohesive component?
  - One that not only performs the task for which it was designed but
  - it performs only that function and nothing else.

# Examples of Cohesion

Function A	
Function B	Function C
Function D	Function E

*Coincidental*  
Parts unrelated



*Logical*  
Similar functions

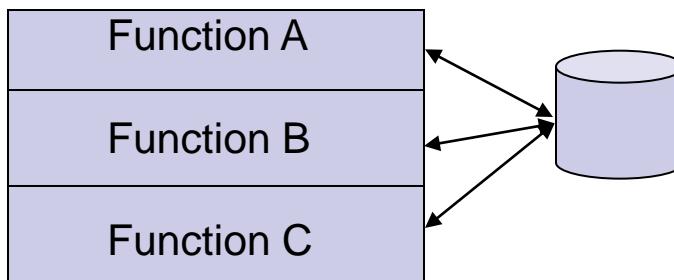
Time $t_0$
Time $t_0 + X$
Time $t_0 + 2X$

*Temporal*  
Related by time

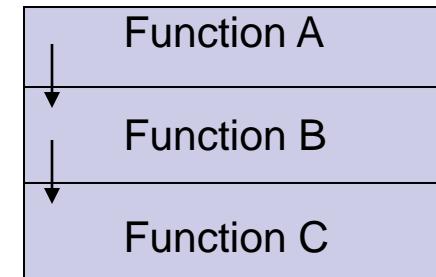
Function A
Function B
Function C

*Procedural*  
Related by order of functions

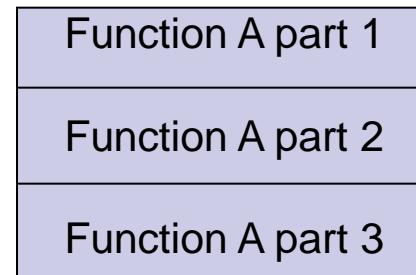
# Examples of Cohesion (Cont.)



*Communicational*  
Access same data



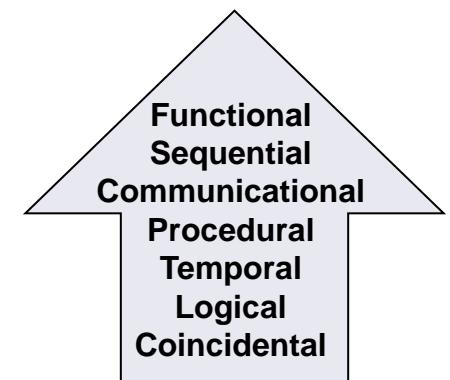
*Sequential*  
Output of one is input to another



*Functional*  
Sequential with complete, related functions

# Exercise: Cohesion for Each Module?

- Compute average daily temperatures at various sites
- Initialize sums and open files
- Create new temperature record
- Store temperature record
- Close files and print average temperatures
- Read in site, time, and temperature
- Store record for specific site
- Edit site, time, or temperature field

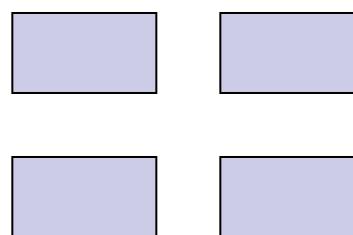


# Outline

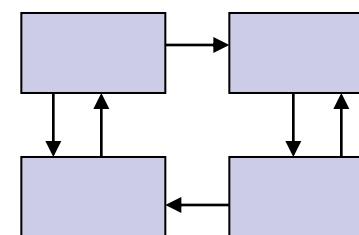
- ✓ Cohesion
- Coupling

# Coupling

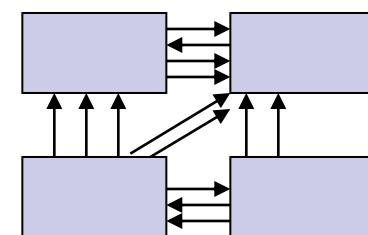
- The degree of dependence such as the amount of interactions among components



No dependencies



Loosely coupled  
some dependencies



Highly coupled  
many dependencies

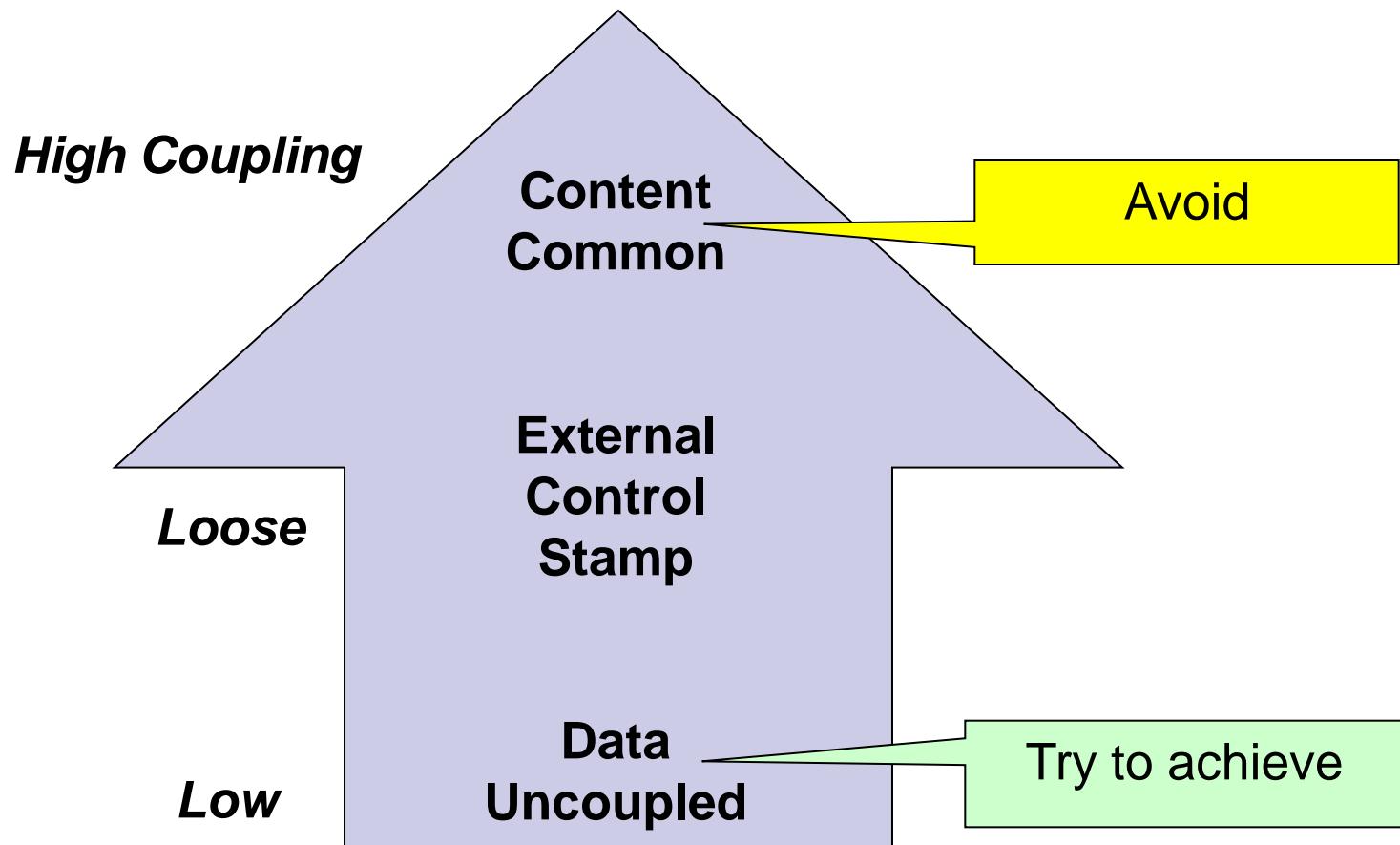
# Coupling

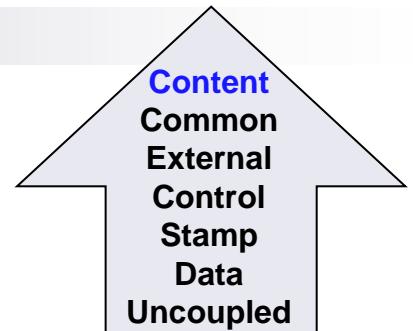
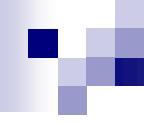
- The degree of dependence such as the amount of interactions among components
- How can you tell if two components are coupled?
- (In pairs, 2 minutes)

# Indications of Coupling

- ?

# Type of Coupling





Content  
Common  
External  
Control  
Stamp  
Data  
Uncoupled

# Content Coupling

- Def: One component modifies another.
- Example:
  - Component directly modifies another's data
  - Component modifies another's code, e.g., jumps (goto) into the middle of a routine
- Question
  - Language features allowing this?

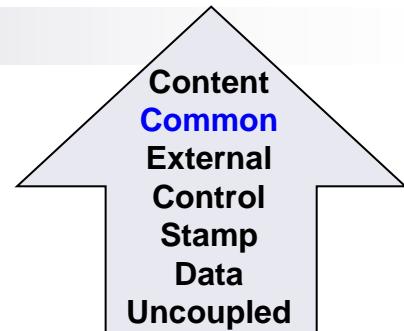
# Example

Part of a program handles lookup for customer.

When customer not found, component adds customer by directly modifying the contents of the data structure containing customer data.

Improvement?

# Common Coupling



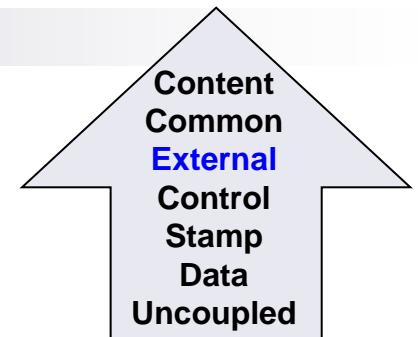
Content  
Common  
External  
Control  
Stamp  
Data  
Uncoupled

- Def: More than one component share data such as global data structures
- Usually a poor design choice because
  - Lack of clear responsibility for the data
  - Reduces readability
  - Difficult to determine all the components that affect a data element (reduces maintainability)
  - Difficult to reuse components
  - Reduces ability to control data accesses

# Example

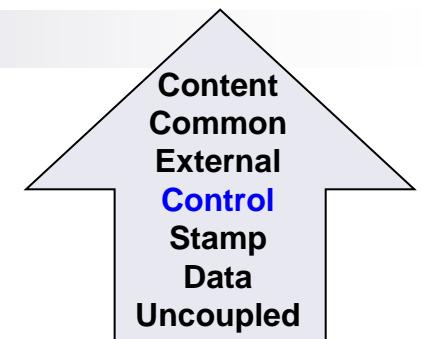
Process control component maintains current data about state of operation. Gets data from multiple sources. Supplies data to multiple sinks. Each source process writes directly to global data store. Each sink process reads directly from global data store.

Improvement?



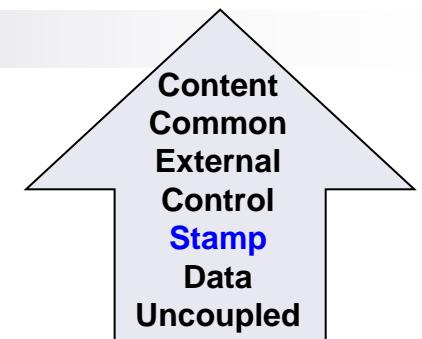
# External Coupling

- Def: Two components share something externally imposed, e.g.,
  - External file
  - Device interface
  - Protocol
  - Data format
- Improvement?



# Control Coupling

- Def: Component passes control parameters to coupled components.
- May be either good or bad, depending on situation.
  - Bad if parameters indicate completely different behavior
  - Good if parameters allow factoring and reuse of functionality
- Good example: sort that takes a comparison function as an argument.
  - The sort function is clearly defined: return a list in sorted order, where sorted is determined by a parameter.



# Stamp Coupling

- Def: Component passes a data structure to another component that does not have access to the entire structure.
- Requires second component to know how to manipulate the data structure (e.g., needs to know about implementation).
- The second has access to more information than it needs.
- May be necessary due to efficiency factors: this is a choice made by insightful designer, not lazy programmer.

# Example

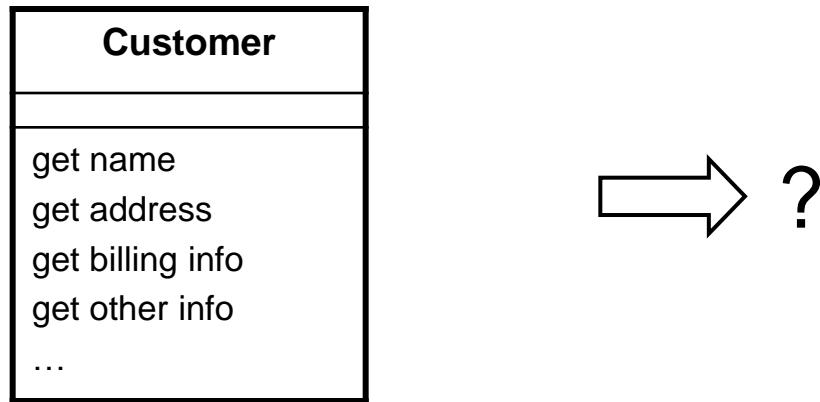
## Customer Billing System

The print routine of the customer billing accepts customer data structure as an argument, parses it, and prints the name, address, and billing information.

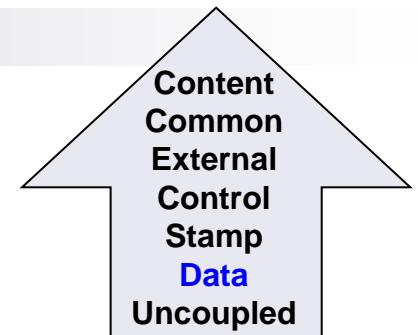
## Improvement?

# Improvement --- OO Solution

- Use an interface to limit access from clients

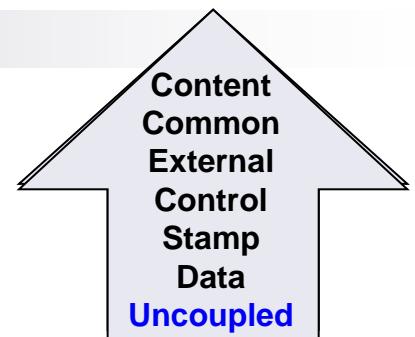


```
void print (Customer c) { ... }
```



# Data Coupling

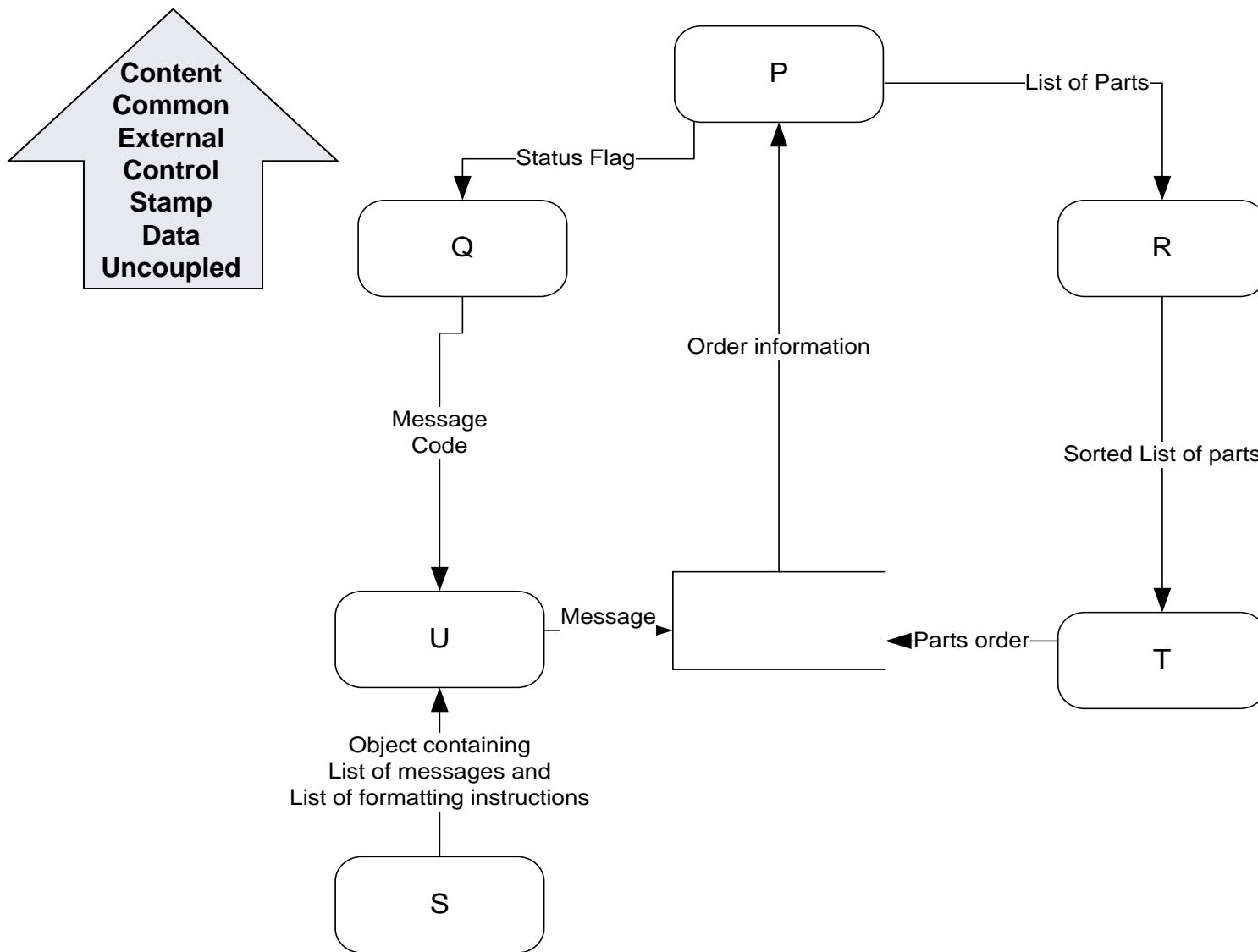
- Def: Component passes data (not data structures) to another component.
- Every argument is simple argument or data structure in which all elements are used
- Good, if it can be achieved.
- Example: Customer billing system
  - The print routine takes the customer name, address, and billing information as arguments.



# Uncoupled

- Completely uncoupled components are not systems.
- Systems are made of interacting components.

# Exercise: Define Coupling between Pairs of Modules



# Coupling between Pairs of Modules

	Q	R	S	T	U
P					
Q					
R					
S					
T					

# Consequences of Coupling

- Why does coupling matter? What are the costs and benefits of coupling?
- (pairs, 3 minutes)

# Consequences of Coupling

- High coupling
  - Components are difficult to understand in isolation
  - Changes in component ripple to others
  - Components are difficult to reuse
    - Need to include all coupled components
    - Difficult to understand
- Low coupling
  - May incur performance cost
  - Generally faster to build systems with low coupling

# In Class

Groups of 2 or 3:

- P1: What is the effect of cohesion on maintenance?
- P2: What is the effect of coupling on maintenance?

# Chapter 5

---

## ■ Agile Development

*Slide Set to accompany*

*Software Engineering: A Practitioner's Approach, 8/e*  
**by Roger S. Pressman and Bruce R. Maxim**

Slides copyright © 1996, 2001, 2005, 2009, 2014 by Roger S. Pressman

***For non-profit educational use only***

May be reproduced ONLY for student use at the university level when used in conjunction with *Software Engineering: A Practitioner's Approach, 8/e*. Any other reproduction or use is prohibited without the express written permission of the author.

All copyright information MUST appear if these slides are posted on a website for student use.

# The Manifesto for Agile Software Development

---

“We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

- ***Individuals and interactions over processes and tools***
- ***Working software over comprehensive documentation***
- ***Customer collaboration over contract negotiation***
- ***Responding to change over following a plan***

That is, while there is value in the items on the right, we value the items on the left more.”

*Kent Beck et al*

# What is “Agility”?

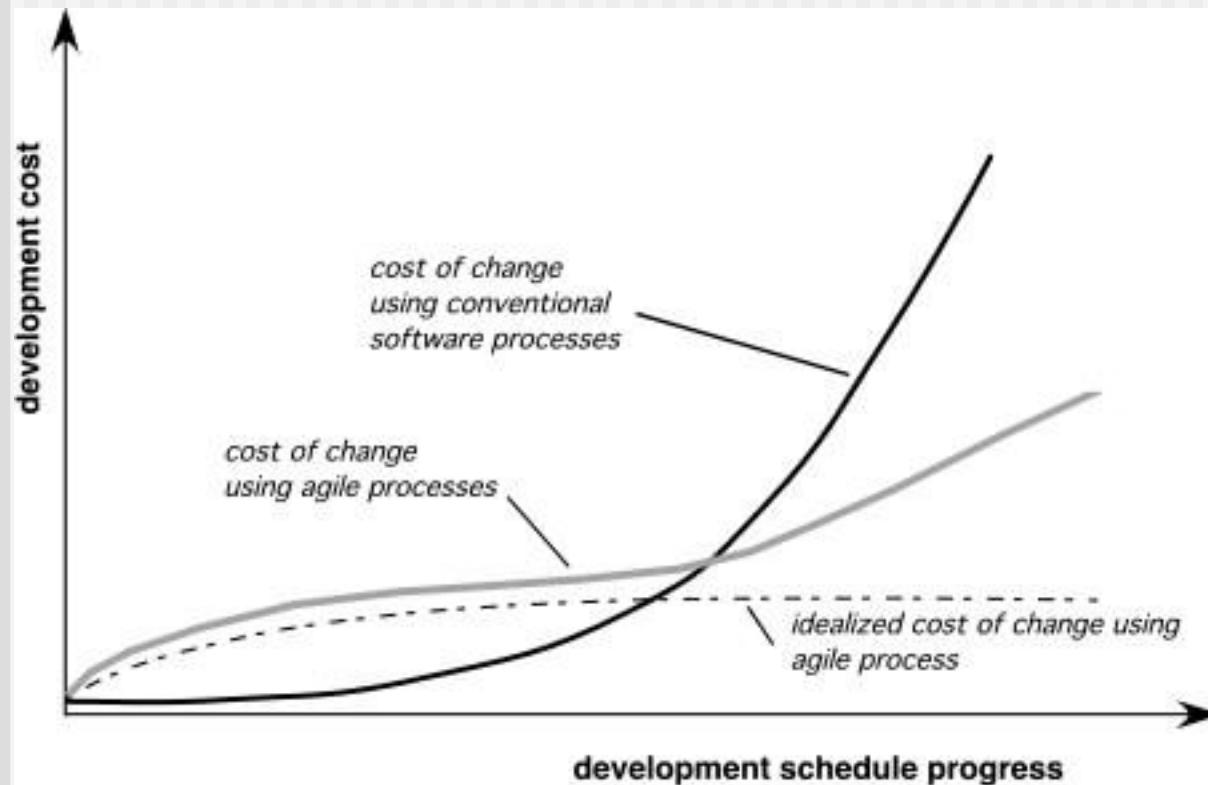
---

- Effective (rapid and adaptive) response to change
- Effective communication among all stakeholders
- Drawing the customer onto the team
- Organizing a team so that it is in control of the work performed

*Yielding ...*

- Rapid, incremental delivery of software

# Agility and the Cost of Change



# An Agile Process

---

- Is driven by customer descriptions of what is required (scenarios)
- Recognizes that plans are short-lived
- Develops software iteratively with a heavy emphasis on construction activities
- Delivers multiple ‘software increments’
- Adapts as changes occur

# Agility Principles - I

---

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
4. Business people and developers must work together daily throughout the project.
5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.

# Agility Principles - II

---

7. Working software is the primary measure of progress.
8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
9. Continuous attention to technical excellence and good design enhances agility.
10. Simplicity – the art of maximizing the amount of work not done – is essential.
11. The best architectures, requirements, and designs emerge from self-organizing teams.
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

# Human Factors

---

- *the process molds to the needs of the people and team, not the other way around*
- key traits must exist among the people on an agile team and the team itself:
  - **Competence.**
  - **Common focus.**
  - **Collaboration.**
  - **Decision-making ability.**
  - **Fuzzy problem-solving ability.**
  - **Mutual trust and respect.**
  - **Self-organization.**

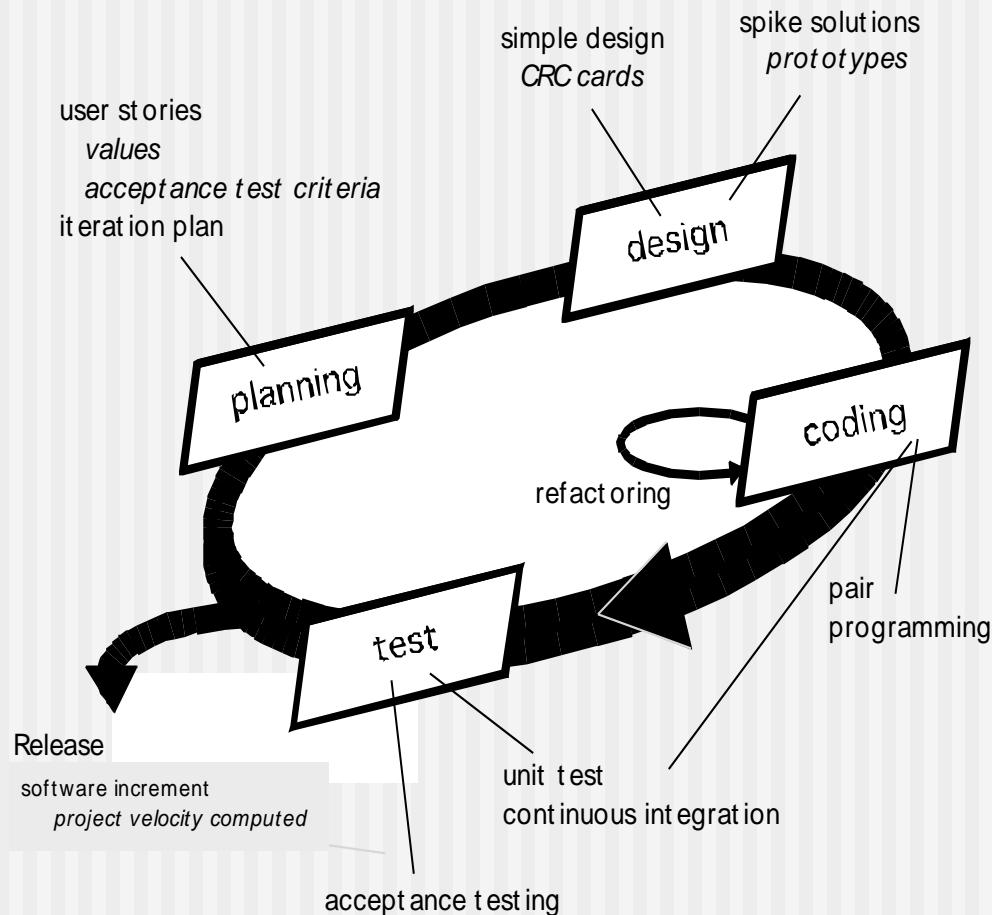
# Extreme Programming (XP)

- The most widely used agile process, originally proposed by Kent Beck
- XP Planning
  - Begins with the creation of “**user stories**”
  - Agile team assesses each story and assigns a **cost**
  - Stories are grouped to form a **deliverable increment**
  - A **commitment** is made on delivery date
  - After the first increment “**project velocity**” is used to help define subsequent delivery dates for other increments

# Extreme Programming (XP)

- XP Design
  - Follows the **KIS principle**
  - Encourage the use of **CRC cards** (see Chapter 8)
  - For difficult design problems, suggests the creation of “**spike solutions**”—a design prototype
  - Encourages “**refactoring**”—an iterative refinement of the internal program design
- XP Coding
  - Recommends the **construction of a unit test** for a store *before* coding commences
  - Encourages “**pair programming**”
- XP Testing
  - All **unit tests are executed daily**
  - “**Acceptance tests**” are defined by the customer and executed to assess customer visible functionality

# Extreme Programming (XP)



# Industrial XP (IXP)

---

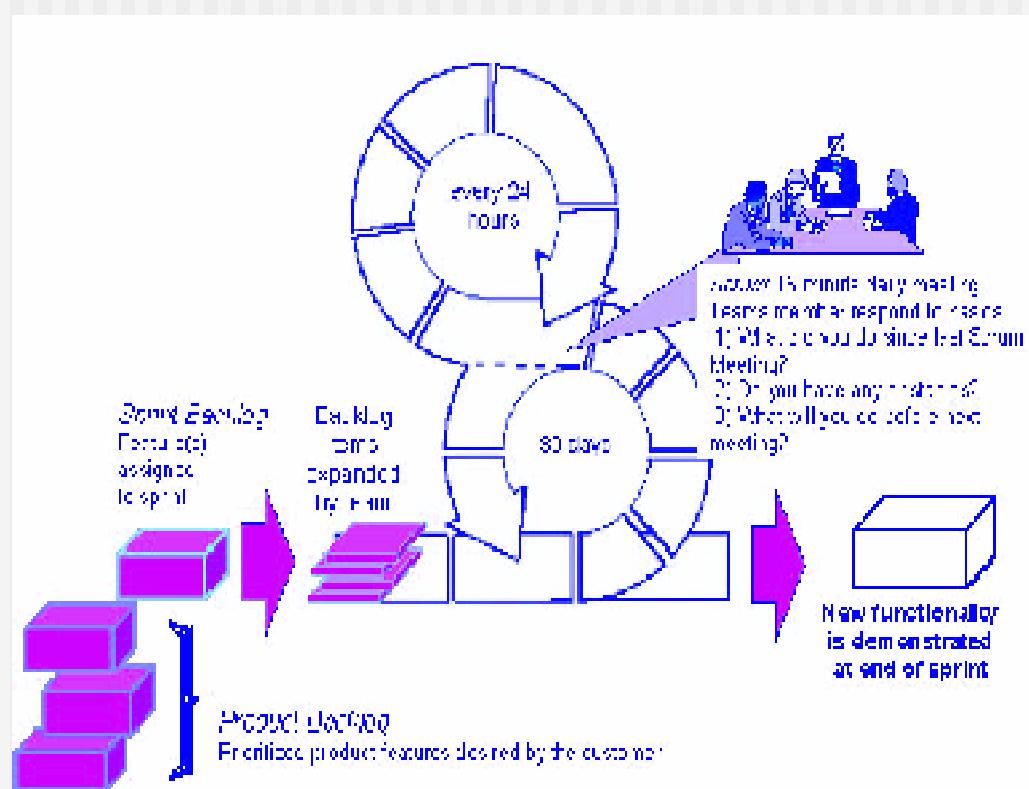
- IXP has greater inclusion of management, expanded customer roles, and upgraded technical practices
- IXP incorporates six new practices:
  - Readiness assessment
  - Project community
  - Project chartering
  - Test driven management
  - Retrospectives
  - Continuous learning

# Scrum

---

- Originally proposed by Schwaber and Beedle
- Scrum—distinguishing features
  - Development work is partitioned into “**packets**”
  - **Testing and documentation are on-going** as the product is constructed
  - Work occurs in “**sprints**” and is derived from a “**backlog**” of existing requirements
  - **Meetings are very short** and sometimes conducted without chairs
  - “**demos**” are delivered to the customer with the time-box allocated

# Scrum



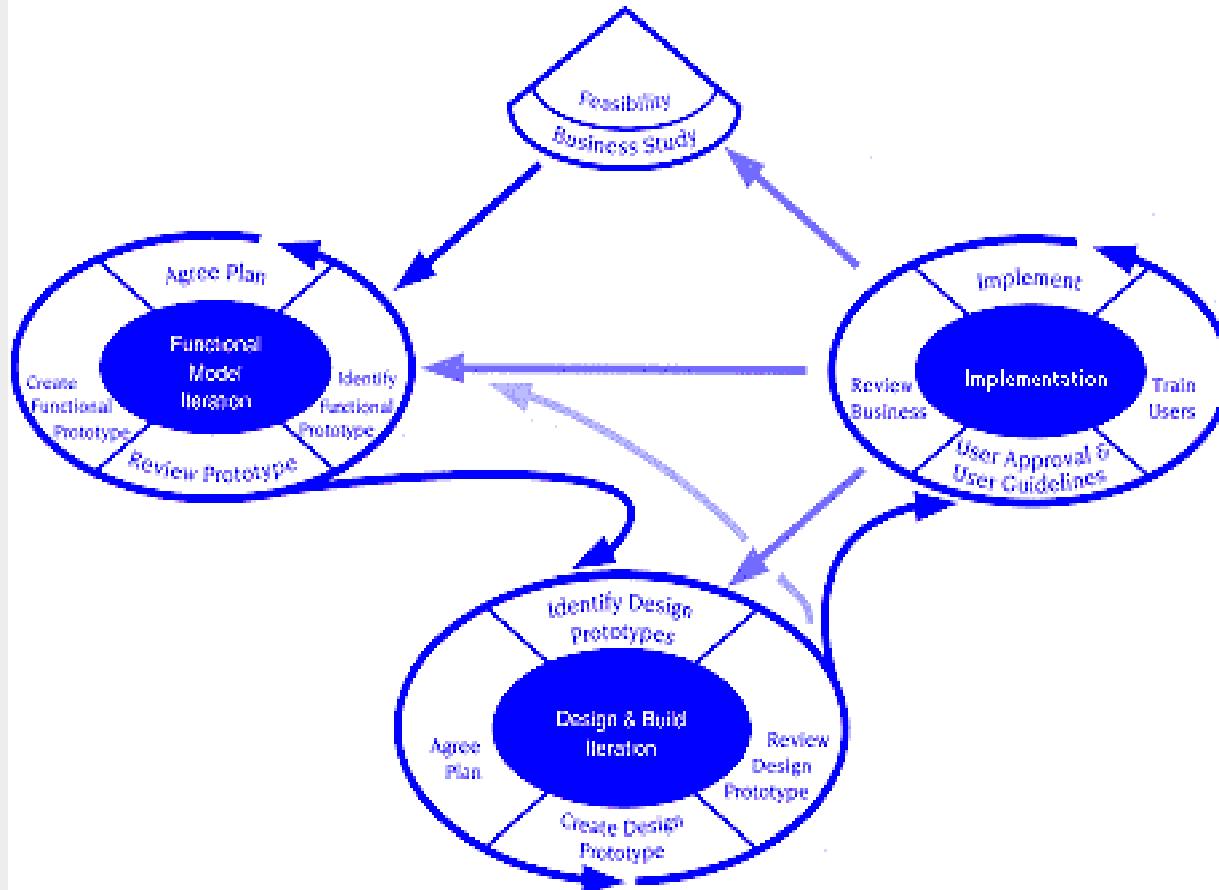
Scrum Process Flow (used with permission)

# Dynamic Systems Development Method

---

- Promoted by the DSDM Consortium ([www.dsdm.org](http://www.dsdm.org))
- DSDM—distinguishing features
  - Similar in most respects to XP
  - Nine guiding principles
    - Active user involvement is imperative.
    - DSDM teams must be empowered to make decisions.
    - The focus is on frequent delivery of products.
    - Fitness for business purpose is the essential criterion for acceptance of deliverables.
    - Iterative and incremental development is necessary to converge on an accurate business solution.
    - All changes during development are reversible.
    - Requirements are baselined at a high level
    - Testing is integrated throughout the life-cycle.

# Dynamic Systems Development Method



**DSDM Life Cycle (with permission of the DSDM consortium)**

These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 8/e* (McGraw-Hill, 2014) Slides copyright 2014 by Roger Pressman.

# Agile Modeling

---

- Originally proposed by Scott Ambler
- Suggests a set of agile modeling principles
  - Model with a purpose
  - Use multiple models
  - Travel light
  - Content is more important than representation
  - Know the models and the tools you use to create them
  - Adapt locally

# Agile Unified Process

---

- Each AUP iteration addresses these activities:
  - Modeling
  - Implementation
  - Testing
  - Deployment
  - Configuration and project management
  - Environment management

# **Chapter 3 – Agile Software Development**

# Topics covered

---

- ✧ Agile methods
- ✧ Agile development techniques
- ✧ Agile project management
- ✧ Scaling agile methods

# Rapid software development

---

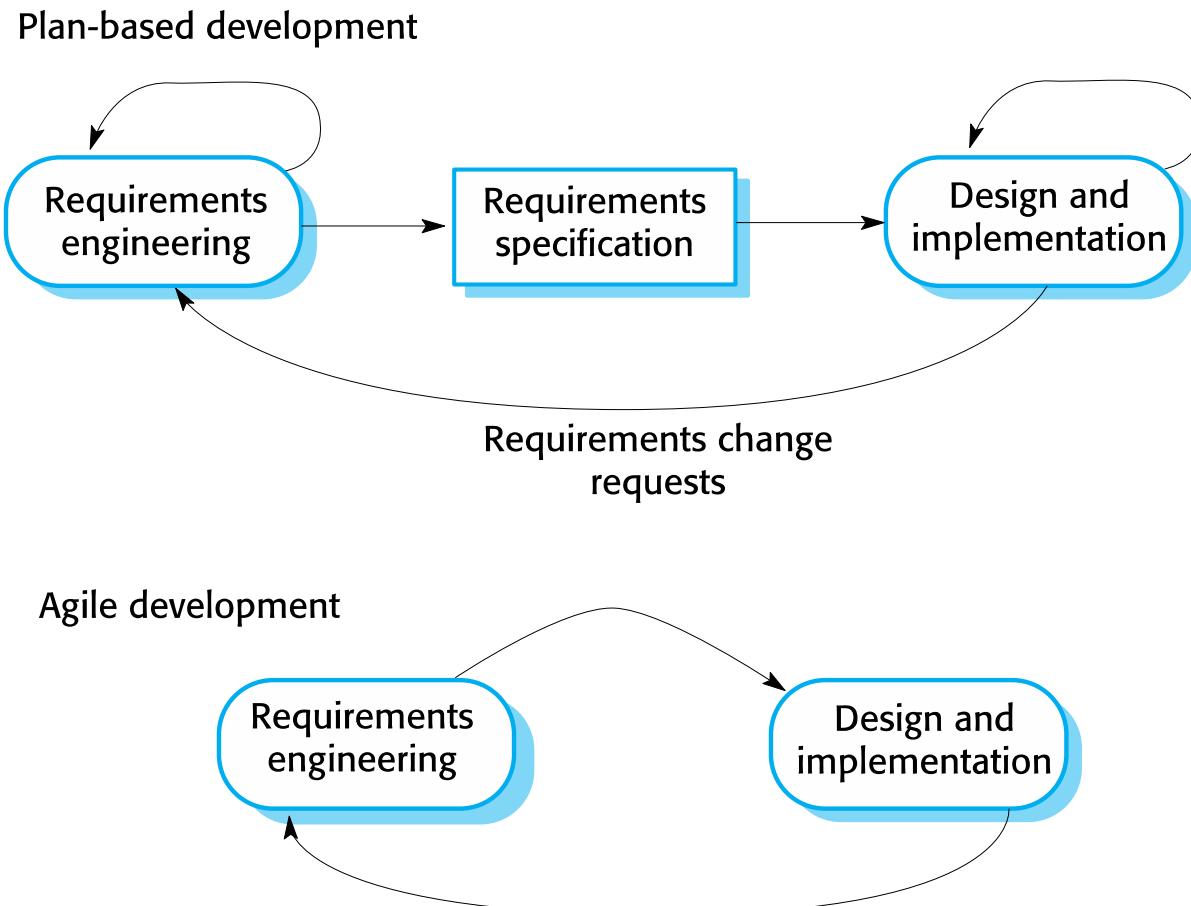
- ✧ Rapid development and delivery is now often the most important requirement for software systems
  - Businesses operate in a fast –changing requirement and it is practically impossible to produce a set of stable software requirements
  - Software has to evolve quickly to reflect changing business needs.
- ✧ Plan-driven development is essential for some types of system but does not meet these business needs.
- ✧ Agile development methods emerged in the late 1990s whose aim was to radically reduce the delivery time for working software systems

# Agile development

---

- ✧ Program specification, design and implementation are inter-leaved
- ✧ The system is developed as a series of versions or increments with stakeholders involved in version specification and evaluation
- ✧ Frequent delivery of new versions for evaluation
- ✧ Extensive tool support (e.g. automated testing tools) used to support development.
- ✧ Minimal documentation – focus on working code

# Plan-driven and agile development



# Plan-driven and agile development

---

## ✧ Plan-driven development

- A plan-driven approach to software engineering is based around separate development stages with the outputs to be produced at each of these stages planned in advance.
- Not necessarily waterfall model – plan-driven, incremental development is possible
- Iteration occurs within activities.

## ✧ Agile development

- Specification, design, implementation and testing are interleaved and the outputs from the development process are decided through a process of negotiation during the software development process.

# Agile methods

# Agile methods

---

- ✧ Dissatisfaction with the overheads involved in software design methods of the 1980s and 1990s led to the creation of agile methods. These methods:
  - Focus on the code rather than the design
  - Are based on an iterative approach to software development
  - Are intended to deliver working software quickly and evolve this quickly to meet changing requirements.
- ✧ The aim of agile methods is to reduce overheads in the software process (e.g. by limiting documentation) and to be able to respond quickly to changing requirements without excessive rework.

# Agile manifesto

---

- ✧ *We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:*
  - *Individuals and interactions over processes and tools  
Working software over comprehensive documentation  
Customer collaboration over contract negotiation  
Responding to change over following a plan*
- ✧ *That is, while there is value in the items on the right, we value the items on the left more.*

# The principles of agile methods

Principle	Description
Customer involvement	Customers should be closely involved throughout the development process. Their role is provide and prioritize new system requirements and to evaluate the iterations of the system.
Incremental delivery	The software is developed in increments with the customer specifying the requirements to be included in each increment.
People not process	The skills of the development team should be recognized and exploited. Team members should be left to develop their own ways of working without prescriptive processes.
Embrace change	Expect the system requirements to change and so design the system to accommodate these changes.
Maintain simplicity	Focus on simplicity in both the software being developed and in the development process. Wherever possible, actively work to eliminate complexity from the system.

# Agile method applicability

---

- ✧ Product development where a software company is developing a small or medium-sized product for sale.
  - Virtually all software products and apps are now developed using an agile approach
- ✧ Custom system development within an organization, where there is a clear commitment from the customer to become involved in the development process and where there are few external rules and regulations that affect the software.

---

# **Agile development techniques**

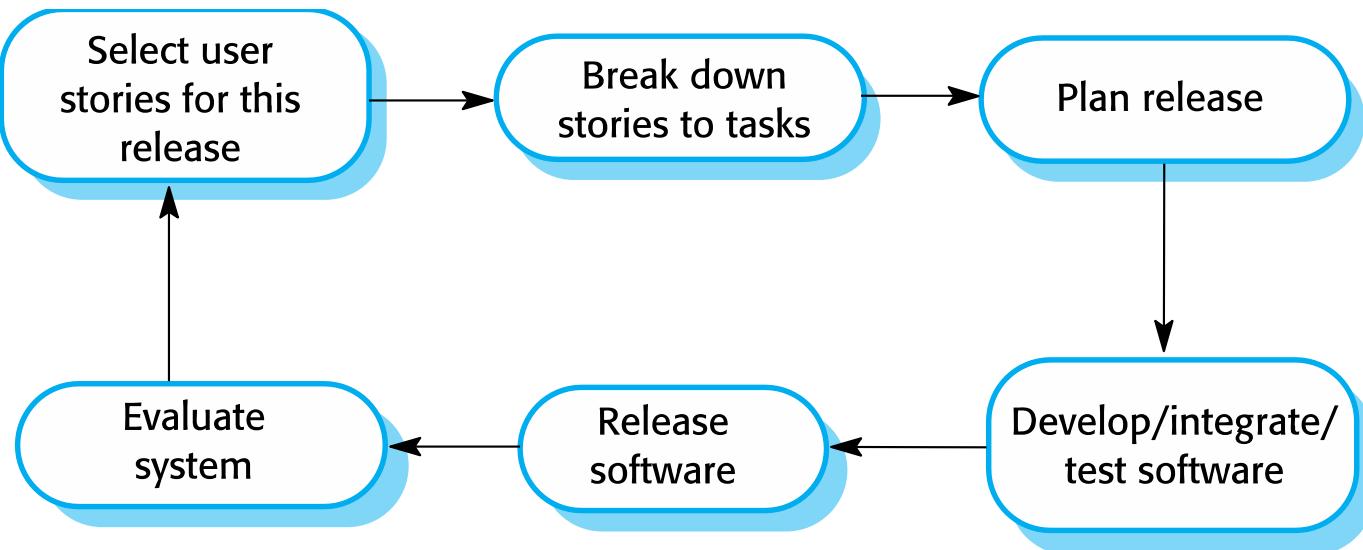
# Extreme programming

---

- ✧ A very influential agile method, developed in the late 1990s, that introduced a range of agile development techniques.
- ✧ Extreme Programming (XP) takes an ‘extreme’ approach to iterative development.
  - New versions may be built several times per day;
  - Increments are delivered to customers every 2 weeks;
  - All tests must be run for every build and the build is only accepted if tests run successfully.

# The extreme programming release cycle

---



# Extreme programming practices (a)

Principle or practice	Description
Incremental planning	Requirements are recorded on story cards and the stories to be included in a release are determined by the time available and their relative priority. The developers break these stories into development 'Tasks'. See Figures 3.5 and 3.6.
Small releases	The minimal useful set of functionality that provides business value is developed first. Releases of the system are frequent and incrementally add functionality to the first release.
Simple design	Enough design is carried out to meet the current requirements and no more.
Test-first development	An automated unit test framework is used to write tests for a new piece of functionality before that functionality itself is implemented.
Refactoring	All developers are expected to refactor the code continuously as soon as possible code improvements are found. This keeps the code simple and maintainable.

# Extreme programming practices (b)

Pair programming	Developers work in pairs, checking each other's work and providing the support to always do a good job.
Collective ownership	The pairs of developers work on all areas of the system, so that no islands of expertise develop and all the developers take responsibility for all of the code. Anyone can change anything.
Continuous integration	As soon as the work on a task is complete, it is integrated into the whole system. After any such integration, all the unit tests in the system must pass.
Sustainable pace	Large amounts of overtime are not considered acceptable as the net effect is often to reduce code quality and medium term productivity
On-site customer	A representative of the end-user of the system (the customer) should be available full time for the use of the XP team. In an extreme programming process, the customer is a member of the development team and is responsible for bringing system requirements to the team for implementation.

# XP and agile principles

---

- ✧ Incremental development is supported through small, frequent system releases.
- ✧ Customer involvement means full-time customer engagement with the team.
- ✧ People not process through pair programming, collective ownership and a process that avoids long working hours.
- ✧ Change supported through regular system releases.
- ✧ Maintaining simplicity through constant refactoring of code.

# Influential XP practices

---

- ✧ Extreme programming has a technical focus and is not easy to integrate with management practice in most organizations.
- ✧ Consequently, while agile development uses practices from XP, the method as originally defined is not widely used.
- ✧ Key practices
  - User stories for specification
  - Refactoring
  - Test-first development
  - Pair programming

# User stories for requirements

---

- ✧ In XP, a customer or user is part of the XP team and is responsible for making decisions on requirements.
- ✧ User requirements are expressed as user stories or scenarios.
- ✧ These are written on cards and the development team break them down into implementation tasks. These tasks are the basis of schedule and cost estimates.
- ✧ The customer chooses the stories for inclusion in the next release based on their priorities and the schedule estimates.

# A ‘prescribing medication’ story

## Prescribing medication

The record of the patient must be open for input. Click on the medication field and select either ‘current medication’, ‘new medication’ or ‘formulary’.

If you select ‘current medication’, you will be asked to check the dose; If you wish to change the dose, enter the new dose then confirm the prescription.

If you choose, ‘new medication’, the system assumes that you know which medication you wish to prescribe. Type the first few letters of the drug name. You will then see a list of possible drugs starting with these letters. Choose the required medication. You will then be asked to check that the medication you have selected is correct. Enter the dose then confirm the prescription.

If you choose ‘formulary’, you will be presented with a search box for the approved formulary. Search for the drug required then select it. You will then be asked to check that the medication you have selected is correct. Enter the dose then confirm the prescription.

In all cases, the system will check that the dose is within the approved range and will ask you to change it if it is outside the range of recommended doses.

After you have confirmed the prescription, it will be displayed for checking. Either click ‘OK’ or ‘Change’. If you click ‘OK’, your prescription will be recorded on the audit database. If you click ‘Change’, you reenter the ‘Prescribing medication’ process.

# Examples of task cards for prescribing medication

---

## **Task 1: Change dose of prescribed drug**

## **Task 2: Formulary selection**

## **Task 3: Dose checking**

Dose checking is a safety precaution to check that the doctor has not prescribed a dangerously small or large dose.

Using the formulary id for the generic drug name, lookup the formulary and retrieve the recommended maximum and minimum dose.

Check the prescribed dose against the minimum and maximum. If outside the range, issue an error message saying that the dose is too high or too low. If within the range, enable the 'Confirm' button.

# Refactoring

---

- ✧ Conventional wisdom in software engineering is to design for change. It is worth spending time and effort anticipating changes as this reduces costs later in the life cycle.
- ✧ XP, however, maintains that this is not worthwhile as changes cannot be reliably anticipated.
- ✧ Rather, it proposes constant code improvement (refactoring) to make changes easier when they have to be implemented.

# Refactoring

---

- ✧ Programming team look for possible software improvements and make these improvements even where there is no immediate need for them.
- ✧ This improves the understandability of the software and so reduces the need for documentation.
- ✧ Changes are easier to make because the code is well-structured and clear.
- ✧ However, some changes require architecture refactoring and this is much more expensive.

# Examples of refactoring

---

- ✧ Re-organization of a class hierarchy to remove duplicate code.
- ✧ Tidying up and renaming attributes and methods to make them easier to understand.
- ✧ The replacement of inline code with calls to methods that have been included in a program library.

# Test-first development

---

- ✧ Testing is central to XP and XP has developed an approach where the program is tested after every change has been made.
- ✧ XP testing features:
  - Test-first development.
  - Incremental test development from scenarios.
  - User involvement in test development and validation.
  - Automated test harnesses are used to run all component tests each time that a new release is built.

# Test-driven development

---

- ✧ Writing tests before code clarifies the requirements to be implemented.
- ✧ Tests are written as programs rather than data so that they can be executed automatically. The test includes a check that it has executed correctly.
  - Usually relies on a testing framework such as Junit.
- ✧ All previous and new tests are run automatically when new functionality is added, thus checking that the new functionality has not introduced errors.

# Customer involvement

---

- ✧ The role of the customer in the testing process is to help develop acceptance tests for the stories that are to be implemented in the next release of the system.
- ✧ The customer who is part of the team writes tests as development proceeds. All new code is therefore validated to ensure that it is what the customer needs.
- ✧ However, people adopting the customer role have limited time available and so cannot work full-time with the development team. They may feel that providing the requirements was enough of a contribution and so may be reluctant to get involved in the testing process.

# Test case description for dose checking

---

## **Test 4: Dose checking**

### **Input:**

1. A number in mg representing a single dose of the drug.
2. A number representing the number of single doses per day.

### **Tests:**

1. Test for inputs where the single dose is correct but the frequency is too high.
2. Test for inputs where the single dose is too high and too low.
3. Test for inputs where the single dose \* frequency is too high and too low.
4. Test for inputs where single dose \* frequency is in the permitted range.

### **Output:**

OK or error message indicating that the dose is outside the safe range.

# Test automation

---

- ✧ Test automation means that tests are written as executable components before the task is implemented
  - These testing components should be stand-alone, should simulate the submission of input to be tested and should check that the result meets the output specification. An automated test framework (e.g. Junit) is a system that makes it easy to write executable tests and submit a set of tests for execution.
- ✧ As testing is automated, there is always a set of tests that can be quickly and easily executed
  - Whenever any functionality is added to the system, the tests can be run and problems that the new code has introduced can be caught immediately.

# Problems with test-first development

---

- ✧ Programmers prefer programming to testing and sometimes they take short cuts when writing tests. For example, they may write incomplete tests that do not check for all possible exceptions that may occur.
- ✧ Some tests can be very difficult to write incrementally. For example, in a complex user interface, it is often difficult to write unit tests for the code that implements the ‘display logic’ and workflow between screens.
- ✧ It difficult to judge the completeness of a set of tests. Although you may have a lot of system tests, your test set may not provide complete coverage.

# Pair programming

---

- ✧ Pair programming involves programmers working in pairs, developing code together.
- ✧ This helps develop common ownership of code and spreads knowledge across the team.
- ✧ It serves as an informal review process as each line of code is looked at by more than 1 person.
- ✧ It encourages refactoring as the whole team can benefit from improving the system code.

# Pair programming

---

- ✧ In pair programming, programmers sit together at the same computer to develop the software.
- ✧ Pairs are created dynamically so that all team members work with each other during the development process.
- ✧ The sharing of knowledge that happens during pair programming is very important as it reduces the overall risks to a project when team members leave.
- ✧ Pair programming is not necessarily inefficient and there is some evidence that suggests that a pair working together is more efficient than 2 programmers working separately.

---

# Agile project management

# Agile project management

---

- ✧ The principal responsibility of software project managers is to manage the project so that the software is delivered on time and within the planned budget for the project.
- ✧ The standard approach to project management is plan-driven. Managers draw up a plan for the project showing what should be delivered, when it should be delivered and who will work on the development of the project deliverables.
- ✧ Agile project management requires a different approach, which is adapted to incremental development and the practices used in agile methods.

# Scrum

---

- ✧ Scrum is an agile method that focuses on managing iterative development rather than specific agile practices.
- ✧ There are three phases in Scrum.
  - The initial phase is an outline planning phase where you establish the general objectives for the project and design the software architecture.
  - This is followed by a series of sprint cycles, where each cycle develops an increment of the system.
  - The project closure phase wraps up the project, completes required documentation such as system help frames and user manuals and assesses the lessons learned from the project.



# Scrum terminology (a)

---

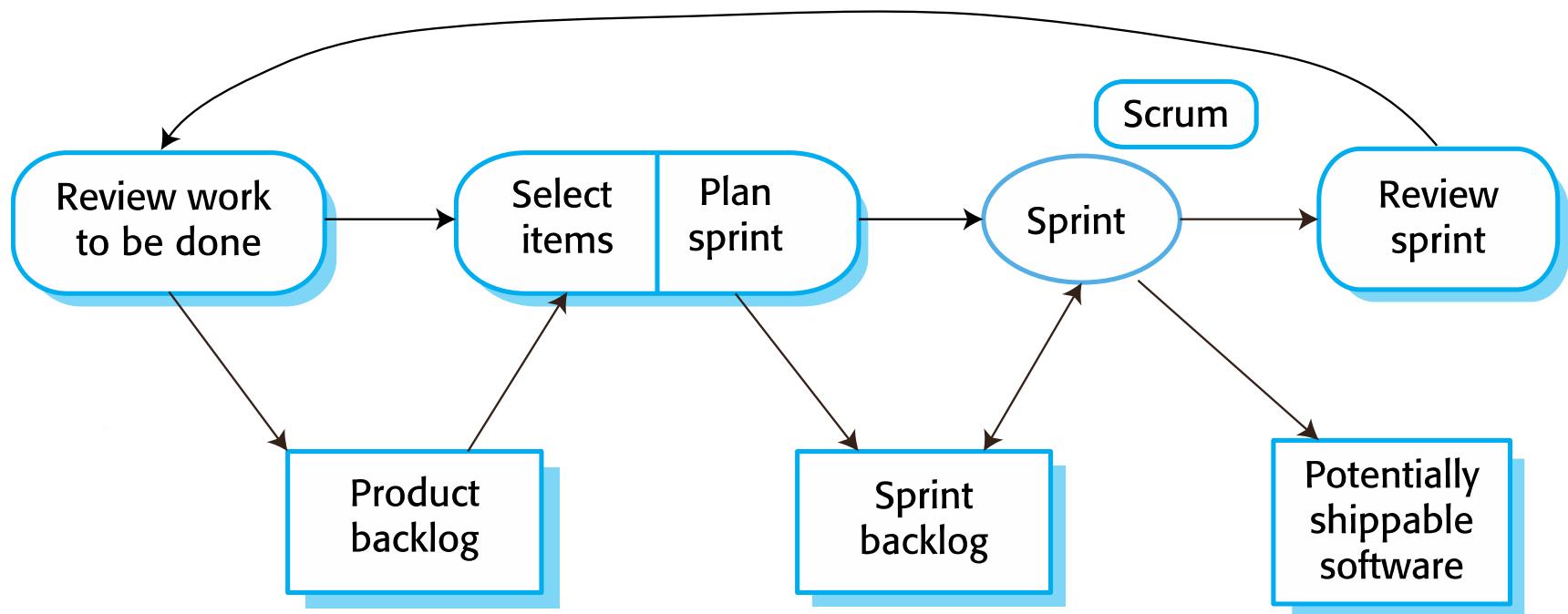
Scrum term	Definition
Development team	A self-organizing group of software developers, which should be no more than 7 people. They are responsible for developing the software and other essential project documents.
Potentially shippable product increment	The software increment that is delivered from a sprint. The idea is that this should be 'potentially shippable' which means that it is in a finished state and no further work, such as testing, is needed to incorporate it into the final product. In practice, this is not always achievable.
Product backlog	This is a list of 'to do' items which the Scrum team must tackle. They may be feature definitions for the software, software requirements, user stories or descriptions of supplementary tasks that are needed, such as architecture definition or user documentation.
Product owner	An individual (or possibly a small group) whose job is to identify product features or requirements, prioritize these for development and continuously review the product backlog to ensure that the project continues to meet critical business needs. The Product Owner can be a customer but might also be a product manager in a software company or other stakeholder representative.

# Scrum terminology (b)

---

Scrum term	Definition
Scrum	A daily meeting of the Scrum team that reviews progress and prioritizes work to be done that day. Ideally, this should be a short face-to-face meeting that includes the whole team.
ScrumMaster	The ScrumMaster is responsible for ensuring that the Scrum process is followed and guides the team in the effective use of Scrum. He or she is responsible for interfacing with the rest of the company and for ensuring that the Scrum team is not diverted by outside interference. The Scrum developers are adamant that the ScrumMaster should not be thought of as a project manager. Others, however, may not always find it easy to see the difference.
Sprint	A development iteration. Sprints are usually 2-4 weeks long.
Velocity	An estimate of how much product backlog effort that a team can cover in a single sprint. Understanding a team's velocity helps them estimate what can be covered in a sprint and provides a basis for measuring improving performance.

# Scrum sprint cycle



# The Scrum sprint cycle

---

- ✧ Sprints are fixed length, normally 2–4 weeks.
- ✧ The starting point for planning is the product backlog, which is the list of work to be done on the project.
- ✧ The selection phase involves all of the project team who work with the customer to select the features and functionality from the product backlog to be developed during the sprint.

# The Sprint cycle

---

- ✧ Once these are agreed, the team organize themselves to develop the software.
- ✧ During this stage the team is isolated from the customer and the organization, with all communications channelled through the so-called ‘Scrum master’.
- ✧ The role of the Scrum master is to protect the development team from external distractions.
- ✧ At the end of the sprint, the work done is reviewed and presented to stakeholders. The next sprint cycle then begins.

# Teamwork in Scrum

---

- ✧ The ‘Scrum master’ is a facilitator who arranges daily meetings, tracks the backlog of work to be done, records decisions, measures progress against the backlog and communicates with customers and management outside of the team.
- ✧ The whole team attends short daily meetings (Scrums) where all team members share information, describe their progress since the last meeting, problems that have arisen and what is planned for the following day.
  - This means that everyone on the team knows what is going on and, if problems arise, can re-plan short-term work to cope with them.

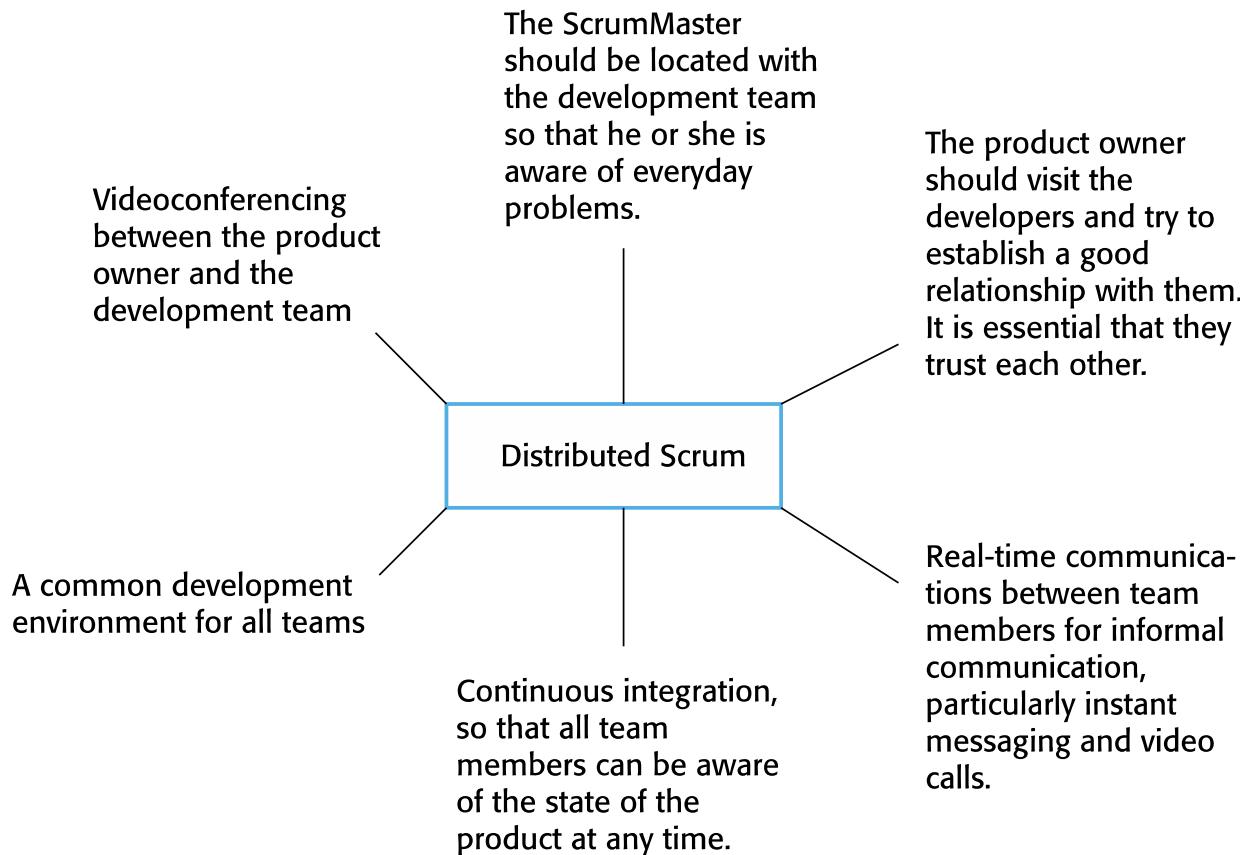
## Scrum benefits

---

- ✧ The product is broken down into a set of manageable and understandable chunks.
- ✧ Unstable requirements do not hold up progress.
- ✧ The whole team have visibility of everything and consequently team communication is improved.
- ✧ Customers see on-time delivery of increments and gain feedback on how the product works.
- ✧ Trust between customers and developers is established and a positive culture is created in which everyone expects the project to succeed.

# Distributed Scrum

---



# Scaling agile methods

# Scaling agile methods

---

- ✧ Agile methods have proved to be successful for small and medium sized projects that can be developed by a small co-located team.
- ✧ It is sometimes argued that the success of these methods comes because of improved communications which is possible when everyone is working together.
- ✧ Scaling up agile methods involves changing these to cope with larger, longer projects where there are multiple development teams, perhaps working in different locations.

# Scaling out and scaling up

---

- ✧ ‘Scaling up’ is concerned with using agile methods for developing large software systems that cannot be developed by a small team.
- ✧ ‘Scaling out’ is concerned with how agile methods can be introduced across a large organization with many years of software development experience.
- ✧ When scaling agile methods it is important to maintain agile fundamentals:
  - Flexible planning, frequent system releases, continuous integration, test-driven development and good team communications.

# Practical problems with agile methods

---

- ✧ The informality of agile development is incompatible with the legal approach to contract definition that is commonly used in large companies.
- ✧ Agile methods are most appropriate for new software development rather than software maintenance. Yet the majority of software costs in large companies come from maintaining their existing software systems.
- ✧ Agile methods are designed for small co-located teams yet much software development now involves worldwide distributed teams.

# Contractual issues

---

- ✧ Most software contracts for custom systems are based around a specification, which sets out what has to be implemented by the system developer for the system customer.
- ✧ However, this precludes interleaving specification and development as is the norm in agile development.
- ✧ A contract that pays for developer time rather than functionality is required.
  - However, this is seen as a high risk my many legal departments because what has to be delivered cannot be guaranteed.

# Agile methods and software maintenance

---

- ✧ Most organizations spend more on maintaining existing software than they do on new software development. So, if agile methods are to be successful, they have to support maintenance as well as original development.
- ✧ Two key issues:
  - Are systems that are developed using an agile approach maintainable, given the emphasis in the development process of minimizing formal documentation?
  - Can agile methods be used effectively for evolving a system in response to customer change requests?
- ✧ Problems may arise if original development team cannot be maintained.

# Agile maintenance

---

- ✧ Key problems are:
  - Lack of product documentation
  - Keeping customers involved in the development process
  - Maintaining the continuity of the development team
- ✧ Agile development relies on the development team knowing and understanding what has to be done.
- ✧ For long-lifetime systems, this is a real problem as the original developers will not always work on the system.

# Agile and plan-driven methods

---

- ✧ Most projects include elements of plan-driven and agile processes. Deciding on the balance depends on:
  - Is it important to have a very detailed specification and design before moving to implementation? If so, you probably need to use a plan-driven approach.
  - Is an incremental delivery strategy, where you deliver the software to customers and get rapid feedback from them, realistic? If so, consider using agile methods.
  - How large is the system that is being developed? Agile methods are most effective when the system can be developed with a small co-located team who can communicate informally. This may not be possible for large systems that require larger development teams so a plan-driven approach may have to be used.

# Agile principles and organizational practice

Principle	Practice
Customer involvement	<p>This depends on having a customer who is willing and able to spend time with the development team and who can represent all system stakeholders. Often, customer representatives have other demands on their time and cannot play a full part in the software development.</p> <p>Where there are external stakeholders, such as regulators, it is difficult to represent their views to the agile team.</p>
Embrace change	<p>Prioritizing changes can be extremely difficult, especially in systems for which there are many stakeholders. Typically, each stakeholder gives different priorities to different changes.</p>
Incremental delivery	<p>Rapid iterations and short-term planning for development does not always fit in with the longer-term planning cycles of business planning and marketing. Marketing managers may need to know what product features several months in advance to prepare an effective marketing campaign.</p>

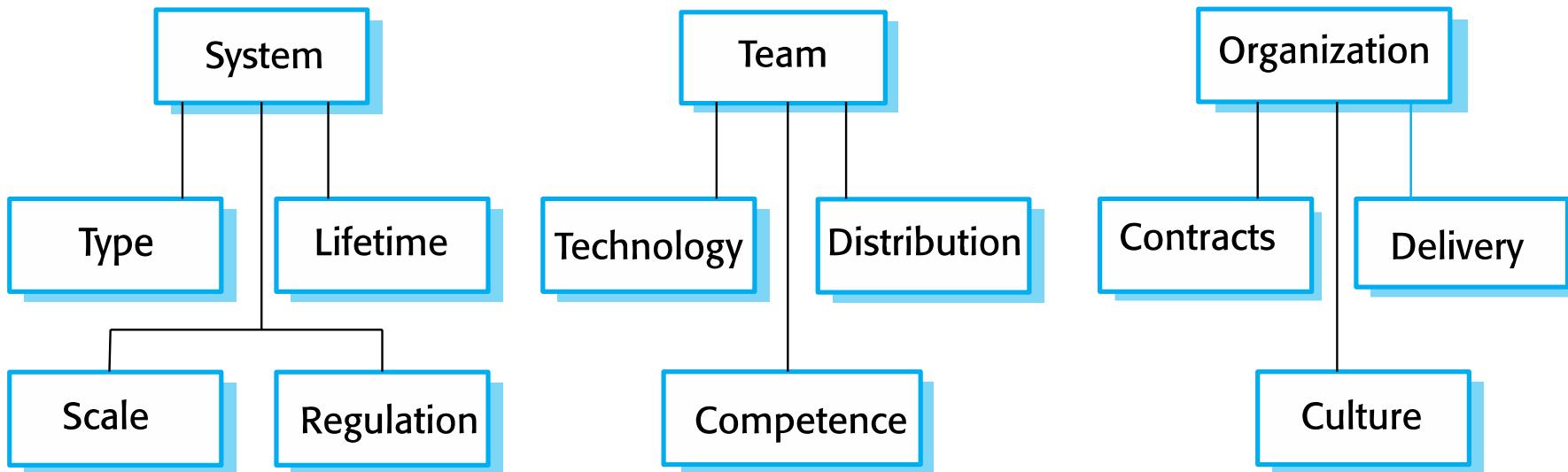
# Agile principles and organizational practice

---

Principle	Practice
Maintain simplicity	Under pressure from delivery schedules, team members may not have time to carry out desirable system simplifications.
People not process	Individual team members may not have suitable personalities for the intense involvement that is typical of agile methods, and therefore may not interact well with other team members.

# Agile and plan-based factors

---



# System issues

---

✧ How large is the system being developed?

- Agile methods are most effective a relatively small co-located team who can communicate informally.

✧ What type of system is being developed?

- Systems that require a lot of analysis before implementation need a fairly detailed design to carry out this analysis.

✧ What is the expected system lifetime?

- Long-lifetime systems require documentation to communicate the intentions of the system developers to the support team.

✧ Is the system subject to external regulation?

- If a system is regulated you will probably be required to produce detailed documentation as part of the system safety case.

# People and teams

---

- ✧ How good are the designers and programmers in the development team?
  - It is sometimes argued that agile methods require higher skill levels than plan-based approaches in which programmers simply translate a detailed design into code.
- ✧ How is the development team organized?
  - Design documents may be required if the team is distributed.
- ✧ What support technologies are available?
  - IDE support for visualisation and program analysis is essential if design documentation is not available.

# Organizational issues

---

- ✧ Traditional engineering organizations have a culture of plan-based development, as this is the norm in engineering.
- ✧ Is it standard organizational practice to develop a detailed system specification?
- ✧ Will customer representatives be available to provide feedback of system increments?
- ✧ Can informal agile development fit into the organizational culture of detailed documentation?

# Agile methods for large systems

---

- ✧ Large systems are usually collections of separate, communicating systems, where separate teams develop each system. Frequently, these teams are working in different places, sometimes in different time zones.
- ✧ Large systems are ‘brownfield systems’, that is they include and interact with a number of existing systems. Many of the system requirements are concerned with this interaction and so don’t really lend themselves to flexibility and incremental development.
- ✧ Where several systems are integrated to create a system, a significant fraction of the development is concerned with system configuration rather than original code development.

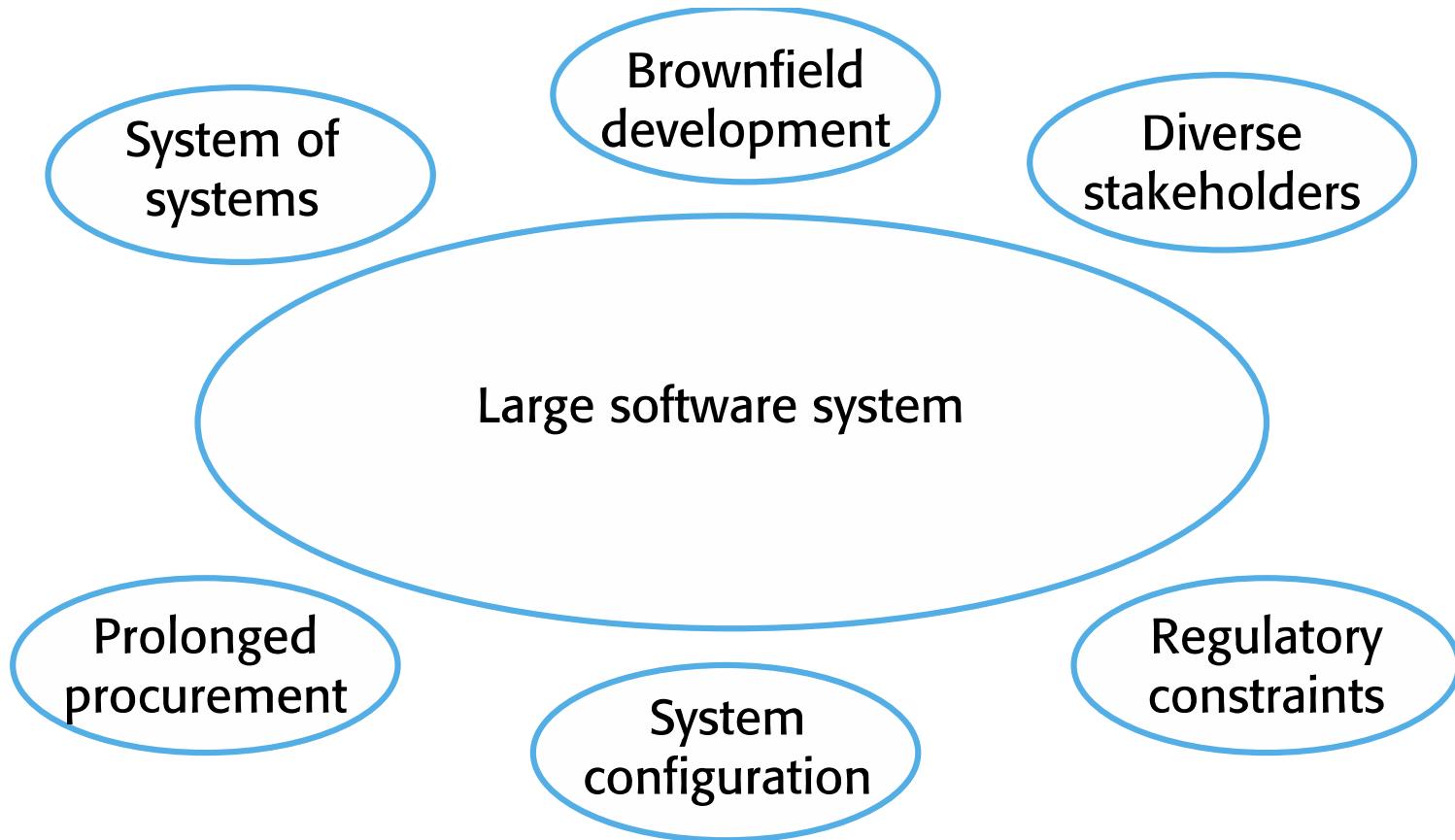
# Large system development

---

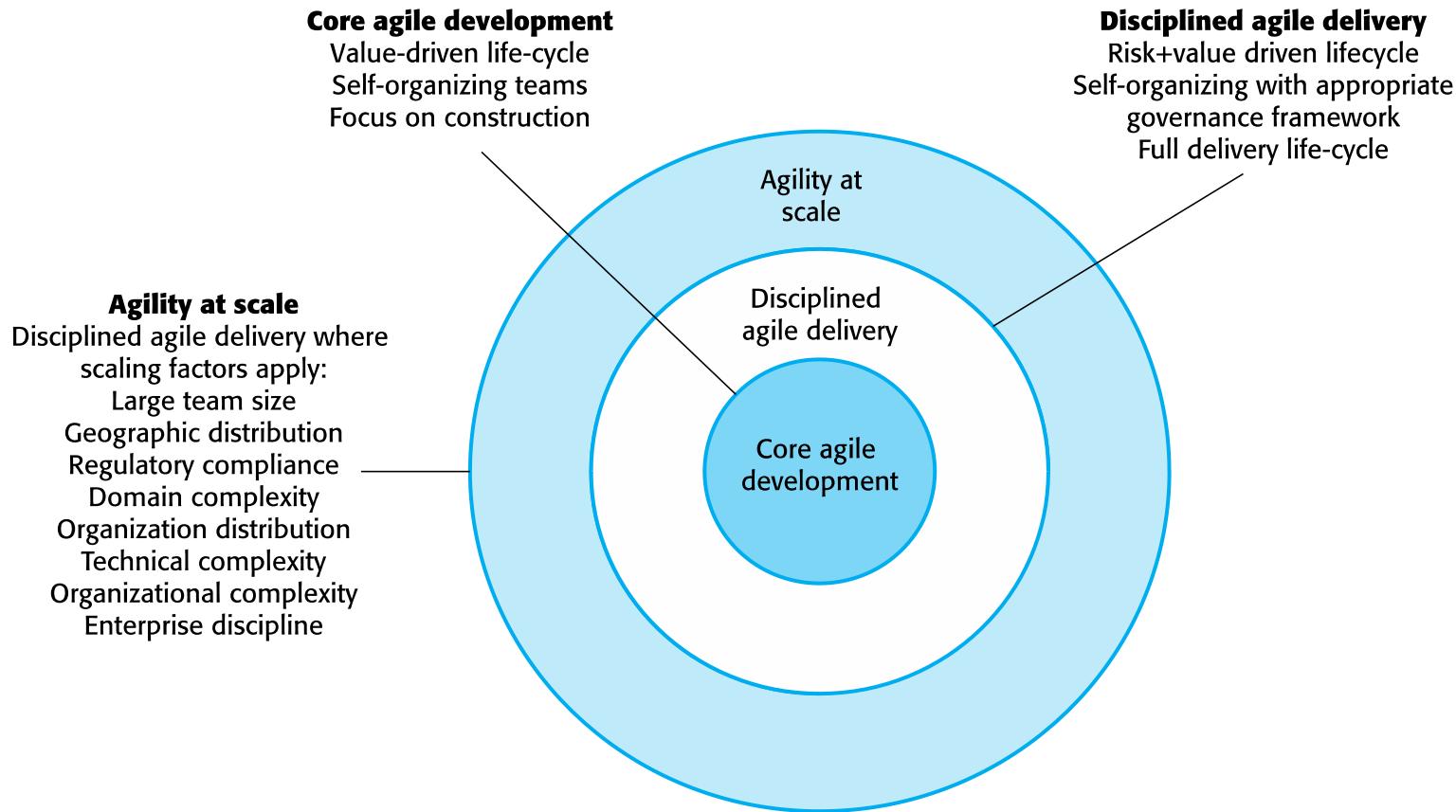
- ✧ Large systems and their development processes are often constrained by external rules and regulations limiting the way that they can be developed.
- ✧ Large systems have a long procurement and development time. It is difficult to maintain coherent teams who know about the system over that period as, inevitably, people move on to other jobs and projects.
- ✧ Large systems usually have a diverse set of stakeholders. It is practically impossible to involve all of these different stakeholders in the development process.

# Factors in large systems

---



# IBM's agility at scale model



# Scaling up to large systems

---

- ✧ A completely incremental approach to requirements engineering is impossible.
- ✧ There cannot be a single product owner or customer representative.
- ✧ For large systems development, it is not possible to focus only on the code of the system.
- ✧ Cross-team communication mechanisms have to be designed and used.
- ✧ Continuous integration is practically impossible. However, it is essential to maintain frequent system builds and regular releases of the system.

# Multi-team Scrum

---

- ✧ *Role replication*
  - Each team has a Product Owner for their work component and ScrumMaster.
- ✧ *Product architects*
  - Each team chooses a product architect and these architects collaborate to design and evolve the overall system architecture.
- ✧ *Release alignment*
  - The dates of product releases from each team are aligned so that a demonstrable and complete system is produced.
- ✧ *Scrum of Scrums*
  - There is a daily Scrum of Scrums where representatives from each team meet to discuss progress and plan work to be done.

# Agile methods across organizations

---

- ✧ Project managers who do not have experience of agile methods may be reluctant to accept the risk of a new approach.
- ✧ Large organizations often have quality procedures and standards that all projects are expected to follow and, because of their bureaucratic nature, these are likely to be incompatible with agile methods.
- ✧ Agile methods seem to work best when team members have a relatively high skill level. However, within large organizations, there are likely to be a wide range of skills and abilities.
- ✧ There may be cultural resistance to agile methods, especially in those organizations that have a long history of using conventional systems engineering processes.

# Key points

---

- ✧ Agile methods are incremental development methods that focus on rapid software development, frequent releases of the software, reducing process overheads by minimizing documentation and producing high-quality code.
- ✧ Agile development practices include
  - User stories for system specification
  - Frequent releases of the software,
  - Continuous software improvement
  - Test-first development
  - Customer participation in the development team.

# Key points

---

- ✧ Scrum is an agile method that provides a project management framework.
  - It is centred round a set of sprints, which are fixed time periods when a system increment is developed.
- ✧ Many practical development methods are a mixture of plan-based and agile development.
- ✧ Scaling agile methods for large systems is difficult.
  - Large systems need up-front design and some documentation and organizational practice may conflict with the informality of agile approaches.

# Chapter 14

---

## ■ Component-Level Design

*Slide Set to accompany*

*Software Engineering: A Practitioner's Approach, 8/e*  
**by Roger S. Pressman and Bruce R. Maxim**

Slides copyright © 1996, 2001, 2005, 2009, 2014 by Roger S. Pressman

***For non-profit educational use only***

May be reproduced ONLY for student use at the university level when used in conjunction with *Software Engineering: A Practitioner's Approach, 8/e*. Any other reproduction or use is prohibited without the express written permission of the author.

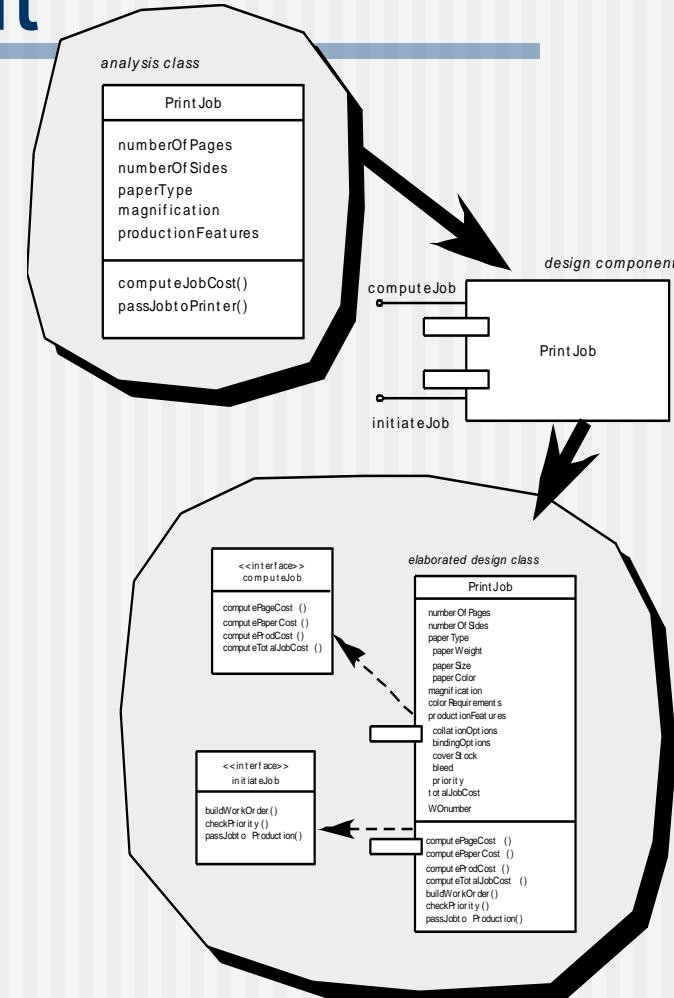
All copyright information MUST appear if these slides are posted on a website for student use.

# What is a Component?

---

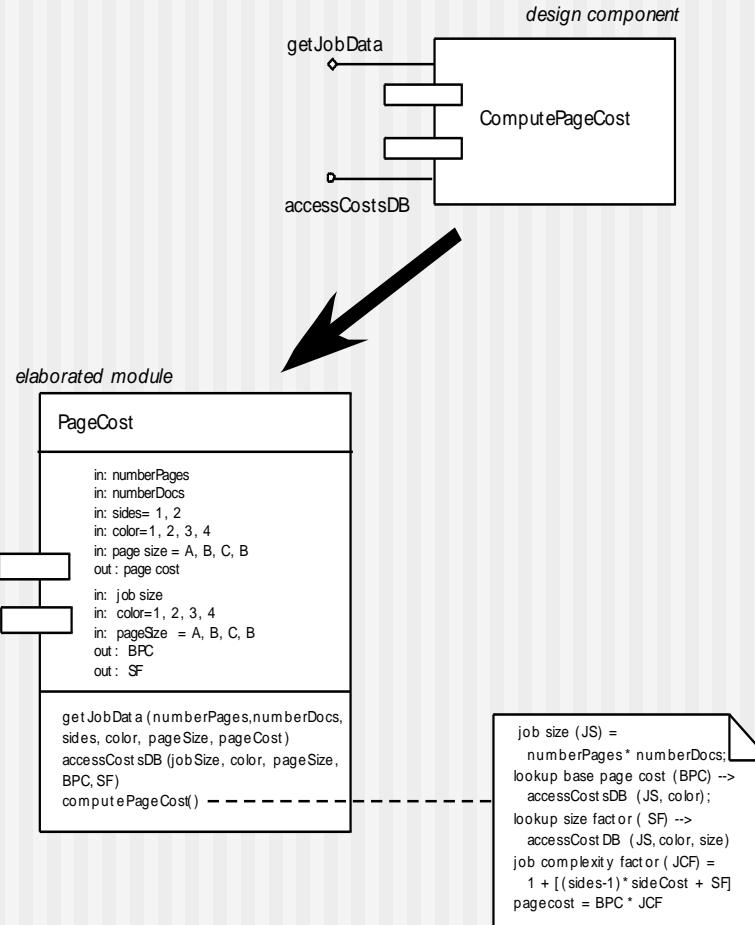
- *OMG Unified Modeling Language Specification* [OMG01] defines a component as
  - “... a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces.”<sup>10</sup>
- **OO view:** a component contains a set of collaborating classes
- **Conventional view:** a component contains processing logic, the internal data structures that are required to implement the processing logic, and an interface that enables the component to be invoked and data to be passed to it.

# OO Component



These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 8/e* (McGraw-Hill, 2014). Slides copyright 2014 by Roger Pressman.

# Conventional Component



# Basic Design Principles

- The Open-Closed Principle (OCP). “*A module [component] should be open for extension but closed for modification.*”
- The Liskov Substitution Principle (LSP). “*Subclasses should be substitutable for their base classes.*”
- Dependency Inversion Principle (DIP). “*Depend on abstractions. Do not depend on concretions.*”
- The Interface Segregation Principle (ISP). “*Many client-specific interfaces are better than one general purpose interface.*”
- The Release Reuse Equivalency Principle (REP). “*The granule of reuse is the granule of release.*”
- The Common Closure Principle (CCP). “*Classes that change together belong together.*”
- The Common Reuse Principle (CRP). “*Classes that aren’t reused together should not be grouped together.*”

Source: Martin, R., “Design Principles and Design Patterns,” downloaded from <http://www.objectmentor.com>, 2000.

# Design Guidelines

---

- Components
  - Naming conventions should be established for components that are specified as part of the architectural model and then refined and elaborated as part of the component-level model
- Interfaces
  - Interfaces provide important information about communication and collaboration
- Dependencies and Inheritance
  - it is a good idea to model dependencies from left to right and inheritance from bottom (derived classes) to top (base classes).

# Cohesion

---

- Conventional view:
  - the “single-mindedness” of a module
- OO view:
  - *cohesion* implies that a component or class encapsulates only attributes and operations that are closely related to one another and to the class or component itself
- Levels of cohesion
  - Functional
  - Layer
  - Communicational
  - Sequential
  - Procedural
  - Temporal
  - utility

# Coupling

---

- Conventional view:
  - The degree to which a component is connected to other components and to the external world
- OO view:
  - a qualitative measure of the degree to which classes are connected to one another
- Level of coupling
  - Content
  - Common
  - Control
  - Stamp
  - Data
  - Routine call
  - Type use
  - Inclusion or import
  - External

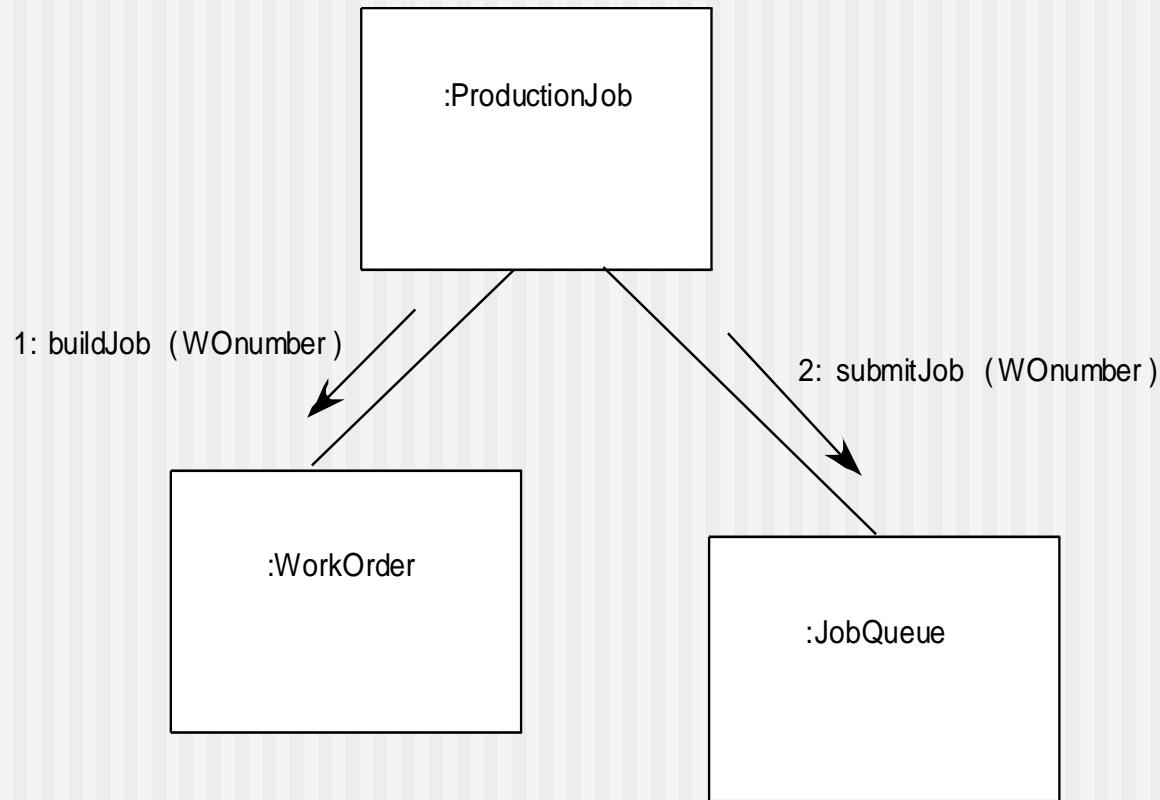
# Component Level Design-I

- Step 1. Identify all design classes that correspond to the problem domain.
- Step 2. Identify all design classes that correspond to the infrastructure domain.
- Step 3. Elaborate all design classes that are not acquired as reusable components.
- Step 3a. Specify message details when classes or component collaborate.
- Step 3b. Identify appropriate interfaces for each component.

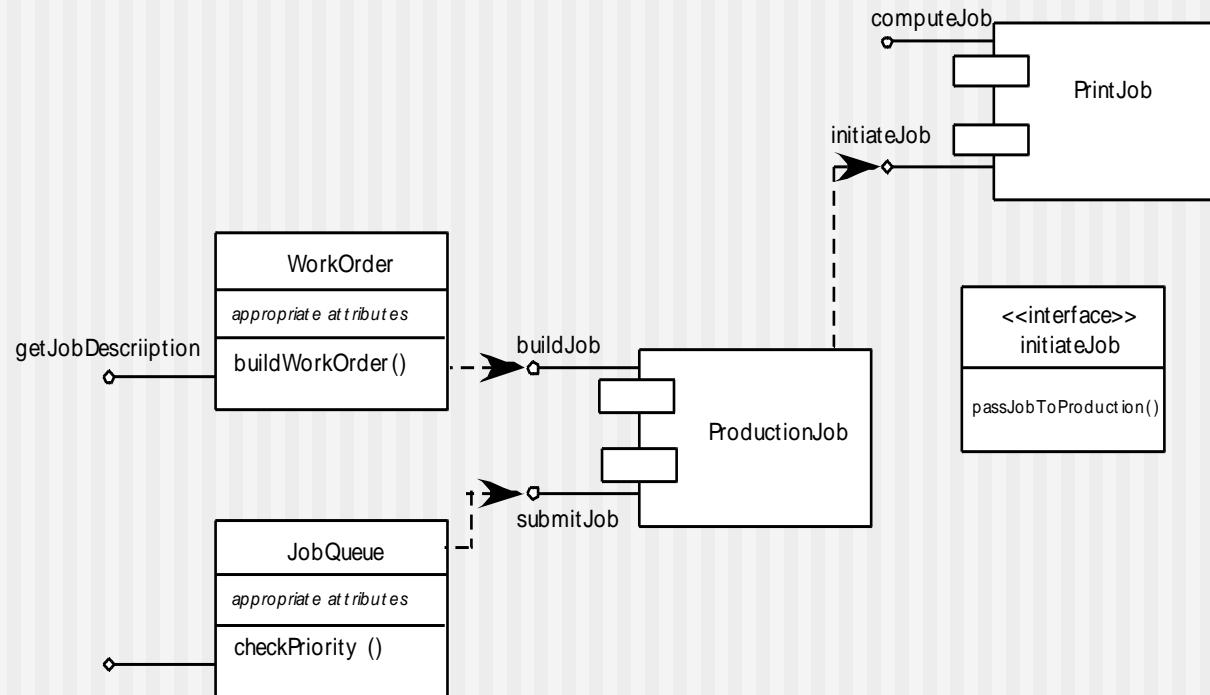
# Component-Level Design-II

- Step 3c. Elaborate attributes and define data types and data structures required to implement them.
- Step 3d. Describe processing flow within each operation in detail.
- Step 4. Describe persistent data sources (databases and files) and identify the classes required to manage them.
- Step 5. Develop and elaborate behavioral representations for a class or component.
- Step 6. Elaborate deployment diagrams to provide additional implementation detail.
- Step 7. Factor every component-level design representation and always consider alternatives.

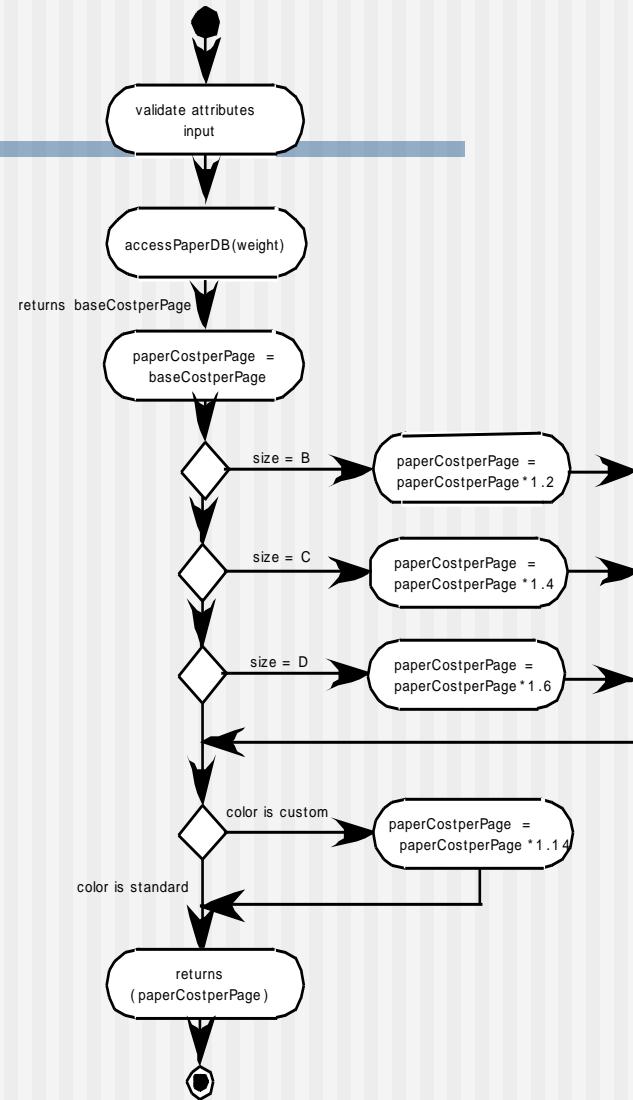
# Collaboration Diagram



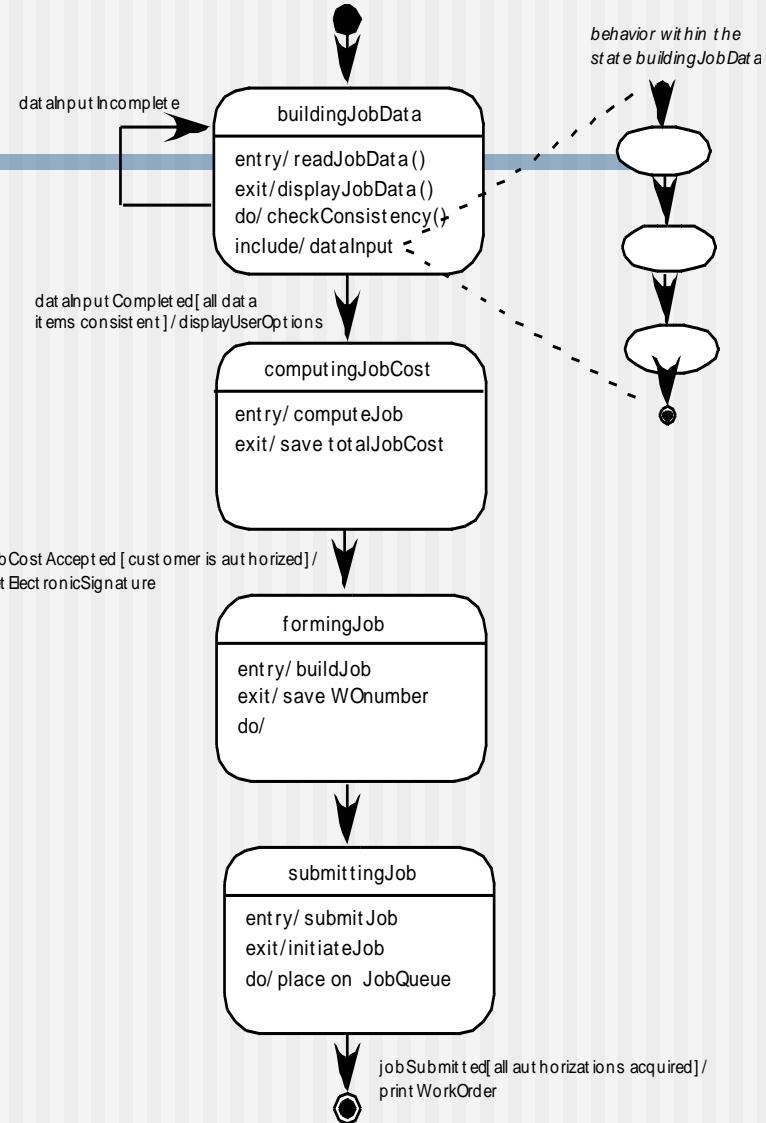
# Refactoring



# Activity Diagram



# Statechart



These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 8/e* (McGraw-Hill, 2014). Slides copyright 2014 by Roger Pressman.

# Component Design for WebApps

---

- WebApp component is
  - (1) a well-defined cohesive function that manipulates content or provides computational or data processing for an end-user, or
  - (2) a cohesive package of content and functionality that provides end-user with some required capability.
- Therefore, component-level design for WebApps often incorporates elements of content design and functional design.

# Content Design for WebApps

---

- focuses on content objects and the manner in which they may be packaged for presentation to a WebApp end-user
- consider a Web-based video surveillance capability within **SafeHomeAssured.com**
  - potential content components can be defined for the video surveillance capability:
    - (1) the content objects that represent the space layout (the floor plan) with additional icons representing the location of sensors and video cameras;
    - (2) the collection of thumbnail video captures (each an separate data object), and
    - (3) the streaming video window for a specific camera.
  - Each of these components can be separately named and manipulated as a package.

# Functional Design for WebApps

---

- Modern Web applications deliver increasingly sophisticated processing functions that:
  - (1) perform localized processing to generate content and navigation capability in a dynamic fashion;
  - (2) provide computation or data processing capability that is appropriate for the WebApp's business domain;
  - (3) provide sophisticated database query and access, or
  - (4) establish data interfaces with external corporate systems.
- To achieve these (and many other) capabilities, you will design and construct WebApp functional components that are identical in form to software components for conventional software.

# Component Design for Mobile Apps

---

- Thin web-based client
  - Interface layer only on device
  - Business and data layers implemented using web or cloud services
- Rich client
  - All three layers (interface, business, data) implemented on device
  - Subject to mobile device limitations

# Designing Conventional Components

---

- The design of processing logic is governed by the basic principles of algorithm design and structured programming
- The design of data structures is defined by the data model developed for the system
- The design of interfaces is governed by the collaborations that a component must effect

# Component-Based Development

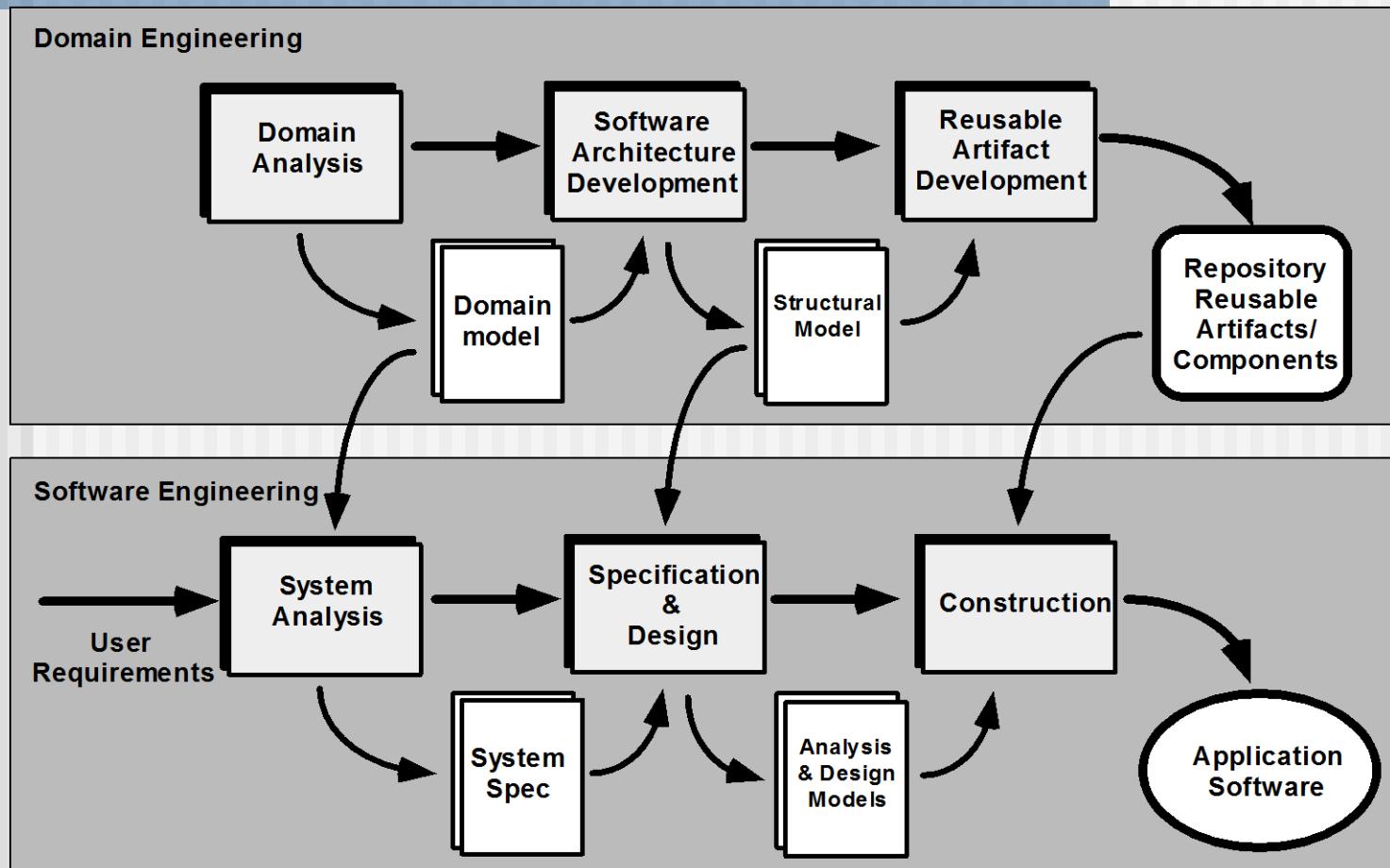
- When faced with the possibility of reuse, the software team asks:
  - Are commercial off-the-shelf (COTS) components available to implement the requirement?
  - Are internally-developed reusable components available to implement the requirement?
  - Are the interfaces for available components compatible within the architecture of the system to be built?
- At the same time, they are faced with the following impediments to reuse ...

# Impediments to Reuse

---

- Few companies and organizations have anything that even slightly resembles a comprehensive software reusability plan.
- Although an increasing number of software vendors currently sell tools or components that provide direct assistance for software reuse, the majority of software developers do not use them.
- Relatively little training is available to help software engineers and managers understand and apply reuse.
- Many software practitioners continue to believe that reuse is “more trouble than it’s worth.”
- Many companies continue to encourage of software development methodologies which do not facilitate reuse
- Few companies provide an incentives to produce reusable program components.

# The CBSE Process



# Domain Engineering

---

1. Define the domain to be investigated.
2. Categorize the items extracted from the domain.
3. Collect a representative sample of applications in the domain.
4. Analyze each application in the sample.
5. Develop an analysis model for the objects.

# Identifying Reusable Components

- Is component functionality required on future implementations?
- How common is the component's function within the domain?
- Is there duplication of the component's function within the domain?
- Is the component hardware-dependent?
- Does the hardware remain unchanged between implementations?
- Can the hardware specifics be removed to another component?
- Is the design optimized enough for the next implementation?
- Can we parameterize a non-reusable component so that it becomes reusable?
- Is the component reusable in many implementations with only minor changes?
- Is reuse through modification feasible?
- Can a non-reusable component be decomposed to yield reusable components?
- How valid is component decomposition for reuse?

# Component-Based SE

---

- a library of components must be available
- components should have a consistent structure
- a standard should exist, e.g.,
  - OMG/CORBA
  - Microsoft COM
  - Sun JavaBeans

# CBSE Activities

---

- Component qualification
- Component adaptation
- Component composition
- Component update

# Qualification

---

*Before a component can be used, you must consider:*

- application programming interface (API)
- development and integration tools required by the component
- run-time requirements including resource usage (e.g., memory or storage), timing or speed, and network protocol
- service requirements including operating system interfaces and support from other components
- security features including access controls and authentication protocol
- embedded design assumptions including the use of specific numerical or non-numerical algorithms
- exception handling

# Adaptation

---

The implication of “easy integration” is:

- (1) that consistent methods of resource management have been implemented for all components in the library;
- (2) that common activities such as data management exist for all components, and
- (3) that interfaces within the architecture and with the external environment have been implemented in a consistent manner.

# Composition

---

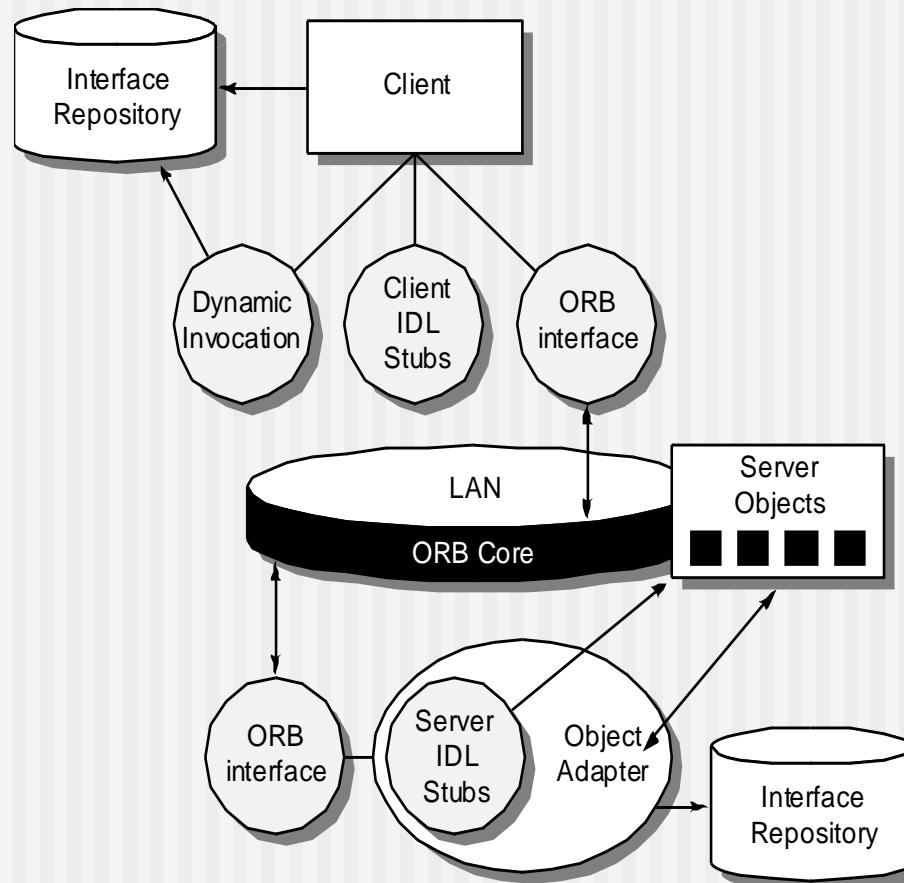
- An infrastructure must be established to bind components together
- Architectural ingredients for composition include:
  - Data exchange model
  - Automation
  - Structured storage
  - Underlying object model

# OMG/ CORBA

---

- The Object Management Group has published a *common object request broker architecture* (OMG/CORBA).
- An object request broker (ORB) provides services that enable reusable components (objects) to communicate with other components, regardless of their location within a system.
- Integration of CORBA components (without modification) within a system is assured if an interface definition language (IDL) interface is created for every component.
- Objects within the client application request one or more services from the ORB server. Requests are made via an IDL or dynamically at run time.
- An interface repository contains all necessary information about the service's request and response formats.

# ORB Architecture



# Microsoft COM

---

- The *component object model* (COM) provides a specification for using components produced by various vendors within a single application running under the Windows operating system.
- COM encompasses two elements:
  - COM interfaces (implemented as COM objects)
  - a set of mechanisms for registering and passing messages between COM interfaces.

# Sun JavaBeans

---

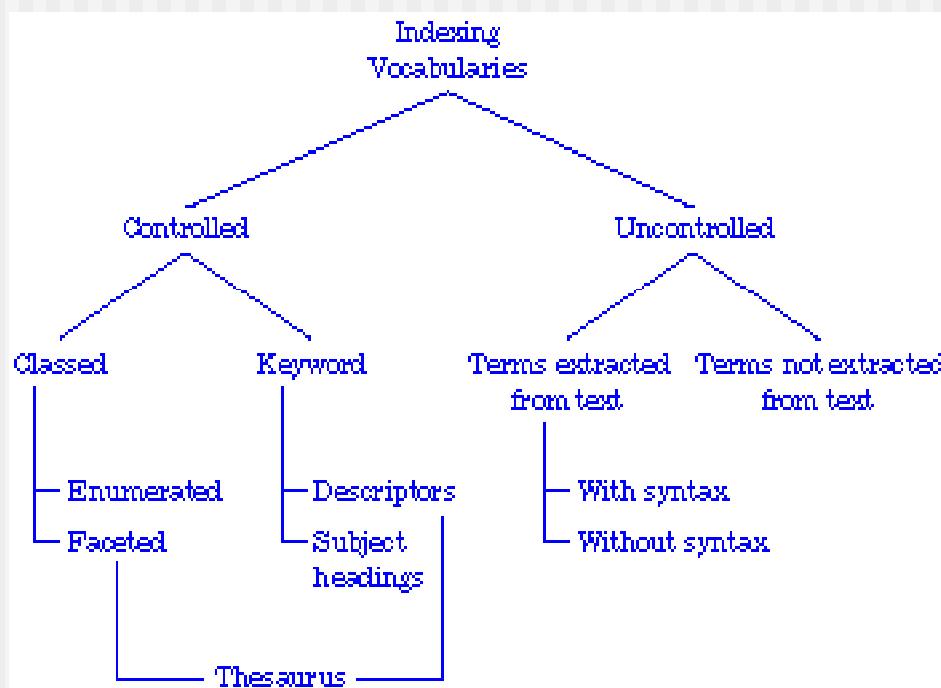
- The JavaBeans component system is a portable, platform independent CBSE infrastructure developed using the Java programming language.
- The JavaBeans component system encompasses a set of tools, called the *Bean Development Kit* (BDK), that allows developers to
  - analyze how existing Beans (components) work
  - customize their behavior and appearance
  - establish mechanisms for coordination and communication
  - develop custom Beans for use in a specific application
  - test and evaluate Bean behavior.

# Classification

---

- **Enumerated classification**—components are described by defining a hierarchical structure in which classes and varying levels of subclasses of software components are defined
- **Faceted classification**—a domain area is analyzed and a set of basic descriptive features are identified
- **Attribute-value classification**—a set of attributes are defined for all components in a domain area

# Indexing



# The Reuse Environment

---

- A component database capable of storing software components and the classification information necessary to retrieve them.
- A library management system that provides access to the database.
- A software component retrieval system (e.g., an object request broker) that enables a client application to retrieve components and services from the library server.
- CBSE tools that support the integration of reused components into a new design or implementation.

# Chapter 4 – Requirements Engineering

# Topics covered

---

- ✧ Functional and non-functional requirements
- ✧ Requirements engineering processes
- ✧ Requirements elicitation
- ✧ Requirements specification
- ✧ Requirements validation
- ✧ Requirements change

# Requirements engineering

---

- ✧ The process of establishing the services that a customer requires from a system and the constraints under which it operates and is developed.
- ✧ The system requirements are the descriptions of the system services and constraints that are generated during the requirements engineering process.

# What is a requirement?

---

- ✧ It may range from a high-level abstract statement of a service or of a system constraint to a detailed mathematical functional specification.
- ✧ This is inevitable as requirements may serve a dual function
  - May be the basis for a bid for a contract - therefore must be open to interpretation;
  - May be the basis for the contract itself - therefore must be defined in detail;
  - Both these statements may be called requirements.

# Requirements abstraction (Davis)

---

“If a company wishes to let a contract for a large software development project, it must define its needs in a sufficiently abstract way that a solution is not pre-defined. The requirements must be written so that several contractors can bid for the contract, offering, perhaps, different ways of meeting the client organization’s needs. Once a contract has been awarded, the contractor must write a system definition for the client in more detail so that the client understands and can validate what the software will do. Both of these documents may be called the requirements document for the system.”

# Types of requirement

---

## ✧ User requirements

- Statements in natural language plus diagrams of the services the system provides and its operational constraints. Written for customers.

## ✧ System requirements

- A structured document setting out detailed descriptions of the system's functions, services and operational constraints. Defines what should be implemented so may be part of a contract between client and contractor.

# User and system requirements

---

## User requirements definition

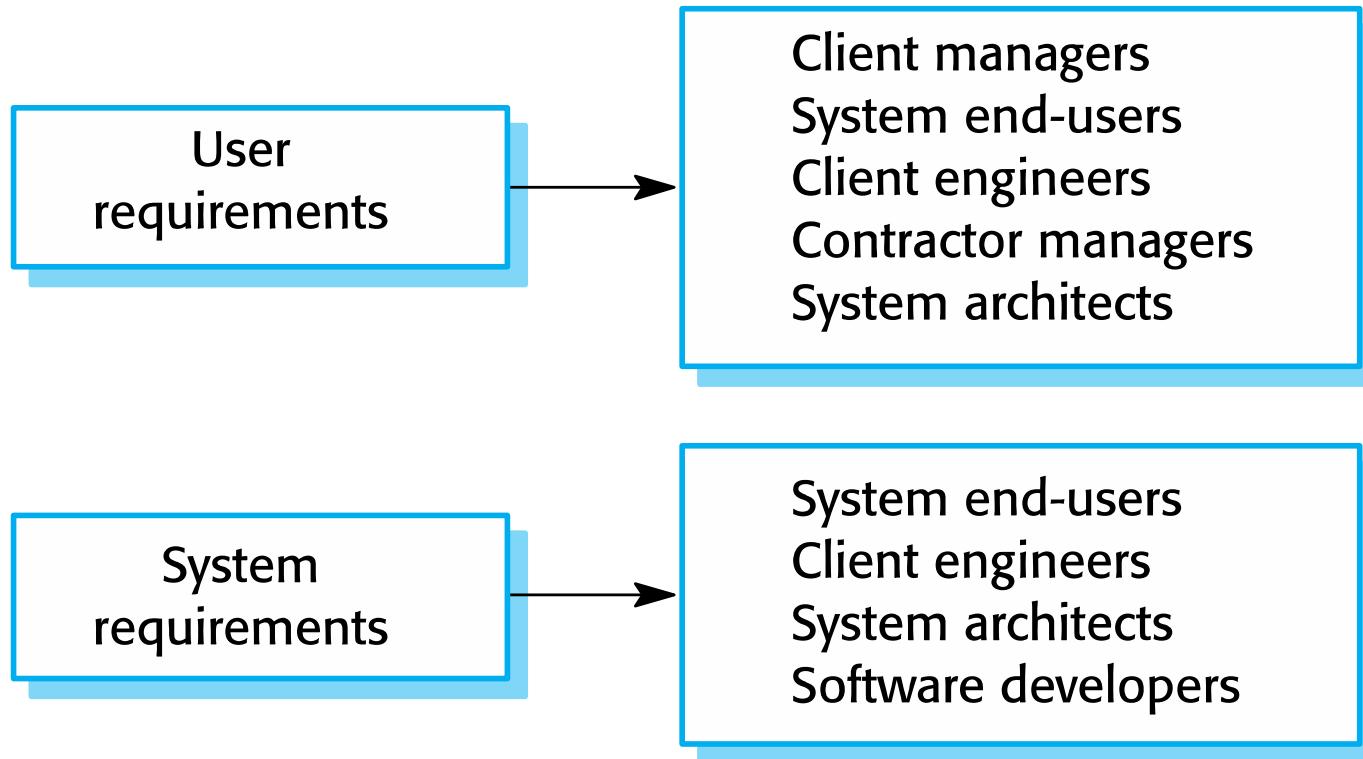
1. The Mentcare system shall generate monthly management reports showing the cost of drugs prescribed by each clinic during that month.

## System requirements specification

- 1.1 On the last working day of each month, a summary of the drugs prescribed, their cost and the prescribing clinics shall be generated.
- 1.2 The system shall generate the report for printing after 17.30 on the last working day of the month.
- 1.3 A report shall be created for each clinic and shall list the individual drug names, the total number of prescriptions, the number of doses prescribed and the total cost of the prescribed drugs.
- 1.4 If drugs are available in different dose units (e.g. 10mg, 20mg, etc) separate reports shall be created for each dose unit.
- 1.5 Access to drug cost reports shall be restricted to authorized users as listed on a management access control list.

# Readers of different types of requirements specification

---



# System stakeholders

---

- ✧ Any person or organization who is affected by the system in some way and so who has a legitimate interest
- ✧ Stakeholder types
  - End users
  - System managers
  - System owners
  - External stakeholders

# Stakeholders in the Mentcare system

---

- ✧ Patients whose information is recorded in the system.
- ✧ Doctors who are responsible for assessing and treating patients.
- ✧ Nurses who coordinate the consultations with doctors and administer some treatments.
- ✧ Medical receptionists who manage patients' appointments.
- ✧ IT staff who are responsible for installing and maintaining the system.

# Stakeholders in the Mentcare system

---

- ✧ A medical ethics manager who must ensure that the system meets current ethical guidelines for patient care.
- ✧ Health care managers who obtain management information from the system.
- ✧ Medical records staff who are responsible for ensuring that system information can be maintained and preserved, and that record keeping procedures have been properly implemented.

# Agile methods and requirements

---

- ✧ Many agile methods argue that producing detailed system requirements is a waste of time as requirements change so quickly.
- ✧ The requirements document is therefore always out of date.
- ✧ Agile methods usually use incremental requirements engineering and may express requirements as ‘user stories’ (discussed in Chapter 3).
- ✧ This is practical for business systems but problematic for systems that require pre-delivery analysis (e.g. critical systems) or systems developed by several teams.

---

# **Functional and non-functional requirements**

# Functional and non-functional requirements

---

## ✧ Functional requirements

- Statements of services the system should provide, how the system should react to particular inputs and how the system should behave in particular situations.
- May state what the system should not do.

## ✧ Non-functional requirements

- Constraints on the services or functions offered by the system such as timing constraints, constraints on the development process, standards, etc.
- Often apply to the system as a whole rather than individual features or services.

## ✧ Domain requirements

- Constraints on the system from the domain of operation

# Functional requirements

---

- ✧ Describe functionality or system services.
- ✧ Depend on the type of software, expected users and the type of system where the software is used.
- ✧ Functional user requirements may be high-level statements of what the system should do.
- ✧ Functional system requirements should describe the system services in detail.

# Mentcare system: functional requirements

---

- ✧ A user shall be able to search the appointments lists for all clinics.
- ✧ The system shall generate each day, for each clinic, a list of patients who are expected to attend appointments that day.
- ✧ Each staff member using the system shall be uniquely identified by his or her 8-digit employee number.

# Requirements imprecision

---

- ✧ Problems arise when functional requirements are not precisely stated.
- ✧ Ambiguous requirements may be interpreted in different ways by developers and users.
- ✧ Consider the term ‘search’ in requirement 1
  - User intention – search for a patient name across all appointments in all clinics;
  - Developer interpretation – search for a patient name in an individual clinic. User chooses clinic then search.

# Requirements completeness and consistency

---

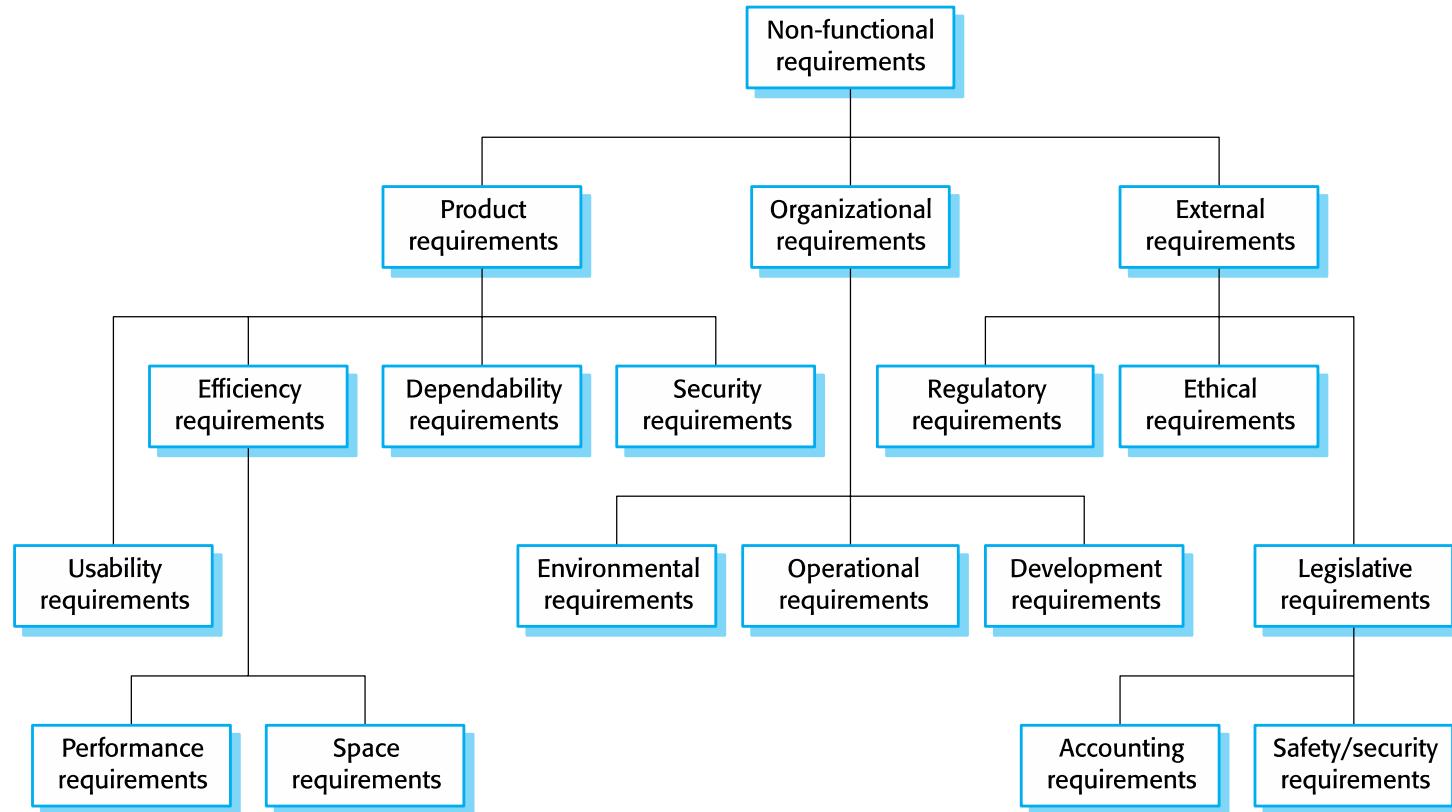
- ✧ In principle, requirements should be both complete and consistent.
- ✧ Complete
  - They should include descriptions of all facilities required.
- ✧ Consistent
  - There should be no conflicts or contradictions in the descriptions of the system facilities.
- ✧ In practice, because of system and environmental complexity, it is impossible to produce a complete and consistent requirements document.

# Non-functional requirements

---

- ✧ These define system properties and constraints e.g. reliability, response time and storage requirements. Constraints are I/O device capability, system representations, etc.
- ✧ Process requirements may also be specified mandating a particular IDE, programming language or development method.
- ✧ Non-functional requirements may be more critical than functional requirements. If these are not met, the system may be useless.

# Types of nonfunctional requirement



# Non-functional requirements implementation

---

- ✧ Non-functional requirements may affect the overall architecture of a system rather than the individual components.
  - For example, to ensure that performance requirements are met, you may have to organize the system to minimize communications between components.
- ✧ A single non-functional requirement, such as a security requirement, may generate a number of related functional requirements that define system services that are required.
  - It may also generate requirements that restrict existing requirements.

# Non-functional classifications

---

## ✧ Product requirements

- Requirements which specify that the delivered product must behave in a particular way e.g. execution speed, reliability, etc.

## ✧ Organisational requirements

- Requirements which are a consequence of organisational policies and procedures e.g. process standards used, implementation requirements, etc.

## ✧ External requirements

- Requirements which arise from factors which are external to the system and its development process e.g. interoperability requirements, legislative requirements, etc.

# Examples of nonfunctional requirements in the Mentcare system

---

## **Product requirement**

The Mentcare system shall be available to all clinics during normal working hours (Mon–Fri, 0830–17.30). Downtime within normal working hours shall not exceed five seconds in any one day.

## **Organizational requirement**

Users of the Mentcare system shall authenticate themselves using their health authority identity card.

## **External requirement**

The system shall implement patient privacy provisions as set out in HStan-03-2006-priv.

# Goals and requirements

---

- ✧ Non-functional requirements may be very difficult to state precisely and imprecise requirements may be difficult to verify.
- ✧ Goal
  - A general intention of the user such as ease of use.
- ✧ Verifiable non-functional requirement
  - A statement using some measure that can be objectively tested.
- ✧ Goals are helpful to developers as they convey the intentions of the system users.

# Usability requirements

---

- ✧ The system should be easy to use by medical staff and should be organized in such a way that user errors are minimized. (Goal)
- ✧ Medical staff shall be able to use all the system functions after four hours of training. After this training, the average number of errors made by experienced users shall not exceed two per hour of system use. (Testable non-functional requirement)

# Metrics for specifying nonfunctional requirements

Property	Measure
Speed	Processed transactions/second User/event response time Screen refresh time
Size	Mbytes Number of ROM chips
Ease of use	Training time Number of help frames
Reliability	Mean time to failure Probability of unavailability Rate of failure occurrence Availability
Robustness	Time to restart after failure Percentage of events causing failure Probability of data corruption on failure
Portability	Percentage of target dependent statements Number of target systems

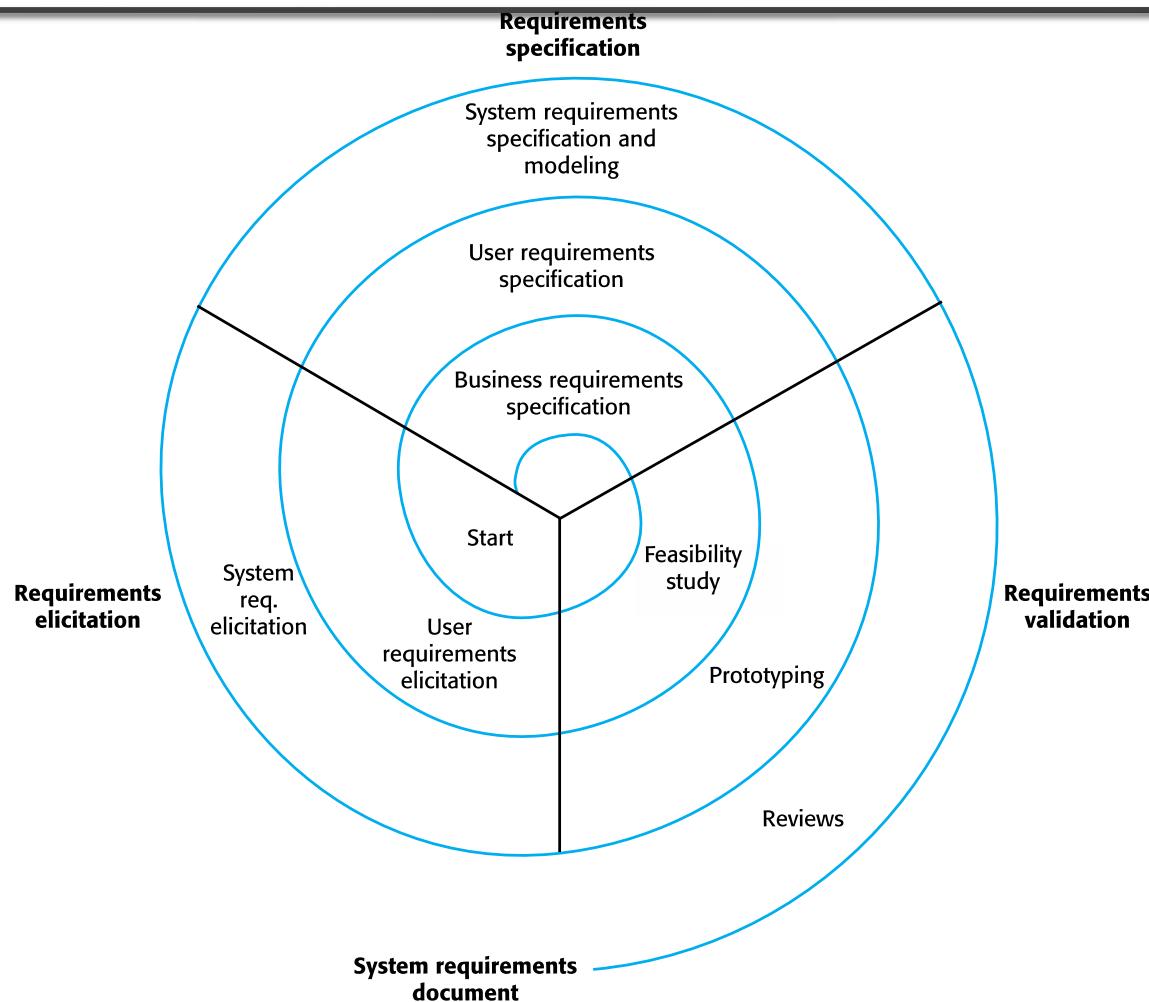
# Requirements engineering processes

# Requirements engineering processes

---

- ✧ The processes used for RE vary widely depending on the application domain, the people involved and the organisation developing the requirements.
- ✧ However, there are a number of generic activities common to all processes
  - Requirements elicitation;
  - Requirements analysis;
  - Requirements validation;
  - Requirements management.
- ✧ In practice, RE is an iterative activity in which these processes are interleaved.

# A spiral view of the requirements engineering process



# Requirements elicitation

# Requirements elicitation and analysis

---

- ✧ Sometimes called requirements elicitation or requirements discovery.
- ✧ Involves technical staff working with customers to find out about the application domain, the services that the system should provide and the system's operational constraints.
- ✧ May involve end-users, managers, engineers involved in maintenance, domain experts, trade unions, etc. These are called *stakeholders*.

# Requirements elicitation

# Requirements elicitation

---

- ✧ Software engineers work with a range of system stakeholders to find out about the application domain, the services that the system should provide, the required system performance, hardware constraints, other systems, etc.
- ✧ Stages include:
  - Requirements discovery,
  - Requirements classification and organization,
  - Requirements prioritization and negotiation,
  - Requirements specification.

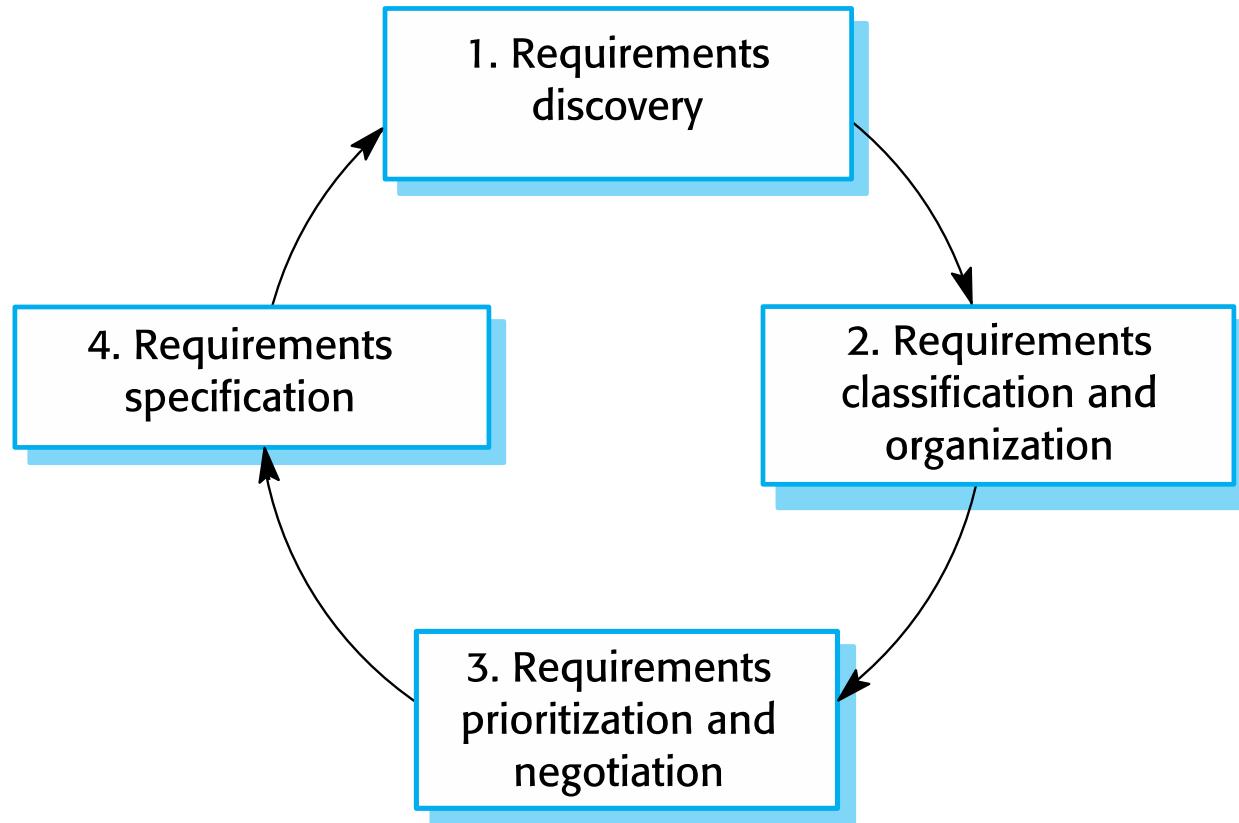
# Problems of requirements elicitation

---

- ✧ Stakeholders don't know what they really want.
- ✧ Stakeholders express requirements in their own terms.
- ✧ Different stakeholders may have conflicting requirements.
- ✧ Organisational and political factors may influence the system requirements.
- ✧ The requirements change during the analysis process.  
New stakeholders may emerge and the business environment may change.

# The requirements elicitation and analysis process

---



# Process activities

---

## ✧ Requirements discovery

- Interacting with stakeholders to discover their requirements.  
Domain requirements are also discovered at this stage.

## ✧ Requirements classification and organisation

- Groups related requirements and organises them into coherent clusters.

## ✧ Prioritisation and negotiation

- Prioritising requirements and resolving requirements conflicts.

## ✧ Requirements specification

- Requirements are documented and input into the next round of the spiral.

# Requirements discovery

---

- ✧ The process of gathering information about the required and existing systems and distilling the user and system requirements from this information.
- ✧ Interaction is with system stakeholders from managers to external regulators.
- ✧ Systems normally have a range of stakeholders.

# Interviewing

---

- ✧ Formal or informal interviews with stakeholders are part of most RE processes.
- ✧ Types of interview
  - Closed interviews based on pre-determined list of questions
  - Open interviews where various issues are explored with stakeholders.
- ✧ Effective interviewing
  - Be open-minded, avoid pre-conceived ideas about the requirements and are willing to listen to stakeholders.
  - Prompt the interviewee to get discussions going using a springboard question, a requirements proposal, or by working together on a prototype system.

# Interviews in practice

---

- ✧ Normally a mix of closed and open-ended interviewing.
- ✧ Interviews are good for getting an overall understanding of what stakeholders do and how they might interact with the system.
- ✧ Interviewers need to be open-minded without pre-conceived ideas of what the system should do
- ✧ You need to prompt the user to talk about the system by suggesting requirements rather than simply asking them what they want.

# Problems with interviews

---

- ✧ Application specialists may use language to describe their work that isn't easy for the requirements engineer to understand.
- ✧ Interviews are not good for understanding domain requirements
  - Requirements engineers cannot understand specific domain terminology;
  - Some domain knowledge is so familiar that people find it hard to articulate or think that it isn't worth articulating.

# Ethnography

---

- ✧ A social scientist spends a considerable time observing and analysing how people actually work.
- ✧ People do not have to explain or articulate their work.
- ✧ Social and organisational factors of importance may be observed.
- ✧ Ethnographic studies have shown that work is usually richer and more complex than suggested by simple system models.

# Scope of ethnography

---

- ✧ Requirements that are derived from the way that people actually work rather than the way I which process definitions suggest that they ought to work.
- ✧ Requirements that are derived from cooperation and awareness of other people's activities.
  - Awareness of what other people are doing leads to changes in the ways in which we do things.
- ✧ Ethnography is effective for understanding existing processes but cannot identify new features that should be added to a system.

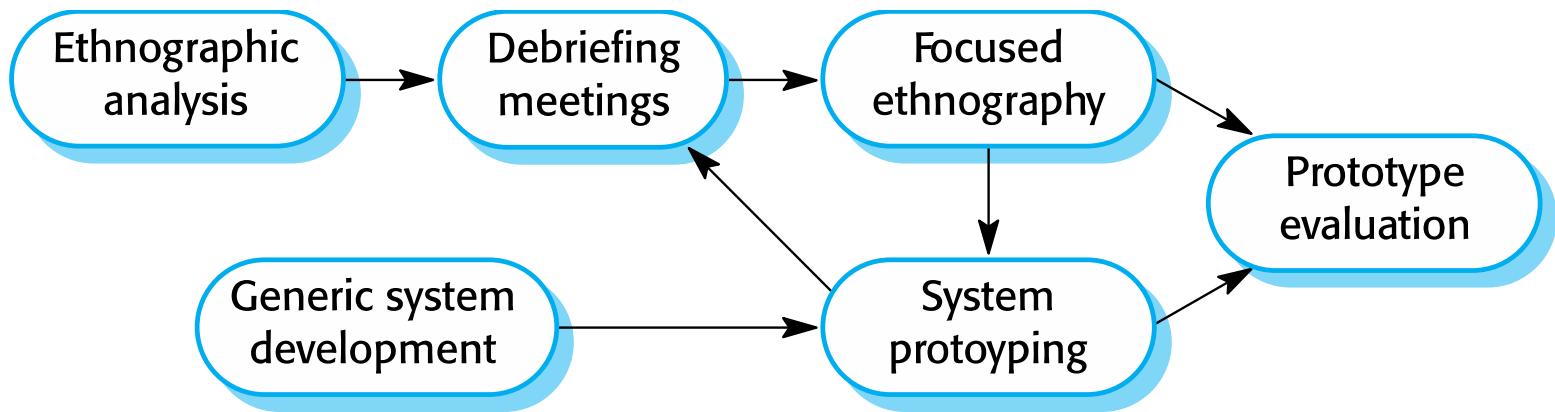
# Focused ethnography

---

- ✧ Developed in a project studying the air traffic control process
- ✧ Combines ethnography with prototyping
- ✧ Prototype development results in unanswered questions which focus the ethnographic analysis.
- ✧ The problem with ethnography is that it studies existing practices which may have some historical basis which is no longer relevant.

# Ethnography and prototyping for requirements analysis

---



# Stories and scenarios

---

- ✧ Scenarios and user stories are real-life examples of how a system can be used.
- ✧ Stories and scenarios are a description of how a system may be used for a particular task.
- ✧ Because they are based on a practical situation, stakeholders can relate to them and can comment on their situation with respect to the story.

# Scenarios

---

- ✧ A structured form of user story
- ✧ Scenarios should include
  - A description of the starting situation;
  - A description of the normal flow of events;
  - A description of what can go wrong;
  - Information about other concurrent activities;
  - A description of the state when the scenario finishes.

# Photo sharing in the classroom (iLearn)

---

- ❖ Jack is a primary school teacher in Ullapool (a village in northern Scotland). He has decided that a class project should be focused around the fishing industry in the area, looking at the history, development and economic impact of fishing. As part of this, pupils are asked to gather and share reminiscences from relatives, use newspaper archives and collect old photographs related to fishing and fishing communities in the area. Pupils use an iLearn wiki to gather together fishing stories and SCRAP (a history resources site) to access newspaper archives and photographs. However, Jack also needs a photo sharing site as he wants pupils to take and comment on each others' photos and to upload scans of old photographs that they may have in their families.

Jack sends an email to a primary school teachers group, which he is a member of to see if anyone can recommend an appropriate system. Two teachers reply and both suggest that he uses KidsTakePics, a photo sharing site that allows teachers to check and moderate content. As KidsTakePics is not integrated with the iLearn authentication service, he sets up a teacher and a class account. He uses the iLearn setup service to add KidsTakePics to the services seen by the pupils in his class so that when they log in, they can immediately use the system to upload photos from their mobile devices and class computers.

# Uploading photos iLearn)

---

- ✧ **Initial assumption:** A user or a group of users have one or more digital photographs to be uploaded to the picture sharing site. These are saved on either a tablet or laptop computer. They have successfully logged on to KidsTakePics.
- ✧ **Normal:** The user chooses upload photos and they are prompted to select the photos to be uploaded on their computer and to select the project name under which the photos will be stored. They should also be given the option of inputting keywords that should be associated with each uploaded photo. Uploaded photos are named by creating a conjunction of the user name with the filename of the photo on the local computer.
- ✧ On completion of the upload, the system automatically sends an email to the project moderator asking them to check new content and generates an on-screen message to the user that this has been done.

# Uploading photos

---

- ✧ **What can go wrong:**
- ✧ No moderator is associated with the selected project. An email is automatically generated to the school administrator asking them to nominate a project moderator. Users should be informed that there could be a delay in making their photos visible.
- ✧ Photos with the same name have already been uploaded by the same user. The user should be asked if they wish to re-upload the photos with the same name, rename the photos or cancel the upload. If they chose to re-upload the photos, the originals are overwritten. If they chose to rename the photos, a new name is automatically generated by adding a number to the existing file name.
- ✧ **Other activities:** The moderator may be logged on to the system and may approve photos as they are uploaded.
- ✧ **System state on completion:** User is logged on. The selected photos have been uploaded and assigned a status ‘awaiting moderation’. Photos are visible to the moderator and to the user who uploaded them.

---

# Requirements specification

# Requirements specification

---

- ✧ The process of writing down the user and system requirements in a requirements document.
- ✧ User requirements have to be understandable by end-users and customers who do not have a technical background.
- ✧ System requirements are more detailed requirements and may include more technical information.
- ✧ The requirements may be part of a contract for the system development
  - It is therefore important that these are as complete as possible.

# Ways of writing a system requirements specification

Notation	Description
<b>Natural language</b>	The requirements are written using numbered sentences in natural language. Each sentence should express one requirement.
Structured natural language	The requirements are written in natural language on a standard form or template. Each field provides information about an aspect of the requirement.
Design description languages	This approach uses a language like a programming language, but with more abstract features to specify the requirements by defining an operational model of the system. This approach is now rarely used although it can be useful for interface specifications.
Graphical notations	Graphical models, supplemented by text annotations, are used to define the functional requirements for the system; UML use case and sequence diagrams are commonly used.
Mathematical specifications	These notations are based on mathematical concepts such as finite-state machines or sets. Although these unambiguous specifications can reduce the ambiguity in a requirements document, most customers don't understand a formal specification. They cannot check that it represents what they want and are reluctant to accept it as a system contract

# Requirements and design

---

- ✧ In principle, requirements should state what the system should do and the design should describe how it does this.
- ✧ In practice, requirements and design are inseparable
  - A system architecture may be designed to structure the requirements;
  - The system may inter-operate with other systems that generate design requirements;
  - The use of a specific architecture to satisfy non-functional requirements may be a domain requirement.
  - This may be the consequence of a regulatory requirement.

# Natural language specification

---

- ✧ Requirements are written as natural language sentences supplemented by diagrams and tables.
- ✧ Used for writing requirements because it is expressive, intuitive and universal. This means that the requirements can be understood by users and customers.

# Guidelines for writing requirements

---

- ✧ Invent a standard format and use it for all requirements.
- ✧ Use language in a consistent way. Use shall for mandatory requirements, should for desirable requirements.
- ✧ Use text highlighting to identify key parts of the requirement.
- ✧ Avoid the use of computer jargon.
- ✧ Include an explanation (rationale) of why a requirement is necessary.

# Problems with natural language

---

- ✧ Lack of clarity
  - Precision is difficult without making the document difficult to read.
- ✧ Requirements confusion
  - Functional and non-functional requirements tend to be mixed-up.
- ✧ Requirements amalgamation
  - Several different requirements may be expressed together.

# Example requirements for the insulin pump software system

---

- 3.2 The system shall measure the blood sugar and deliver insulin, if required, every 10 minutes. (*Changes in blood sugar are relatively slow so more frequent measurement is unnecessary; less frequent measurement could lead to unnecessarily high sugar levels.*)
- 3.6 The system shall run a self-test routine every minute with the conditions to be tested and the associated actions defined in Table 1. (*A self-test routine can discover hardware and software problems and alert the user to the fact the normal operation may be impossible.*)

# Structured specifications

---

- ✧ An approach to writing requirements where the freedom of the requirements writer is limited and requirements are written in a standard way.
- ✧ This works well for some types of requirements e.g. requirements for embedded control system but is sometimes too rigid for writing business system requirements.

# Form-based specifications

---

- ✧ Definition of the function or entity.
- ✧ Description of inputs and where they come from.
- ✧ Description of outputs and where they go to.
- ✧ Information about the information needed for the computation and other entities used.
- ✧ Description of the action to be taken.
- ✧ Pre and post conditions (if appropriate).
- ✧ The side effects (if any) of the function.

# A structured specification of a requirement for an insulin pump

---

## Insulin Pump/Control Software/SRS/3.3.2

**Function** Compute insulin dose: safe sugar level.

### **Description**

Computes the dose of insulin to be delivered when the current measured sugar level is in the safe zone between 3 and 7 units.

**Inputs** Current sugar reading ( $r_2$ ); the previous two readings ( $r_0$  and  $r_1$ ).

**Source** Current sugar reading from sensor. Other readings from memory.

**Outputs** CompDose—the dose in insulin to be delivered.

**Destination** Main control loop.

# A structured specification of a requirement for an insulin pump

---

## Action

CompDose is zero if the sugar level is stable or falling or if the level is increasing but the rate of increase is decreasing. If the level is increasing and the rate of increase is increasing, then CompDose is computed by dividing the difference between the current sugar level and the previous level by 4 and rounding the result. If the result, is rounded to zero then CompDose is set to the minimum dose that can be delivered.

## Requirements

Two previous readings so that the rate of change of sugar level can be computed.

## Pre-condition

The insulin reservoir contains at least the maximum allowed single dose of insulin.

**Post-condition**      r0 is replaced by r1 then r1 is replaced by r2.

**Side effects**    None.

# Tabular specification

---

- ✧ Used to supplement natural language.
- ✧ Particularly useful when you have to define a number of possible alternative courses of action.
- ✧ For example, the insulin pump systems bases its computations on the rate of change of blood sugar level and the tabular specification explains how to calculate the insulin requirement for different scenarios.

# Tabular specification of computation for an insulin pump

---

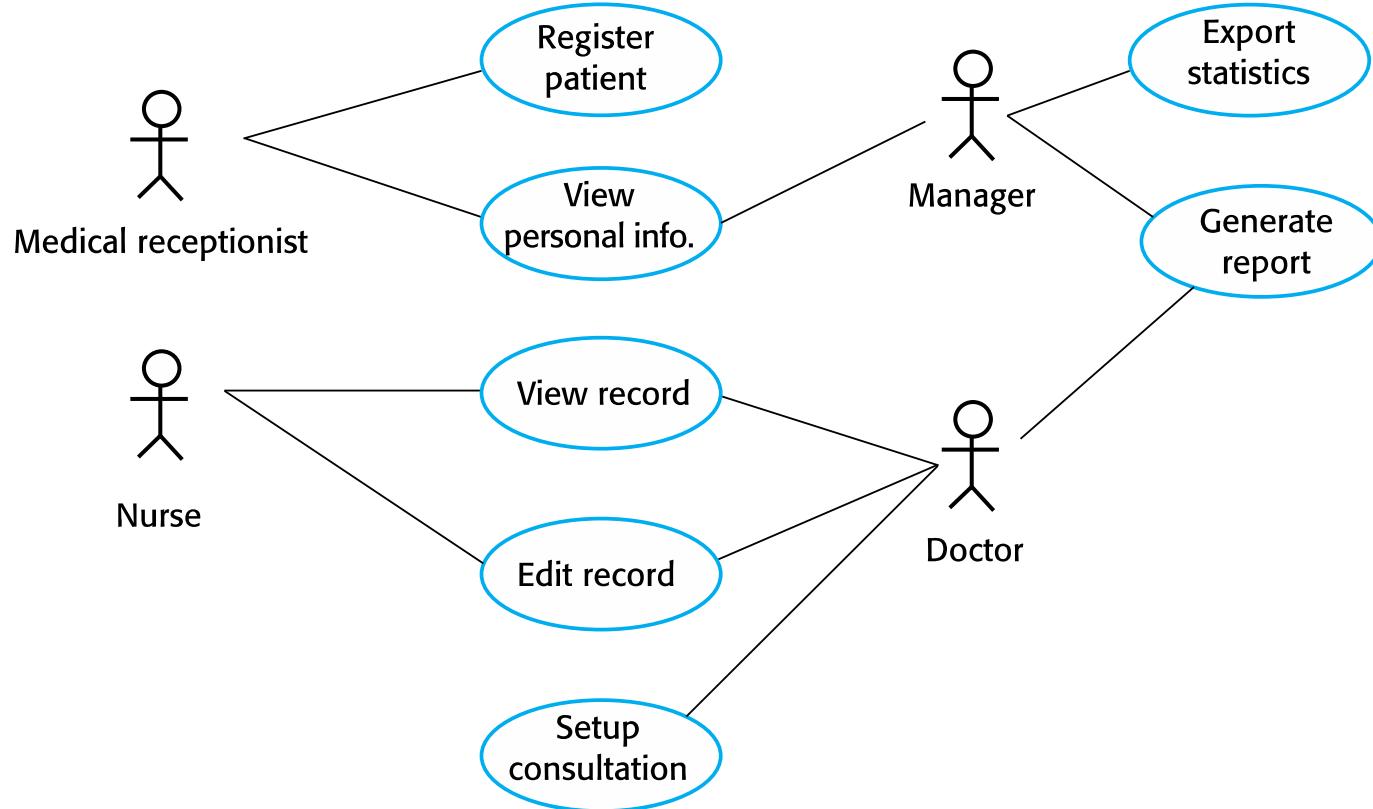
Condition	Action
Sugar level falling ( $r_2 < r_1$ )	$\text{CompDose} = 0$
Sugar level stable ( $r_2 = r_1$ )	$\text{CompDose} = 0$
Sugar level increasing and rate of increase decreasing $((r_2 - r_1) < (r_1 - r_0))$	$\text{CompDose} = 0$
Sugar level increasing and rate of increase stable or increasing $((r_2 - r_1) \geq (r_1 - r_0))$	$\text{CompDose} = \text{round}((r_2 - r_1)/4)$ If rounded result = 0 then $\text{CompDose} = \text{MinimumDose}$

# Use cases

---

- ✧ Use-cases are a kind of scenario that are included in the UML.
- ✧ Use cases identify the actors in an interaction and which describe the interaction itself.
- ✧ A set of use cases should describe all possible interactions with the system.
- ✧ High-level graphical model supplemented by more detailed tabular description (see Chapter 5).
- ✧ UML sequence diagrams may be used to add detail to use-cases by showing the sequence of event processing in the system.

# Use cases for the Mentcare system

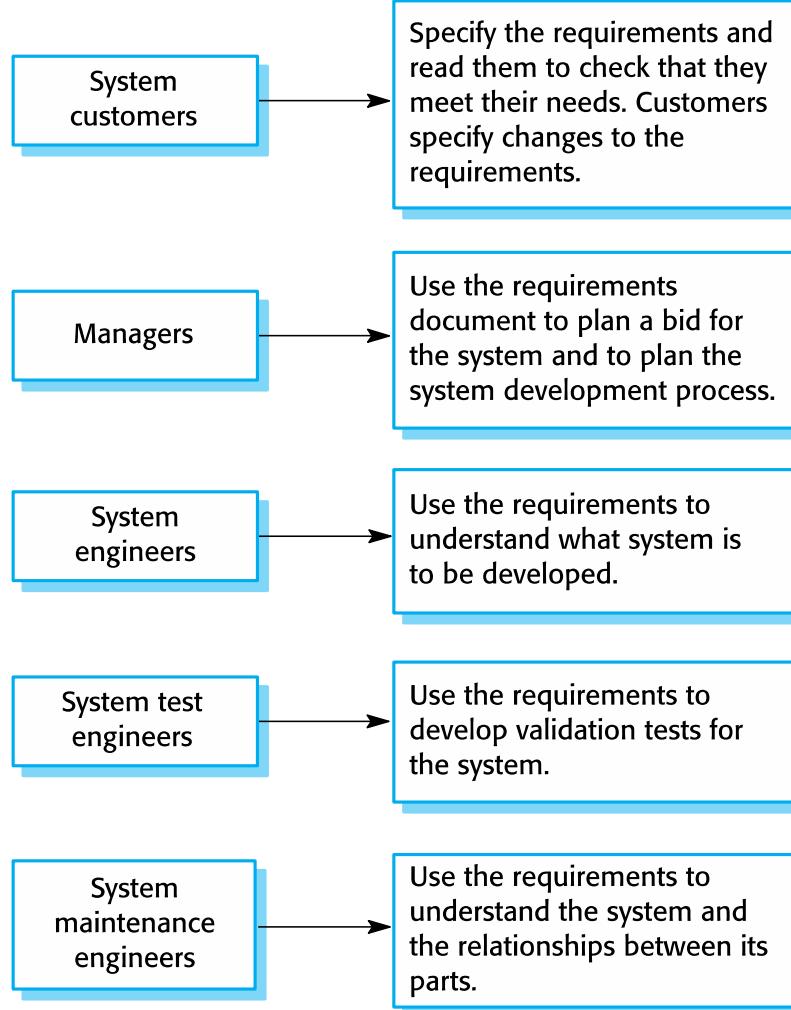


# The software requirements document

---

- ✧ The software requirements document is the official statement of what is required of the system developers.
- ✧ Should include both a definition of user requirements and a specification of the system requirements.
- ✧ It is NOT a design document. As far as possible, it should set of WHAT the system should do rather than HOW it should do it.

# Users of a requirements document



# Requirements document variability

---

- ✧ Information in requirements document depends on type of system and the approach to development used.
- ✧ Systems developed incrementally will, typically, have less detail in the requirements document.
- ✧ Requirements documents standards have been designed e.g. IEEE standard. These are mostly applicable to the requirements for large systems engineering projects.

# The structure of a requirements document

---

Chapter	Description
Preface	This should define the expected readership of the document and describe its version history, including a rationale for the creation of a new version and a summary of the changes made in each version.
Introduction	This should describe the need for the system. It should briefly describe the system's functions and explain how it will work with other systems. It should also describe how the system fits into the overall business or strategic objectives of the organization commissioning the software.
Glossary	This should define the technical terms used in the document. You should not make assumptions about the experience or expertise of the reader.
User requirements definition	Here, you describe the services provided for the user. The nonfunctional system requirements should also be described in this section. This description may use natural language, diagrams, or other notations that are understandable to customers. Product and process standards that must be followed should be specified.
System architecture	This chapter should present a high-level overview of the anticipated system architecture, showing the distribution of functions across system modules. Architectural components that are reused should be highlighted.

# The structure of a requirements document

Chapter	Description
System requirements specification	This should describe the functional and nonfunctional requirements in more detail. If necessary, further detail may also be added to the nonfunctional requirements. Interfaces to other systems may be defined.
System models	This might include graphical system models showing the relationships between the system components and the system and its environment. Examples of possible models are object models, data-flow models, or semantic data models.
System evolution	This should describe the fundamental assumptions on which the system is based, and any anticipated changes due to hardware evolution, changing user needs, and so on. This section is useful for system designers as it may help them avoid design decisions that would constrain likely future changes to the system.
Appendices	These should provide detailed, specific information that is related to the application being developed; for example, hardware and database descriptions. Hardware requirements define the minimal and optimal configurations for the system. Database requirements define the logical organization of the data used by the system and the relationships between data.
Index	Several indexes to the document may be included. As well as a normal alphabetic index, there may be an index of diagrams, an index of functions, and so on.

# Requirements validation

# Requirements validation

---

- ✧ Concerned with demonstrating that the requirements define the system that the customer really wants.
- ✧ Requirements error costs are high so validation is very important
  - Fixing a requirements error after delivery may cost up to 100 times the cost of fixing an implementation error.

# Requirements checking

---

- ✧ Validity. Does the system provide the functions which best support the customer's needs?
- ✧ Consistency. Are there any requirements conflicts?
- ✧ Completeness. Are all functions required by the customer included?
- ✧ Realism. Can the requirements be implemented given available budget and technology
- ✧ Verifiability. Can the requirements be checked?

# Requirements validation techniques

---

## ✧ Requirements reviews

- Systematic manual analysis of the requirements.

## ✧ Prototyping

- Using an executable model of the system to check requirements.  
Covered in Chapter 2.

## ✧ Test-case generation

- Developing tests for requirements to check testability.

# Requirements reviews

---

- ✧ Regular reviews should be held while the requirements definition is being formulated.
- ✧ Both client and contractor staff should be involved in reviews.
- ✧ Reviews may be formal (with completed documents) or informal. Good communications between developers, customers and users can resolve problems at an early stage.

# Review checks

---

## ✧ Verifiability

- Is the requirement realistically testable?

## ✧ Comprehensibility

- Is the requirement properly understood?

## ✧ Traceability

- Is the origin of the requirement clearly stated?

## ✧ Adaptability

- Can the requirement be changed without a large impact on other requirements?

# Requirements change

# Changing requirements

---

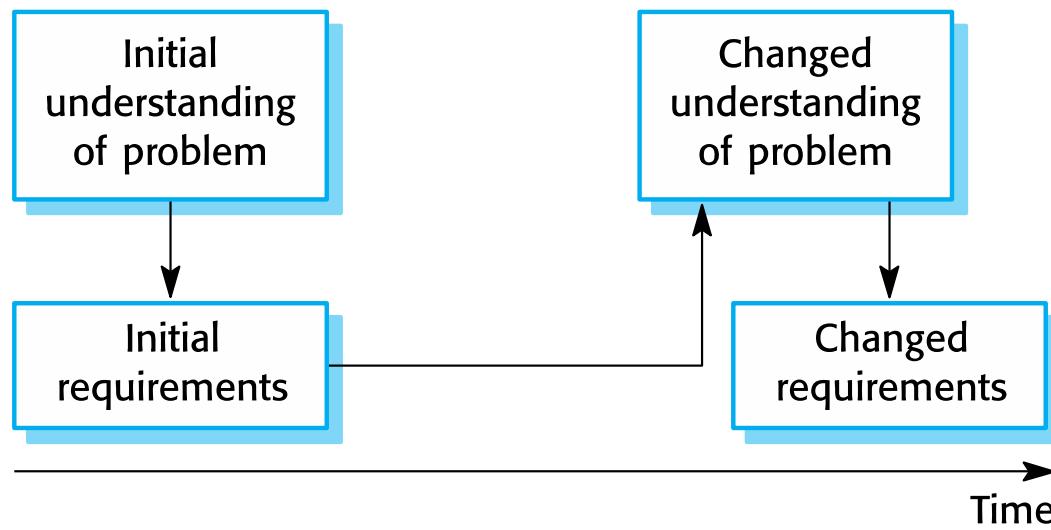
- ✧ The business and technical environment of the system always changes after installation.
  - New hardware may be introduced, it may be necessary to interface the system with other systems, business priorities may change (with consequent changes in the system support required), and new legislation and regulations may be introduced that the system must necessarily abide by.
- ✧ The people who pay for a system and the users of that system are rarely the same people.
  - System customers impose requirements because of organizational and budgetary constraints. These may conflict with end-user requirements and, after delivery, new features may have to be added for user support if the system is to meet its goals.

# Changing requirements

---

- ✧ Large systems usually have a diverse user community, with many users having different requirements and priorities that may be conflicting or contradictory.
  - The final system requirements are inevitably a compromise between them and, with experience, it is often discovered that the balance of support given to different users has to be changed.

# Requirements evolution



# Requirements management

---

- ✧ Requirements management is the process of managing changing requirements during the requirements engineering process and system development.
- ✧ New requirements emerge as a system is being developed and after it has gone into use.
- ✧ You need to keep track of individual requirements and maintain links between dependent requirements so that you can assess the impact of requirements changes. You need to establish a formal process for making change proposals and linking these to system requirements.

# Requirements management planning

---

- ✧ Establishes the level of requirements management detail that is required.
- ✧ Requirements management decisions:
  - *Requirements identification* Each requirement must be uniquely identified so that it can be cross-referenced with other requirements.
  - *A change management process* This is the set of activities that assess the impact and cost of changes. I discuss this process in more detail in the following section.
  - *Traceability policies* These policies define the relationships between each requirement and between the requirements and the system design that should be recorded.
  - *Tool support* Tools that may be used range from specialist requirements management systems to spreadsheets and simple database systems.

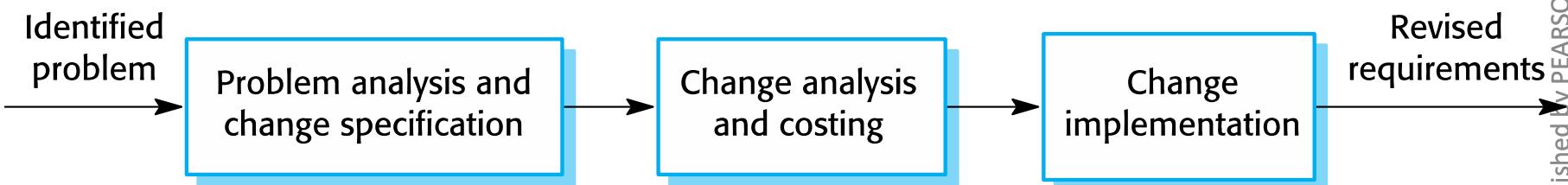
# Requirements change management

---

## ✧ Deciding if a requirements change should be accepted

- *Problem analysis and change specification*
  - During this stage, the problem or the change proposal is analyzed to check that it is valid. This analysis is fed back to the change requestor who may respond with a more specific requirements change proposal, or decide to withdraw the request.
- *Change analysis and costing*
  - The effect of the proposed change is assessed using traceability information and general knowledge of the system requirements. Once this analysis is completed, a decision is made whether or not to proceed with the requirements change.
- *Change implementation*
  - The requirements document and, where necessary, the system design and implementation, are modified. Ideally, the document should be organized so that changes can be easily implemented.

# Requirements change management



# Key points

---

- ✧ Requirements for a software system set out what the system should do and define constraints on its operation and implementation.
- ✧ Functional requirements are statements of the services that the system must provide or are descriptions of how some computations must be carried out.
- ✧ Non-functional requirements often constrain the system being developed and the development process being used.
- ✧ They often relate to the emergent properties of the system and therefore apply to the system as a whole.

# Key points

---

- ✧ The requirements engineering process is an iterative process that includes requirements elicitation, specification and validation.
- ✧ Requirements elicitation is an iterative process that can be represented as a spiral of activities – requirements discovery, requirements classification and organization, requirements negotiation and requirements documentation.
- ✧ You can use a range of techniques for requirements elicitation including interviews and ethnography. User stories and scenarios may be used to facilitate discussions.

# Key points

---

- ✧ Requirements specification is the process of formally documenting the user and system requirements and creating a software requirements document.
- ✧ The software requirements document is an agreed statement of the system requirements. It should be organized so that both system customers and software developers can use it.

## Key points

---

- ✧ Requirements validation is the process of checking the requirements for validity, consistency, completeness, realism and verifiability.
- ✧ Business, organizational and technical changes inevitably lead to changes to the requirements for a software system. Requirements management is the process of managing and controlling these changes.

---

# Unified Process

## Introduction and History

---

# Topics

---

Unified Process

Unified Process Workflows

UML (in general)

Use Cases

# What Is a Process?

---

Defines Who is doing What, When to do it, and How to reach a certain goal.



# What is the Unified Process

---

A popular iterative modern process model (framework) derived from the work on the UML and associated process.

The leading object-oriented methodology for the development of large-scale software

Maps out when and how to use the various UML techniques

# What is the Unified Process

---

Develop high-risk elements in early iterations

Deliver value to customer

Accommodate change early on in project

Work as one team

Adaptable methodology - can be modified for the specific software product to be developed

2-dimensional systems development process described by a set of phases and workflows

Utilizes Millers Law

# Miller's Law

---

At any one time, we can concentrate on only approximately seven *chunks* (units of information)

To handle larger amounts of information, use *stepwise refinement*

Concentrate on the aspects that are currently the most important

Postpone aspects that are currently less critical

# History of UP

---

Some roots in the “*Spiral Model*” of Barry Boehm

Core initial development around 1995-1998

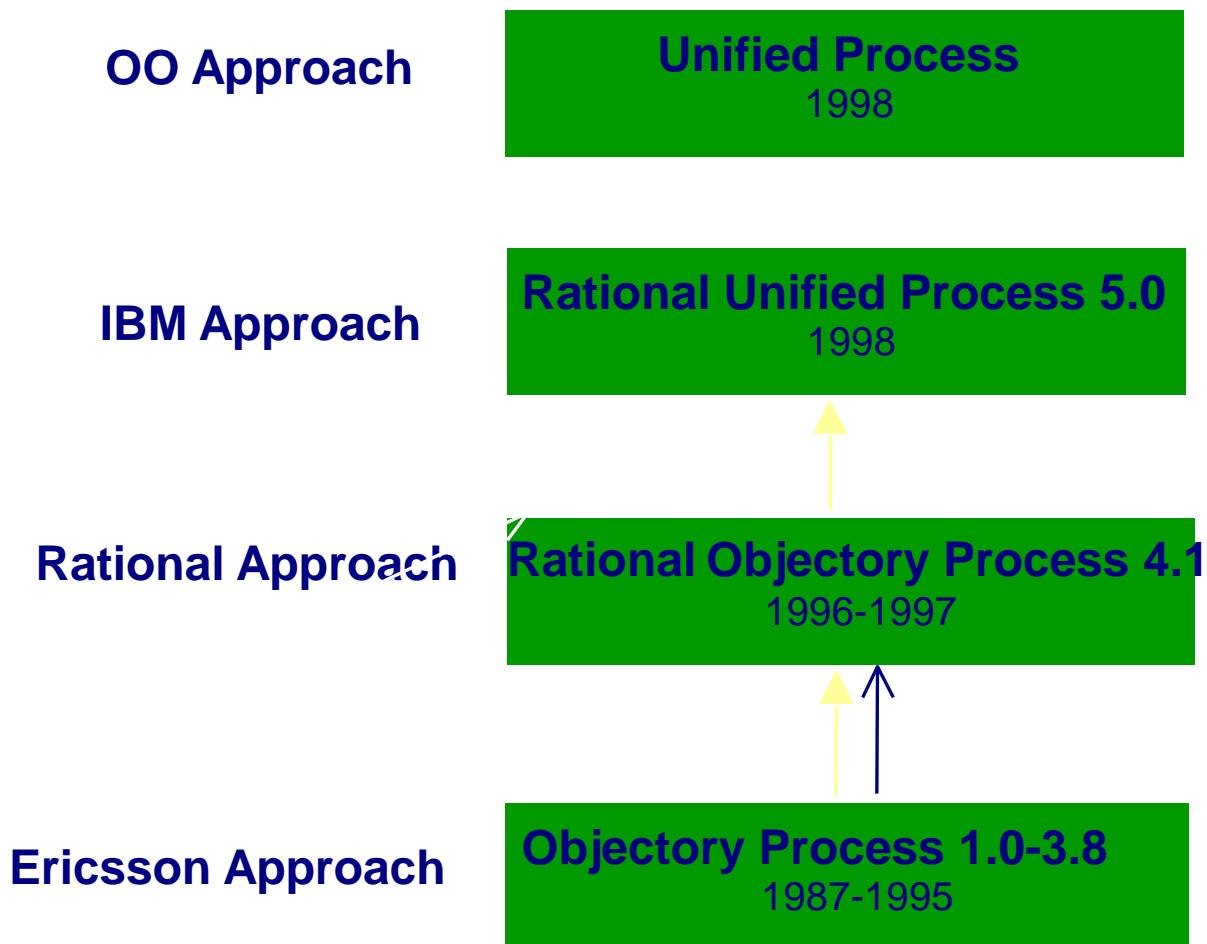
Large Canadian Air Traffic Control project as test bed

Philippe Kruchten chief architect of UP/RUP

Rational Corporation had commercial product in mind (*RUP, now owned by IBM*) but also reached out to public domain (UP)

# Creating the Unified Process

---



# Rational Unified Process (RUP)

---

Commercial version of Unified Process from IBM

A specific commercial subclass that both extends and overrides the features of the Unified Process

Supplies all of the standards, tools, and other necessities that are not included in the Unified Process

# The Rational Unified Process

---

RUP is a method of managing OO Software Development

It can be viewed as a Software Development Framework which is extensible and features:

- Iterative Development

- Requirements Management

- Component-Based Architectural Vision

- Visual Modeling of Systems

- Quality Management

- Change Control Management

# RUP Features

---

Online Repository of Process Information and Description in HTML format

Templates for all major artifacts, including:

- RequisitePro templates (requirements tracking)

- Word Templates for Use Cases

- Project Templates for Project Management

Process Manuals describing key processes

# The Unified Process

---

## In Perspective:

Most of the world is NOT object oriented and doesn't use the process we're presenting here.

However, in practice, they do something very similar that works for them.

In 1999, Booch, Jacobson, and Rumbaugh published a complete object-oriented analysis and design methodology that unified their three separate methodologies. Called the Unified Process.

The Unified Process is an adaptable methodology

It has to be modified for the specific software product to be developed

# The Unified Process (contd)

---

UML is graphical

A picture is worth a thousand words

UML diagrams enable software engineers to communicate quickly and accurately

The Unified Process is a modeling technique

A *model* is a set of UML diagrams that represent various aspects of the software product we want to develop

UML stands for unified *modeling* language

UML is the tool that we use to represent (model) the target software product

The object-oriented paradigm is iterative and incremental in nature

There is no alternative to repeated iteration and incrementation until the UML diagrams are satisfactory

# Iteration and Incrementation

---

We cannot learn the complete Unified Process in one semester or quarter

- Extensive study and unending practice are needed

- The Unified Process has too many features

- A case study of a large-scale software product is huge

In this book, we therefore cover much, but not all, of the Unified Process

- The topics covered are adequate for smaller products

To work on larger software products, experience is needed

- This must be followed by training in the more complex aspects of the Unified Process

# Unified Process Phases

# Basic Characteristics of the Unified Process

---

Object-oriented  
Use-case driven  
Architecture centric  
Iteration and incrementation

# Basic Characteristics of the Unified Process

---

## Object-oriented

Utilizes object oriented technologies.

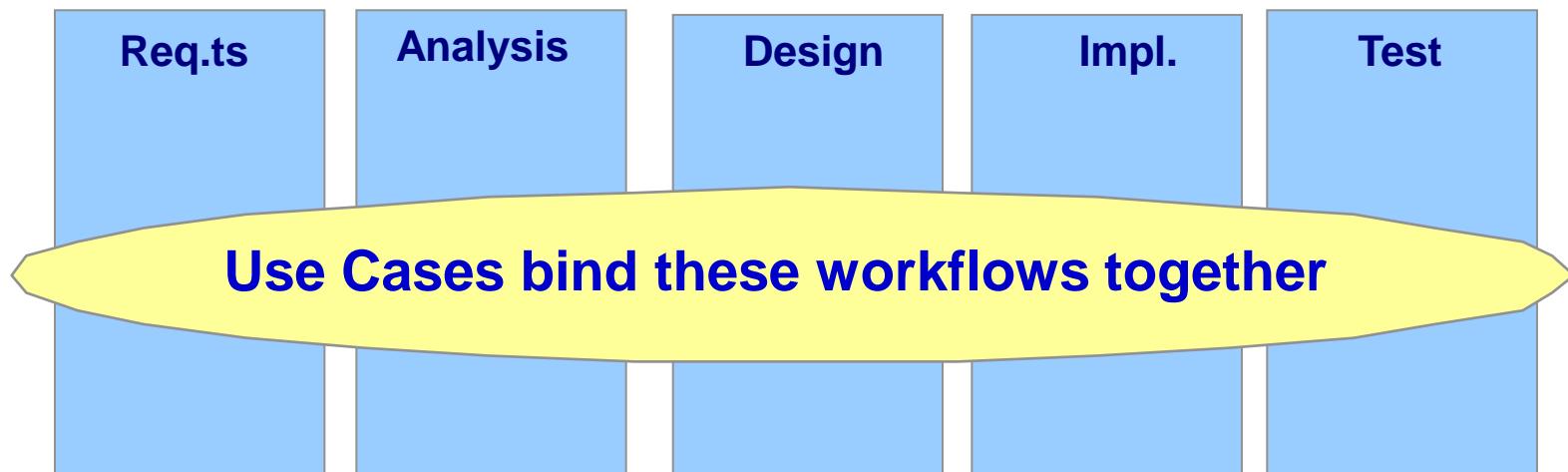
Classes are extracted during object-oriented analysis and designed during object-oriented design.

# Basic Characteristics of the Unified Process

---

## Use-case driven

Utilizes use case model to describe complete functionality of the system



# Basic Characteristics of the Unified Process

---

## Architecture centric

Embodies the most significant aspects of the system

View of the whole design with the important characteristics made more visible

Expressed with class diagram

# Basic Characteristics of the Unified Process

---

## Iteration and incrementation

Way to divide the work

Iterations are steps in the process, and increments are growth of the product

The basic software development process is iterative

Each successive version is intended to be closer to its target than its predecessor

# The Rational Unified Process

---

RUP is a method of managing OO Software Development  
It can be viewed as a Software Development Framework  
which is extensible and features:

Iterative Development

Requirements Management

Component-Based Architectural Vision

Visual Modeling of Systems

Quality Management

Change Control Management

# An Iterative Development Process...

---

Recognizes the reality of changing requirements

Caspers Jones's research on 8000 projects

40% of final requirements arrived after the analysis phase, after development had already begun

Promotes early risk mitigation, by breaking down the system into mini-projects and focusing on the riskier elements first

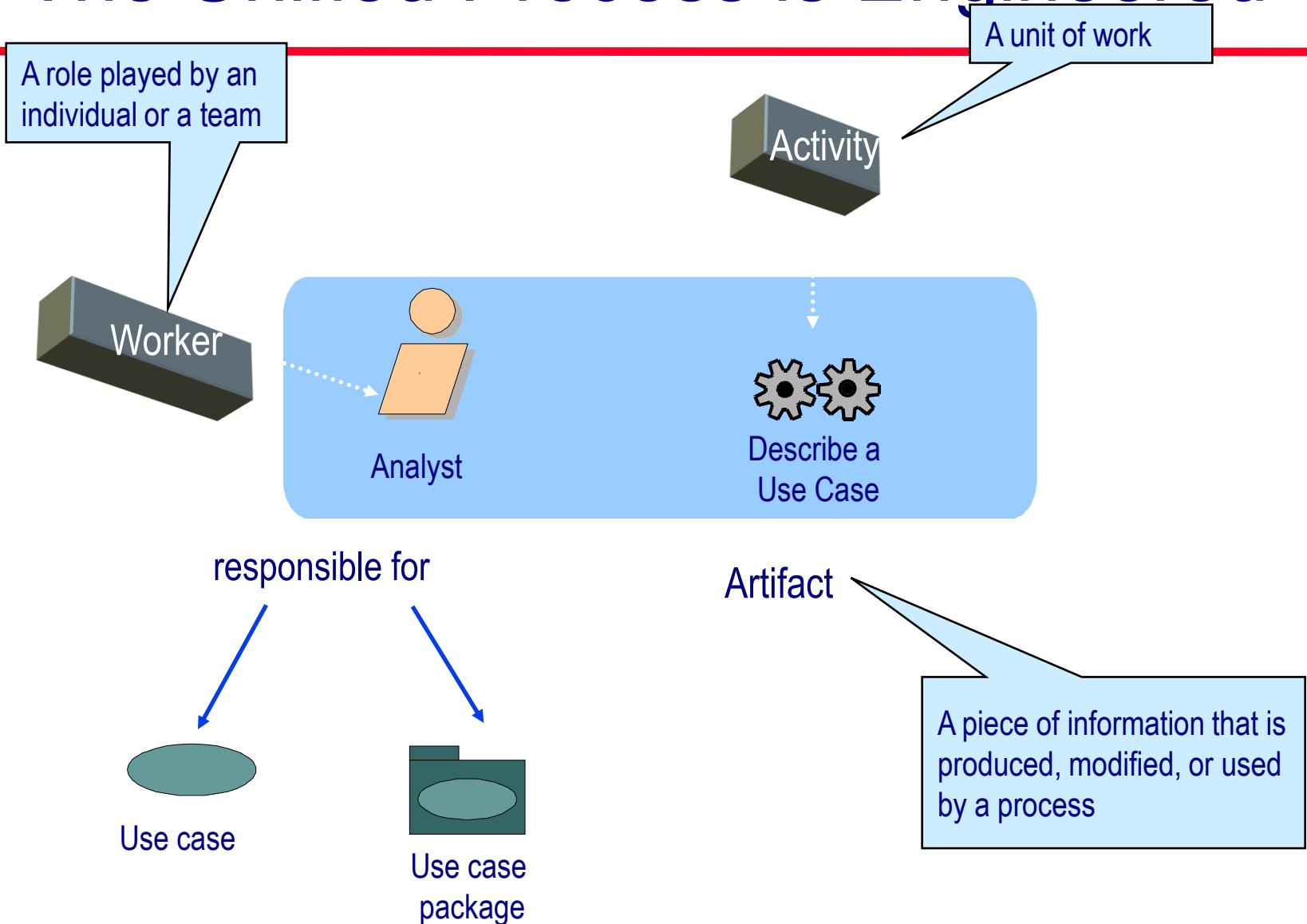
Allows you to “plan a little, design a little, and code a little”

Encourages all participants, including testers, integrators, and technical writers to be involved earlier on

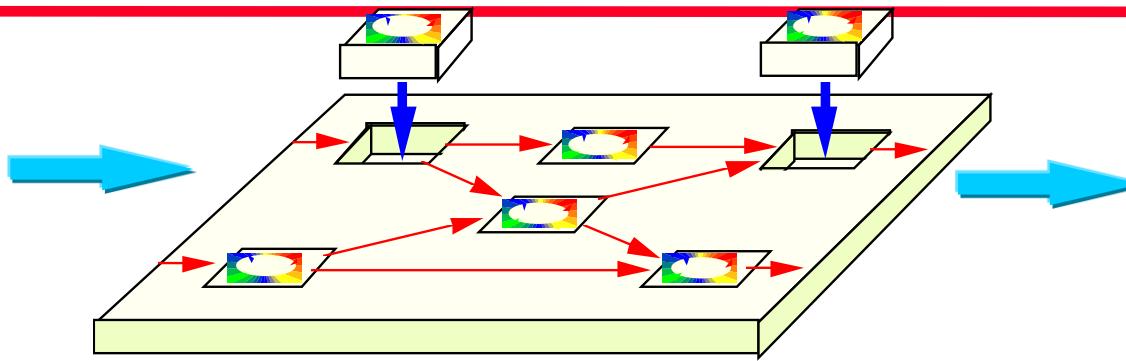
Allows the process itself to modulate with each iteration, allowing you to correct errors sooner and put into practice lessons learned in the prior iteration

Focuses on component architectures, not final big bang deployments

# The Unified Process is Engineered



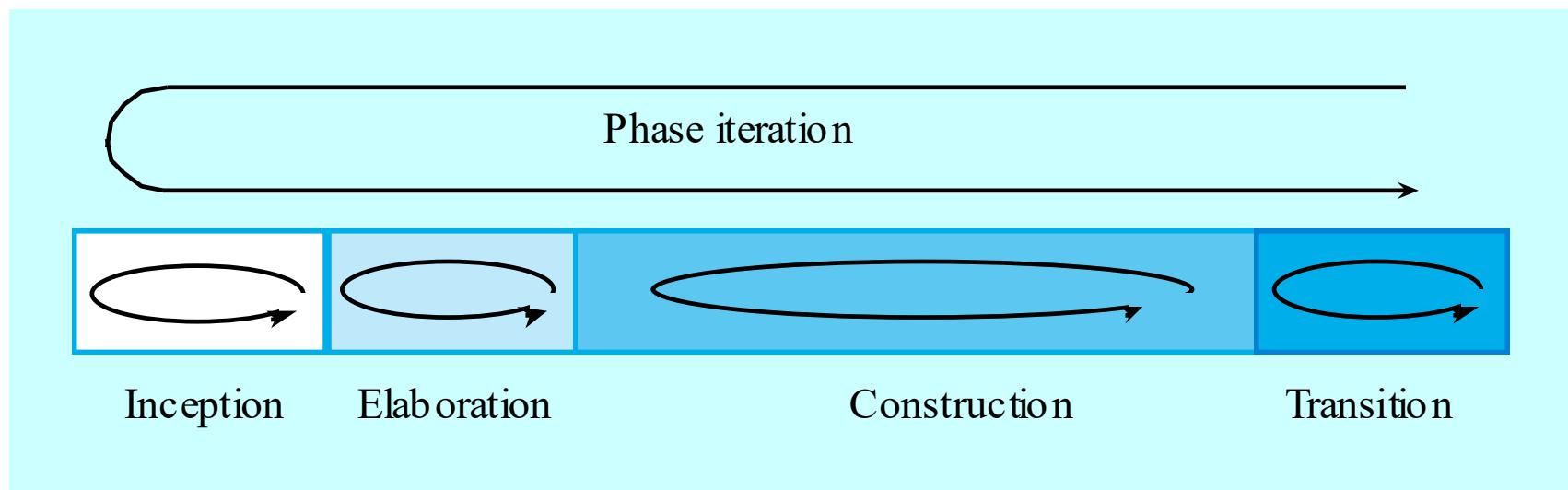
# The Unified Process is a Process Framework



There is NO Universal Process!

- The Unified Process is designed for flexibility and extensibility
  - » allows a variety of lifecycle strategies
  - » selects what artifacts to produce
  - » defines activities and workers
  - » models concepts

# Unified Process Model



# Goals and Features of Each Iteration

---

The primary goal of each iteration is to slowly chip away at the risk facing the project, namely:

- performance risks

- integration risks (different vendors, tools, etc.)

- conceptual risks (ferret out analysis and design flaws)

Perform a “minewaterfall” project that ends with a delivery of something tangible in code, available for scrutiny by the interested parties, which produces validation or correctives

Each iteration is risk-driven

The result of a single iteration is an increment--an incremental improvement of the system, yielding an evolutionary approach

# Unified Process Phases

Inception

Elaboration

Construction

Transition

## Inception

Establish the business case for the system, define risks, obtain 10% of the requirements, estimate next phase effort.

## Elaboration

Develop an understanding of the problem domain and the system architecture, risk significant portions may be coded/tested, 80% major requirements identified.

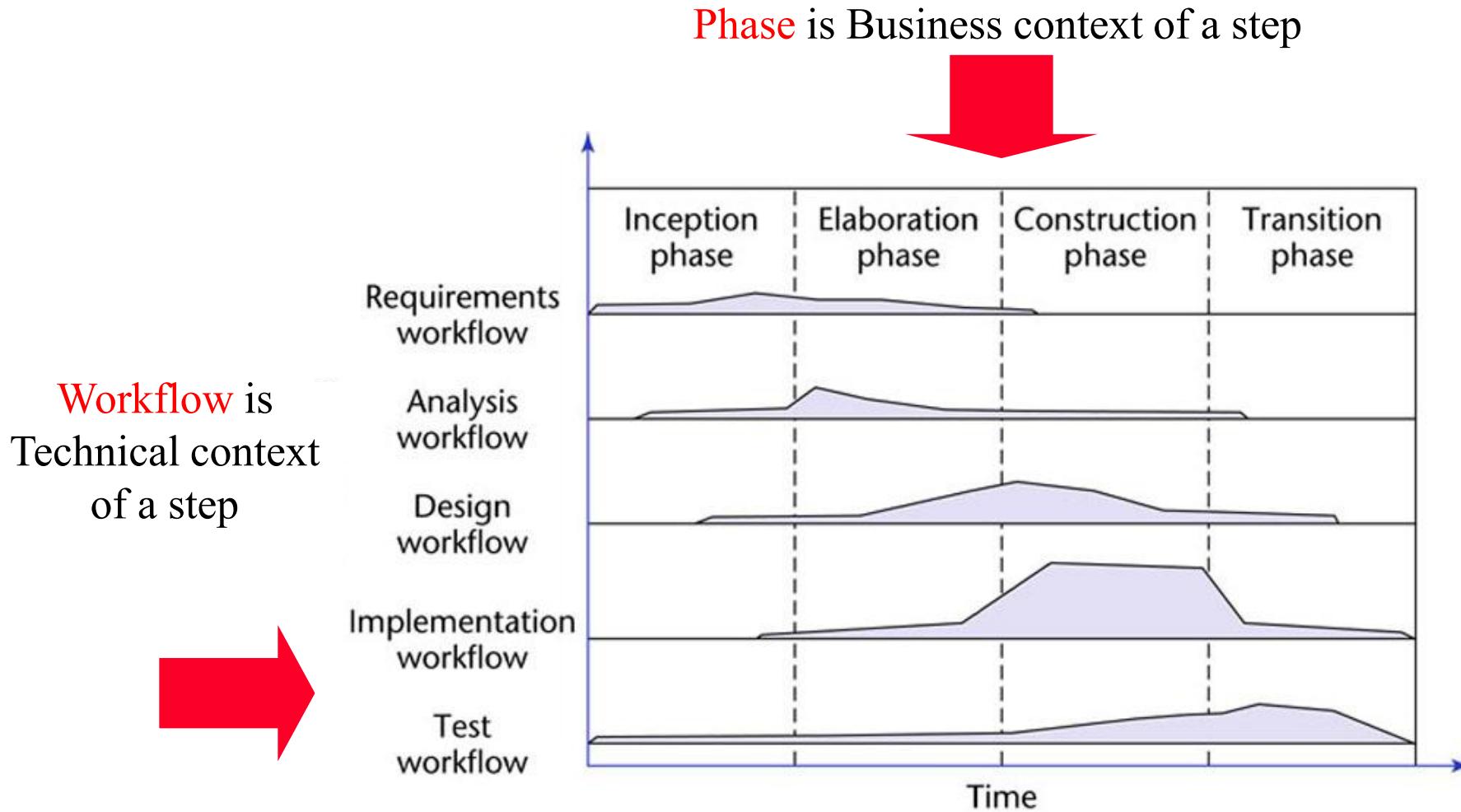
## Construction

System design, programming and testing. Building the remaining system in short iterations.

## Transition

Deploy the system in its operating environment. Deliver releases for feedback and deployment.

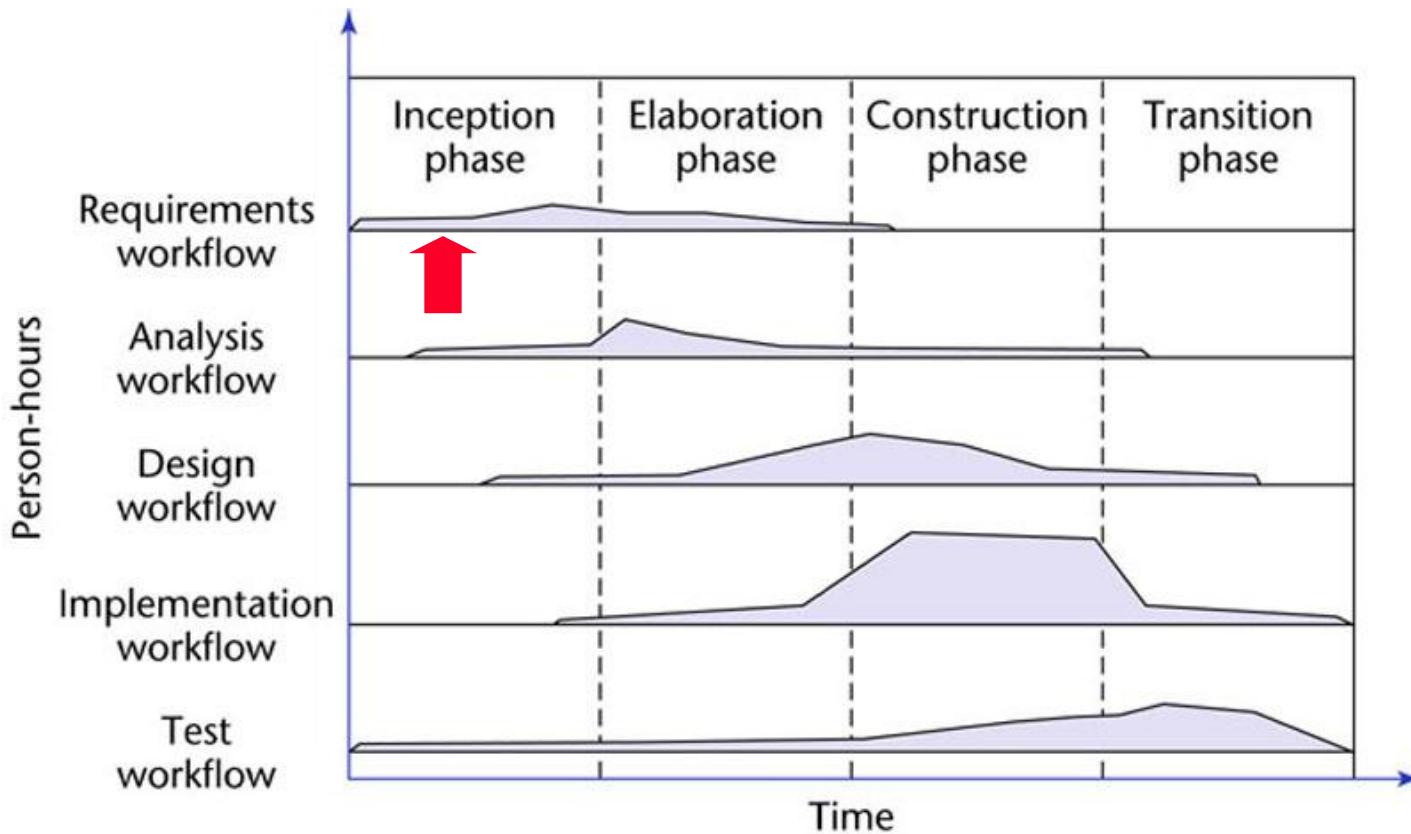
# The Phases/Workflows of the Unified Process



# The Phases/Workflows of the Unified Process

**NOTE:** Most of the requirement s work or workflow is done in the inception phase.

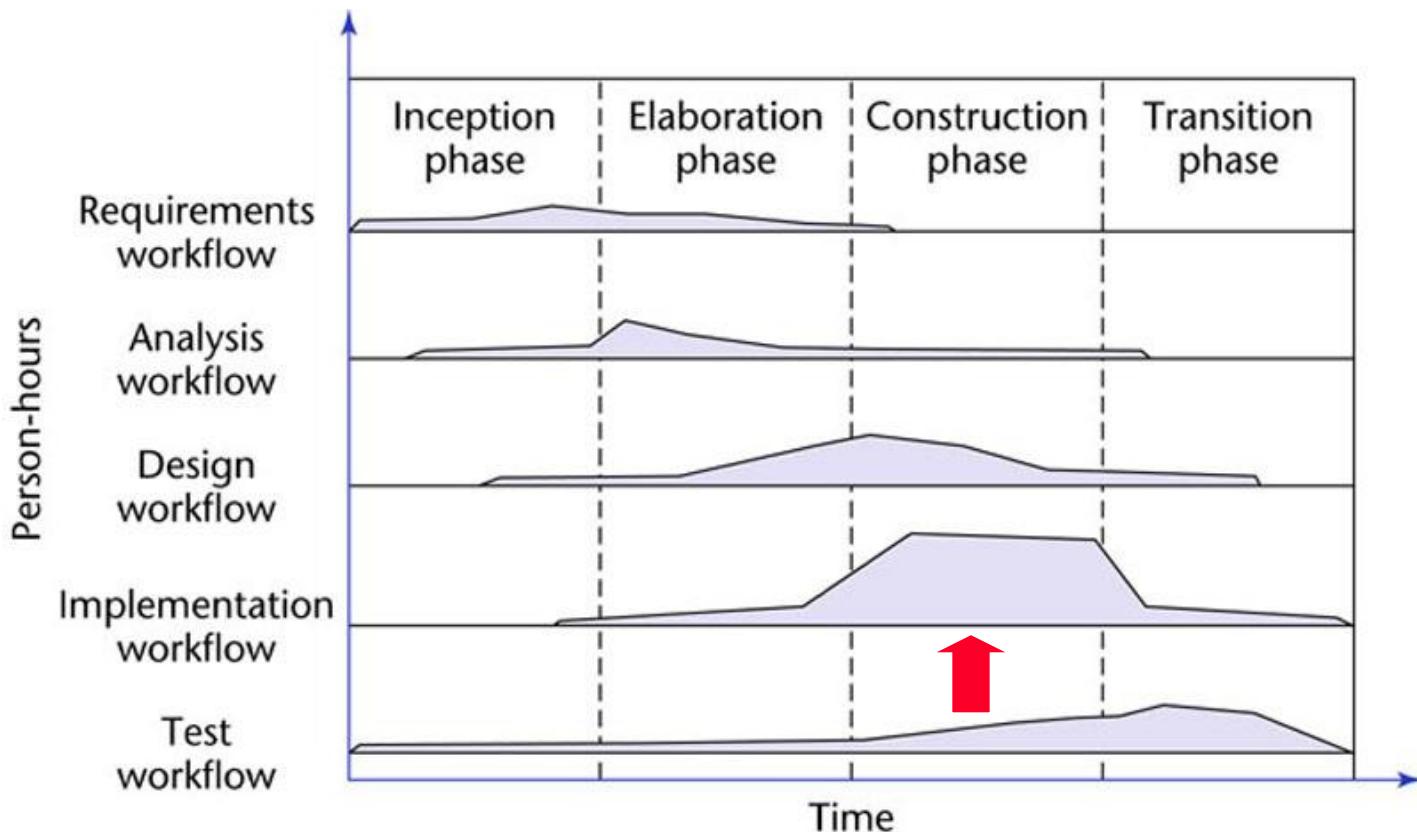
However some is done later.



# The Phases/Workflows of the Unified Process

**NOTE:** Most of the implementation work or workflow is done in construction

However some is done earlier and some later.



# Unified Process – Inception

Inception

## OVERVIEW

*Establish the business case for the system, define risks, obtain 10% to 20% of the requirements, estimate next phase effort.*

Primary Goal

Obtain buy-in from all interested parties

# Unified Process – Inception Objectives

## Inception

- Gain an understanding of the domain.
- Delimit the scope of the proposed project with a focus on the subset of the business model that is covered by the proposed software product
- Define an initial business case for the proposed system including costs, schedules, risks, priorities, and the development plan.
- Define any needed prototypes to mitigate risks.
- Obtain stakeholder concurrence on scope definition, expenditures, cost/schedule estimates, risks, development plan and priorities.

# Unified Process – Inception Activities

## Inception

- Project Initiation activities that allow new ideas to be evaluated for potential software development
- Project Planning work activities to build the team and perform the initial project planning activities
- Requirements work activities to define the business case for this potential software.
- Analysis and maybe Design work activities to define and refine costs, risks, scope, candidate architecture.
- Testing activities to define evaluation criteria for end-product vision

# Unified Process – Inception Activities

Inception

## Project Initiation

- Start with an idea
- Specify the end-product vision
- Analyze the project to assess scope
- Work the business case for the project including overall costs and schedule, and known risks
- Identify Stakeholders
- Obtain funding

# Unified Process – Inception Activities

## Inception

### Project Planning

- Build Team
- Define initial iteration
- Assess project risks and risk mitigation plan

There is insufficient information at the beginning of the inception phase to plan the entire development

The only planning that is done at the start of the project is the planning for the inception phase itself

For the same reason, the only planning that can be done at the end of the inception phase is the plan for just the next phase, the elaboration phase

# Unified Process – Inception Activities

## Inception

### Requirements

- Define or Refine Project Scope
- Begin to identify business model critical use cases of the system. (10% to 20% complete)
- Synthesize and exhibit least one candidate architectures by evaluating trade-offs, design, buy/reuse/build to refine costs.
- Prepare the supporting environment.
- Prepare development environment, selecting tools, deciding which parts of the process to improve
- Revisit estimation of overall costs and schedule.

### Analysis and maybe Design

- Define or refine costs, risks, scope, candidate architecture.
- Testing
  - Define evaluation criteria for end-product vision

# Unified Process – Inception Activities

## Inception

### Risk Assessment Activities

What are the risks involved in developing the software product, and

How can these risks be mitigated?

Does the team who will develop the proposed software product have the necessary experience?

Is new hardware needed for this software product?

If so, is there a risk that it will not be delivered in time?

If so, is there a way to mitigate that risk, perhaps by ordering back-up hardware from another supplier?

Are software tools needed?

Are they currently available?

Do they have all the necessary functionality?

# Unified Process – Inception Activities

## Inception

### Risk Assessment Activities

There are three major risk categories:

Technical risks

See earlier slide

The risk of not getting the requirements right

Mitigated by performing the requirements workflow correctly

The risk of not getting the architecture right

The architecture may not be sufficiently robust

To mitigate all three classes of risks

The risks need to be ranked so that the critical risks are mitigated first

# Unified Process – Inception Deliverables

Inception

## Primary deliverables:

- A vision document
- Initial version of the environment adoption (candidate)
- Any needed models or artifacts such as a domain model, business model, or requirements and analysis artifacts.
- Project plan, with phases and iterations with a more detailed plan for the elaboration phase.
- A project glossary
- One or several prototypes.

# Unified Process – Inception Deliverables

Inception

## Primary deliverables:

A vision document

NOTE: we use IEEE SRS Sec I,II

A general vision of the project's core requirements, key features and main constraints. Sets the scope of the project, identifies the primary requirements and constraints, sets up an initial project plan, and describes the feasibility of and risks associated with the project

Any needed models or artifacts such as a domain model, business model, or requirements and analysis artifacts.

An use-case model (10%-20% complete) – all Use Cases and Actors that can be identified so far with initial ordering.

An initial business case, which includes business context, success criteria (revenue projection, market recognition, and so on), and financial forecast;

A risk assessment analysis;

# Unified Process – Inception Questions

## Inception

- Is the proposed software product cost effective?
- How long will it take to obtain a return on investment?
- Alternatively, what will be the cost if the company decides not to develop the proposed software product?
- If the software product is to be sold in the marketplace, have the necessary marketing studies been performed?
- Can the proposed software product be delivered in time?
- If the software product is to be developed to support the client organization's own activities, what will be the impact if the proposed software product is delivered late?

# Unified Process – Elaboration Phase

Elaboration

## Elaboration

*Develop an understanding of the problem domain and the system architecture, risk significant portions may be coded/tested, 80% major requirements identified.*

The goal of the elaboration phase is to baseline the most significant requirements.

# Unified Process – Elaboration Objectives

Elaboration

## Elaboration Objectives

- *To refine the initial requirements and business case*
- *To ensure architecture, requirements, and plans are stable*
- *To monitor and address all architecturally significant risks of the project*
- *To refine or establish a baselined architecture*
- *To produce an evolutionary, throwaway, or exploratory prototypes*
- *To demonstrate that the baselined architecture will support the requirements of the system at a reasonable cost and in a reasonable time.*
- *To establish a supporting environment.*
- *To produce the project management plan*

# Unified Process – Elaboration Activities

## Elaboration

### Elaboration Essential Workflow Progress

- *All but completing the requirements workflow*
- *Performing virtually the entire analysis workflow*
- *Starting the design workflow by performing the architecture design*
- *Performing any construction workflows needed for prototypes to eliminate risks*

# Unified Process – Elaboration Activities

Elaboration

## Elaboration Essential Activities

- Analyze the problem domain.
- Define, validate and baseline the architecture
- Refine the Vision to understand the most critical Use Cases
- Create and baseline iteration plans for construction phase.
- Refine the development business case
- Put in place the development environment,
- Refine component architecture and decide build/buy/reuse
- Develop a project plan and schedule.
- Mitigate high-risk elements identified in the previous phase.

# Unified Process – Elaboration Deliverables

Elaboration

## Primary deliverables:

- Requirements model for the system
- The completed domain model (use cases, classes)
- The completed business model (costs, benefits, risks)
- The completed requirements artifacts
- The completed analysis artifacts
- Updated Architectural model
- Software project management plan

# Unified Process – Elaboration Outcomes

## Elaboration

Use Case model (at least 80% complete).

- All Use Cases identified.

- All Actors identified.

- Most Use-Case descriptions developed.

Supplementary requirements.

- (non-functional or not associated with a Use Case)

Software architecture description.

Executable architectural prototype.

Revised risk list and revised business case.

Development plan for overall project.

- coarse grained project plan, with iterations and evaluation criteria for each iteration.

Updated development case that specifies process to be used.

Preliminary user manual (optional).

# Unified Process – Elaboration Questions

Elaboration

Is the vision of the product stable?

Is the architecture stable?

Does the executable demonstration show that the major risk elements have been addressed and credibly resolved?

Is the plan for the construction phase sufficiently detailed and accurate?

Do all stakeholders agree that the current vision can be achieved if the current plan is executed to develop the complete system, in the context of the current architecture?

Is the actual resource expenditure versus planned expenditure acceptable?

# Unified Process – Construction Phase

Construction

## Construction

*System design, programming and testing. Building the remaining system in short iterations.*

The goal of the construction phase is to clarify the remaining requirements and complete the development of the first operational quality version of the software product.

# Unified Process – Construction Objectives

Construction

## Construction Objectives

- Minimizing development costs.
- Achieving adequate quality as rapidly as practical
- Achieving useful versions as rapidly as practical
- Complete analysis, design, development and testing of functionality.
- To iteratively and incrementally develop a complete product
- To decide if the software, sites, and users are deployment ready.
- To achieve parallelism in the work of development teams.

# Unified Process – Construction Activities

Construction

## Construction Essential Activities

- Complete component development and testing (beta release)
- Assess product releases against acceptance criteria for the vision. (Unit, Integration, Functional and System testing)
- Integrate all remaining components and features into the product
- Assure resource management control and process optimization

# Unified Process – Construction Deliverables

Construction

## Primary deliverables

Working software system (beta release version)

Associated documentation

Acceptance testing documentation

Updated project management deliverables (plan, risks, business case)

User Manuals

# Unified Process – Construction Outcomes

Construction

- A product ready to put into the hands of end users.
- The software product integrated on the adequate platforms.
- The user manuals.
- A description of the current release.

# Unified Process – Construction Questions

Construction

- Is this product (beta test version) release stable and mature enough to be deployed in the user community?
- Are all stakeholders ready for the transition into the user community?
- Are the actual resource expenditures versus planned expenditures still acceptable?

*Transition may have to be postponed by one release if the project fails to reach this milestone.*

# Unified Process – Transition Phase

Transition

## Transition

*Deploy the system in its operating environment.  
Deliver releases for feedback and deployment.*

The focus of the Transition Phase is to ensure that software is available for its end users and meets their needs. The Transition Phase can span several iterations, and includes testing the product in preparation for release, and making minor adjustments based on user feedback.

# Unified Process – Transition Objectives

Transition

## Transition Objectives

- Assess deployment baselines against acceptance criteria
- Achieve user self-supportability
- Achieving stakeholder concurrence of acceptance

# Unified Process – Transition Activities

Transition

## Transition Essential Activities

- Finalize end-user support material
- Test the deliverable product at the development site
- Validate beta test to assure user expectations met
- Fine-tune the product based on feedback
- Perform parallel operation of replaced legacy system
- Convert operational databases
- Train of users and maintainers
- Roll-out to the marketing, distribution and sales forces
- Perform deployment engineering (cutover, roll-out performance tuning)

# Unified Process – Transition Deliverables

Transition

## Primary deliverable

Final product onto a production platform

## Other deliverables

All the artifacts (final versions)

Completed manual

# Phase Deliverables

Inception Phase	Elaboration Phase	Construction Phase	Transition Phase
<ul style="list-style-type: none"><li>• The initial version of the domain model</li><li>• The initial version of the business model</li><li>• The initial version of the requirements artifacts</li><li>• A preliminary version of the analysis artifacts</li><li>• A preliminary version of the architecture</li><li>• The initial list of risks</li><li>• The initial ordering of the use cases</li><li>• The plan for the elaboration phase</li><li>• The initial version of the business case</li></ul>	<ul style="list-style-type: none"><li>• The completed domain model</li><li>• The completed business model</li><li>• The completed requirements artifacts</li><li>• The completed analysis artifacts</li><li>• An updated version of the architecture</li><li>• An updated list of risks</li><li>• The project management plan (for the rest of the project)</li><li>• The completed business case</li></ul>	<ul style="list-style-type: none"><li>• The initial user manual and other manuals, as appropriate</li><li>• All the artifacts (beta release versions)</li><li>• The completed architecture</li><li>• The updated risk list</li><li>• The project management plan (for the remainder of the project)</li><li>• If necessary, the updated business case</li></ul>	<ul style="list-style-type: none"><li>• All the artifacts (final versions)</li><li>• The completed manuals</li></ul>

# UP Life cycle in four phases

---

- Inception
- Elaboration
- Construction
- Transition

The Enterprise Unified Process (EUP) adds two more phases to this:

*Production:* keep system useful/productive after deployment to customer

*Retirement:* archive, remove, or reuse etc.

# Example roles in UP

---

*Stake Holder:* customer, product manager, etc.

*Software Architect:* established and maintains architectural vision

*Process Engineer:* leads definition and refinement of Development Case

*Graphic Artist:* assists in user interface design, etc.

# Some UP guidelines

---

Attack risks early on and continuously so, before they will attack you

Stay focused on *developing executable software* in early iterations

Prefer component-oriented architectures and reuse existing components

Quality is a way of life, not an afterthought

# Six best “must” UP practices

---

Time-boxed iterations: *avoid speculative powerpoint architectures*

*Strive for cohesive architecture* and reuse existing components:

e.g. core architecture developed by small, co-located team

then early team members divide into sub-project leaders

# Six best “must” UP practices

---

Continuously verify quality: test early & often, and realistically by integrating all software at each iteration

Visual modeling: prior to programming, do at least some visual modeling to explore creative design ideas

# Six best “must” UP practices

---

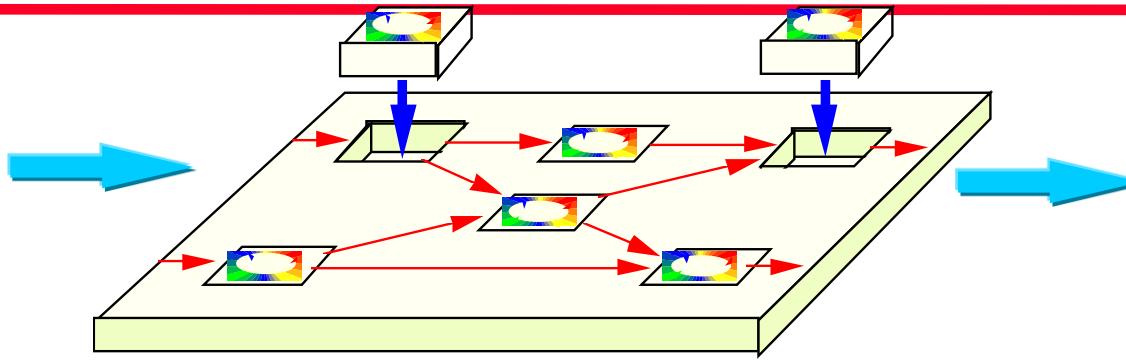
Manage requirements: find, organize, and track requirements through skillful means

Manage change:

disciplined configuration management protocol, version control,  
change request protocol  
baselined releases at iteration ends

# Unified Process Workflows

# The Unified Process is a Process Framework



While the Unified Process is widely used, there is NO Universal Process!

- The Unified Process is designed for flexibility and extensibility
  - » allows a variety of lifecycle strategies
  - » selects what artifacts to produce
  - » defines activities and workers
  - » models concepts
  - » **IT IS A PROCESS FRAMEWORK for development**

# The Unified Process

---

The Unified Process IS A  
2-dimensional systems development  
process described by a  
set of phases and (dimension one)  
Workflows (dimension two)

# The Unified Process

---

## Phases

Describe the business steps needed to develop, buy, and pay for software development.

The business increments are identified as phases

## Workflows

Describe the tasks or activities that a developer performs to evolve an information system over time

# Why a Two-Dimensional Model?

---

In an ideal world, each workflow would be completed before the next workflow is started

In reality, the development task is too big for this

As a consequence of Miller's Law

The development task has to be divided into increments (phases) Within each increment, iteration is performed until the task is complete

At the beginning of the process, there is not enough information about the software product to carry out the requirements workflow

Similarly for the other core workflows

# Why a Two-Dimensional Model?

---

A software product has to be broken into subsystems. Even subsystems can be too large at times. Modules may be all that can be handled until a fuller understanding of all the parts of the product as a whole has been obtained

The Unified Process handles the inevitable changes well

- The moving target problem

- The inevitable mistakes

The Unified Process works for treating a large problem as a set of smaller, largely independent sub problems

- It provides a framework for incrementation and iteration

- In the future, it will inevitably be superseded by some better methodology

# Process Overview

	Phases (time)			
Workflow (tasks)	Inception	Elaboration	Construction	Transition
Requirements				
Analysis				
Design				
Implementation				
Test				

# Static workflows

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.
Test	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 managed changes to the system (see Chapter 29).
Project management	This supporting workflow manages the system development (see Chapter 5).
Environment	This workflow is concerned with making appropriate software tools available to the software development team.

# Primary Workflows

---

The Unified Process

## PRIMARY WORKFLOWS

**Requirements workflow**

**Analysis workflow**

**Design workflow**

**Implementation workflow**

**Test workflow**

**Post delivery maintenance workflow**

## Supplemental Workflows

**Planning Workflow**

# Planning Workflow

---

Define scope of Project

Define scope of next iteration

Identify Stakeholders

Capture Stakeholders expectation

Build team

Assess Risks

Plan work for the iteration

Plan work for Project

Develop Criteria for iteration/project closure/success

UML concepts used: initial Business Model, using class diagram

# Requirements Workflow

---

## Primary focus

To determine the client's needs by eliciting both functional and nonfunctional requirements

Gain an understanding of the *application domain*

Described in the language of the customer

# Requirements Workflow

---

The aim is to determine the client's needs

First, gain an understanding of the *domain*

How does the specific business environment work

Second, build a business model

Use UML to describe the client's business processes

If at any time the client does not feel that the cost is justified,  
development terminates immediately

It is vital to determine the client's constraints

Deadline -- Nowadays software products are often mission critical

Parallel running

Portability

Reliability

Rapid response time

Cost

The aim of this *concept exploration* is to determine

What the client needs, and

*Not* what the client wants

# Requirements Workflow

---

List candidate requirements  
textual feature list

Understand system context

domain model describing important concepts of the context

business modeling specifying what processes have to be supported by the system using Activity Diagram

Capture functional and nonfunctional requirements

Use Case Model

Supplementary requirements

physical, interface, design constraints, implementation constraints

# Analysis Workflow

---

## Primary focus

Analyzing and refining the requirements to achieve a detailed understanding of the requirements essential for developing a software product correctly

To ensure that both the developer and user organizations understand the underlying problem and its domain

Written in a more precise language

# Analysis Workflow

The aim of the analysis workflow

To analyze and refine the requirements

Two separate workflows are needed

The requirements artifacts must be expressed in the language of the client

The analysis artifacts must be precise, and complete enough for the designers

Specification document (“specifications”)

Constitutes a contract

It must not have imprecise phrases like “optimal,” or “98 percent complete”

Having complete and correct specifications is essential for

Testing, and

Maintenance

The specification document must not have

Contradictions

Omissions

Incompleteness

# Analysis Workflow

Structure the Use Cases

Start reasoning about the internal of the system

Develop Analysis Model: Class Diagram and State Diagram

Focus on what is the problem not how to solve it

Understand the main concepts of the problem

Three main types of classes stereotypes may be used:

Boundary Classes: used to model interaction between system and actors

Entity Classes: used to model information and associated behavior directly derived from real-world concept

Control Class: used to model business logic, computations transactions or coordination.

The specification document must not have

Contradictions

Omissions

Incompleteness

# Design Workflow

---

The aim of the design workflow is to refine the analysis workflow until the material is in a form that can be implemented by the programmers

Determines the internal structure of the software product

# Design Workflow

---

The goal is to refine the analysis workflow until the material is in a form that can be implemented by the programmers

Many nonfunctional requirements need to be finalized at this time, including: Choice of programming language, Reuse issues, Portability issues.

## Classical Design

### Architectural design

Decompose the product into modules

### Detailed design

Design each module using data structures and algorithms

## Object Oriented Design

Classes are extracted during the object-oriented analysis workflow, and

Designed during the design workflow

# Design Workflow

## General Design

Refine the Class Diagram

Structure system with Subsystems, Interfaces, Classes

Define subsystems dependencies

Capture major interfaces between subsystems

Assign responsibilities to new design classes

Describe realization of Use Cases

Assign visibility to class attributes

Design Databases and needed Data Structures

Define Methods signature

Develop state diagram for relevant design classes

Use Interaction Diagram to distribute behavior among classes

Use Design Patterns for parts of the system

# Design Workflow

## Architectural Design

### Identify Design Mechanisms

Refine Analysis based on implementation environment

Characterize needs for specific mechanisms (inter-process communication, real-time computation, access to legacy system, persistence, ...)

Assess existing implementation mechanisms

### Identify Design Classes and Subsystems

A Subsystem is a special kind of Package which has behavioral semantics (realizes one or more interfaces)

Refine analysis classes

Group classes into Packages

Identify Subsystems when analysis classes are complex

- Look for strong interactions between classes
- Try to organize the UI classes into a subsystem
- Separate functionality used by different actors in different subsystems
- Separate subsystems based on the distribution needs

Identify Interfaces of the subsystems

# Implementation Workflow

---

The aim of the implementation workflow is to implement the target software product in the selected implementation language

# Implementation Workflow

---

Distribute the system by mapping executable components onto nodes in the deployment model

Implement Design Classes and subsystems through packaging mechanism:

package in Java, Project in VB, files  
directory in C++

Acquire external components realizing needed interfaces

Unit test the components

Integrate via builds

Requirements

Analysis

Design

Implementation

Testing

# Test Workflow

---

Carried out in parallel with other workflows

Primary purpose

To increase the quality of the evolving system

The test workflow is the responsibility of

Every developer and maintainer

Quality assurance group

# Test Workflow

---

Develop set of test cases that specify what to test in the system

many for each Use Case  
each test case will verify one scenario of the use case  
based on Sequence Diagram

Develop test procedures specifying how to perform test cases

Develop test component that automates test procedures

# Deployment Workflow

---

Activities include

Software packaging

Distribution

Installation

Beta testing

# Deployment Workflow

---

## Producing the Software

Output of implementation is tested executables.

Must be associated with other artifacts to constitute a complete product:

- Installation scripts

- User documentation

- Configuration data

- Additional programs for migration: data conversion.

In some cases:

- different executables needed for different user configurations

- different sets of artifacts needed for different classes of users:

  - new users versus existing users,

  - variants by country or language

# Deployment Workflow

---

Producing the Software (continued)

For distributed software, different sets may have to be produced for different computing nodes in the network

Packaging the Software

Distributing the Software

Installing the Software

Migration

Providing Help and Assistance to Users

Acceptance

# Iterations and Workflow

## Core Workflows

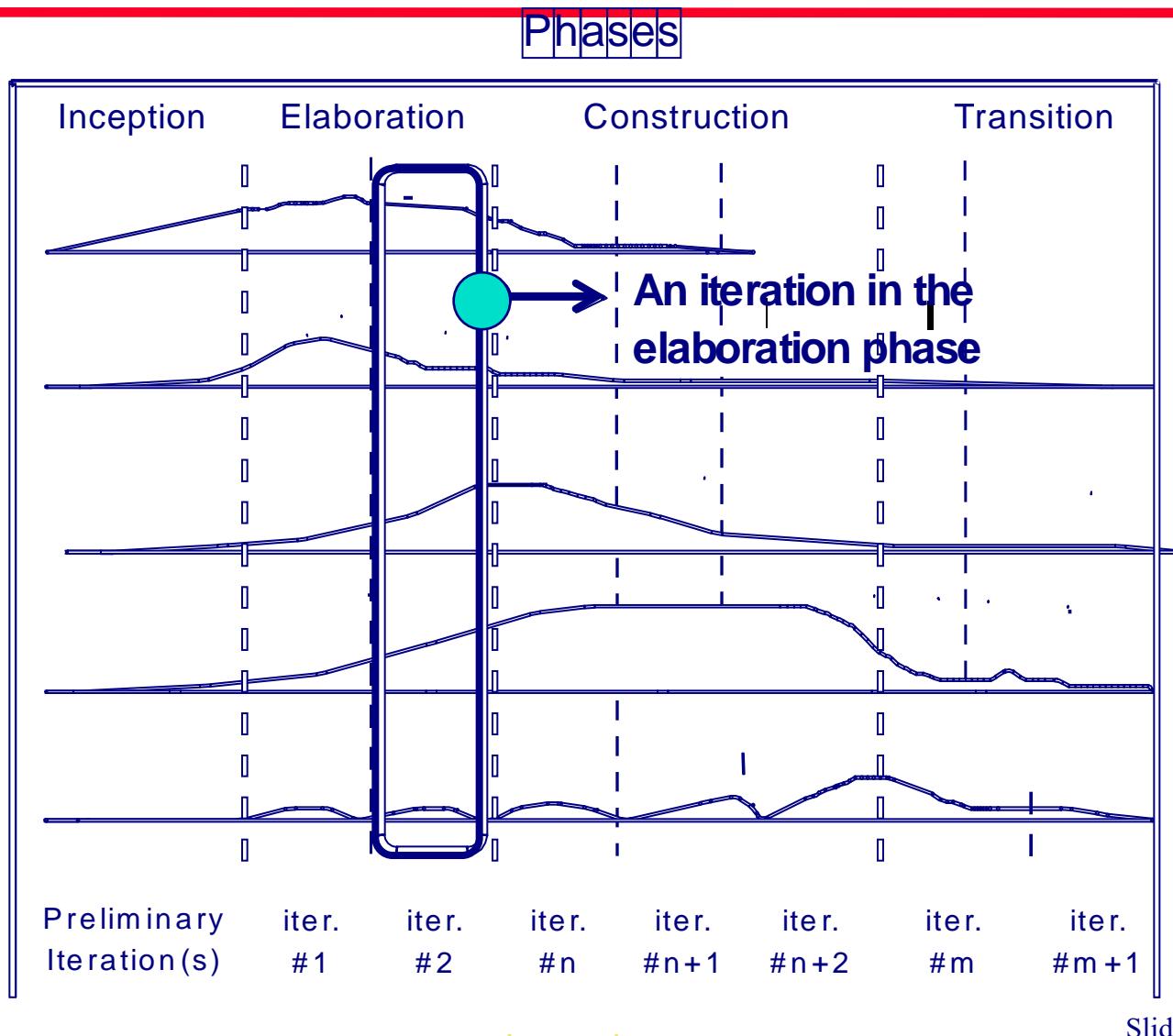
Requirements

Analysis

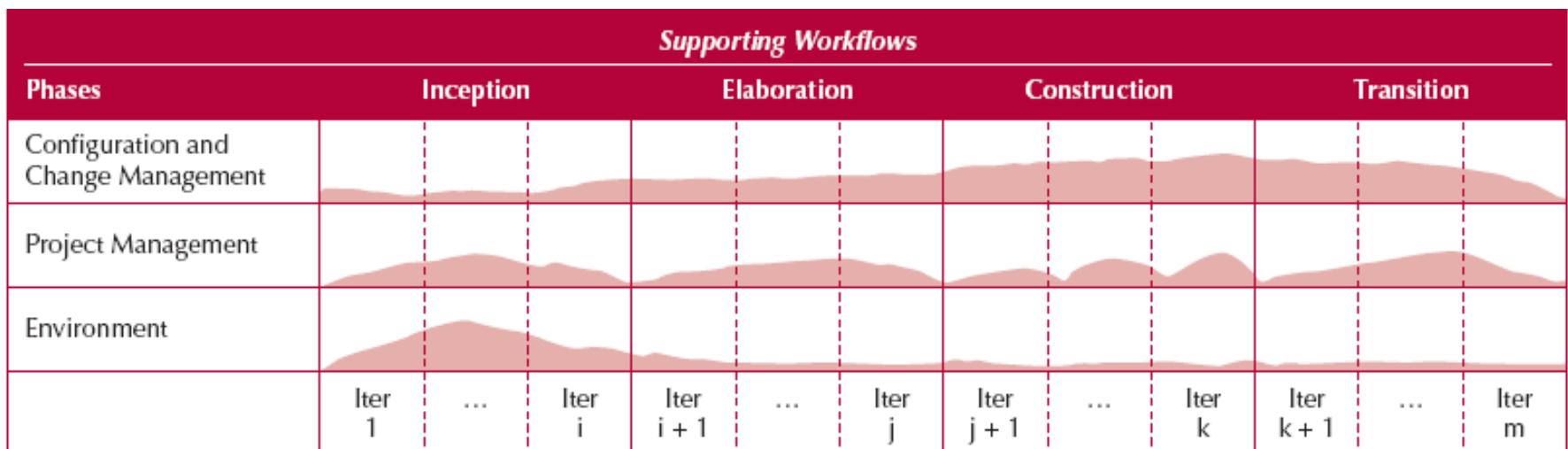
Design

Implementation

Test



# Supporting Workflows of The Unified Process



# Software Project Management Plan

---

Once the client has signed off the specifications, detailed planning and estimating begins

We draw up the software project management plan, including

- Cost estimate
- Duration estimate
- Deliverables
- Milestones
- Budget

This is the earliest possible time for the SPMP

# Post delivery Maintenance

---

Post delivery maintenance is an essential component of software development

More money is spent on post delivery maintenance than on all other activities combined

Problems can be caused by

Lack of documentation of all kinds

Two types of testing are needed

Testing the changes made during post delivery maintenance

Regression testing

All previous test cases (and their expected outcomes) need to be retained

# Retirement

---

Software can be made unmaintainable because

- A drastic change in design has occurred

- The product must be implemented on a totally new hardware/operating system

- Documentation is missing or inaccurate

- Hardware is to be changed—it may be cheaper to rewrite the software from scratch than to modify it

These are instances of maintenance (rewriting of existing software)

True retirement is a rare event

It occurs when the client organization no longer needs the functionality provided by the product

# What to Read...

---

- Dean Leffingwell, Don Widrig, Managing Software Requirements, Addison-Wesley, 2000, 491p.
- Alistair Cockburn, Writing Effective Use Cases, Addison-Wesley, 2001, 270p.
- Alan W. Brown (ed.), Component-Based Software Engineering, IEEE Computer Society, Los Alamitos, CA, 1996, pp.140.
- Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Övergaard, Object-Oriented Software Engineering-A Use Case Driven Approach, Wokingham, England, Addison-Wesley, 1992, 582p.

# Recommended Reading

---

Applying UML and Patterns: An Introduction to OOA/D and the Unified Process, Prentice Hall, 2002, by G. Larman

The Rational Unified Process - An Introduction, Addison-Wesley Professional, 2002, by its lead architect Ph. Kruchten

# **Chapter 22 – Project Management**

# Topics covered

---

- ✧ Risk management
- ✧ Managing people
- ✧ Teamwork

# Software project management

---

- ✧ Concerned with activities involved in ensuring that software is delivered on time and on schedule and in accordance with the requirements of the organisations developing and procuring the software.
- ✧ Project management is needed because software development is always subject to budget and schedule constraints that are set by the organisation developing the software.

## Success criteria

---

- ✧ Deliver the software to the customer at the agreed time.
- ✧ Keep overall costs within budget.
- ✧ Deliver software that meets the customer's expectations.
- ✧ Maintain a coherent and well-functioning development team.

# Software management distinctions

---

- ✧ The product is intangible.
  - Software cannot be seen or touched. Software project managers cannot see progress by simply looking at the artefact that is being constructed.
- ✧ Many software projects are 'one-off' projects.
  - Large software projects are usually different in some ways from previous projects. Even managers who have lots of previous experience may find it difficult to anticipate problems.
- ✧ Software processes are variable and organization specific.
  - We still cannot reliably predict when a particular software process is likely to lead to development problems.

# Factors influencing project management

---

- ✧ Company size
- ✧ Software customers
- ✧ Software size
- ✧ Software type
- ✧ Organizational culture
- ✧ Software development processes
- ✧ These factors mean that project managers in different organizations may work in quite different ways.

# Universal management activities

---

## ✧ *Project planning*

- Project managers are responsible for planning, estimating and scheduling project development and assigning people to tasks.
- Covered in Chapter 23.

## ✧ *Risk management*

- Project managers assess the risks that may affect a project, monitor these risks and take action when problems arise.

## ✧ *People management*

- Project managers have to choose people for their team and establish ways of working that leads to effective team performance.

# Management activities

---

## ✧ *Reporting*

- Project managers are usually responsible for reporting on the progress of a project to customers and to the managers of the company developing the software.

## ✧ *Proposal writing*

- The first stage in a software project may involve writing a proposal to win a contract to carry out an item of work. The proposal describes the objectives of the project and how it will be carried out.

# Risk management

# Risk management

---

- ✧ Risk management is concerned with identifying risks and drawing up plans to minimise their effect on a project.
- ✧ Software risk management is important because of the inherent uncertainties in software development.
  - These uncertainties stem from loosely defined requirements, requirements changes due to changes in customer needs, difficulties in estimating the time and resources required for software development, and differences in individual skills.
- ✧ You have to anticipate risks, understand the impact of these risks on the project, the product and the business, and take steps to avoid these risks.

# Risk classification

---

- ✧ There are two dimensions of risk classification
  - The type of risk (technical, organizational, ..)
  - what is affected by the risk:
- ✧ *Project risks* affect schedule or resources;
- ✧ *Product risks* affect the quality or performance of the software being developed;
- ✧ *Business risks* affect the organisation developing or procuring the software.

# Examples of project, product, and business risks

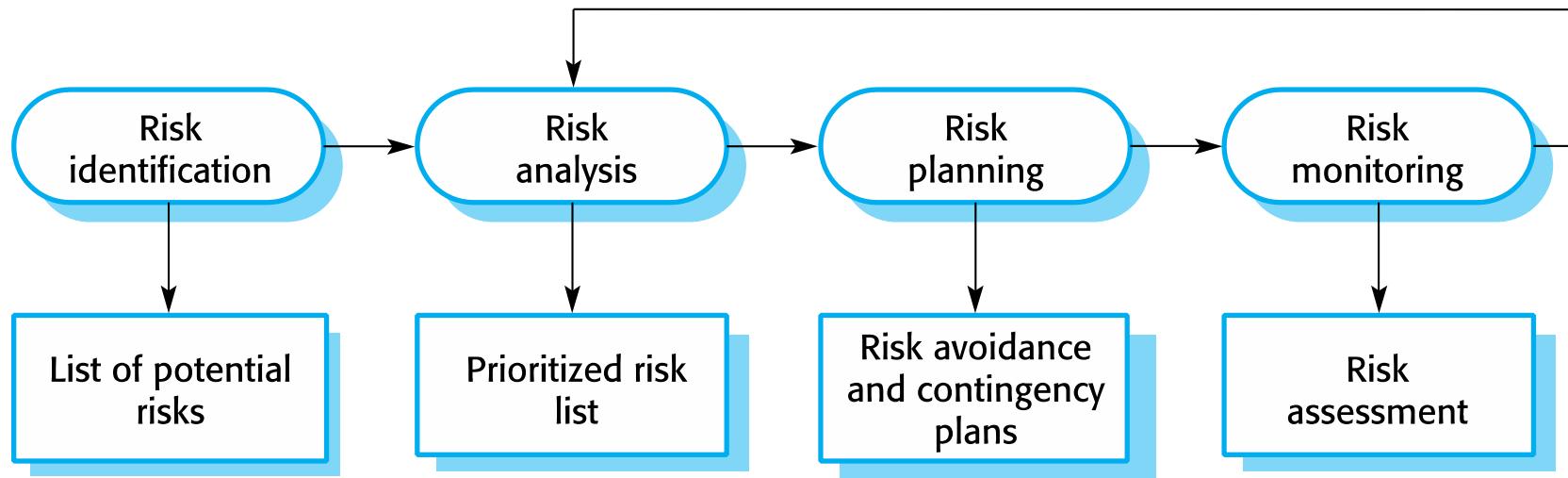
Risk	Affects	Description
Staff turnover	Project	Experienced staff will leave the project before it is finished.
Management change	Project	There will be a change of organizational management with different priorities.
Hardware unavailability	Project	Hardware that is essential for the project will not be delivered on schedule.
Requirements change	Project and product	There will be a larger number of changes to the requirements than anticipated.
Specification delays	Project and product	Specifications of essential interfaces are not available on schedule.
Size underestimate	Project and product	The size of the system has been underestimated.
CASE tool underperformance	Product	CASE tools, which support the project, do not perform as anticipated.
Technology change	Business	The underlying technology on which the system is built is superseded by new technology.
Product competition	Business	A competitive product is marketed before the system is completed.

# The risk management process

---

- ✧ Risk identification
  - Identify project, product and business risks;
- ✧ Risk analysis
  - Assess the likelihood and consequences of these risks;
- ✧ Risk planning
  - Draw up plans to avoid or minimise the effects of the risk;
- ✧ Risk monitoring
  - Monitor the risks throughout the project;

# The risk management process



# Risk identification

---

- ✧ May be a team activities or based on the individual project manager's experience.
- ✧ A checklist of common risks may be used to identify risks in a project
  - Technology risks.
  - Organizational risks.
  - People risks.
  - Requirements risks.
  - Estimation risks.

# Examples of different risk types

Risk type	Possible risks
Estimation	The time required to develop the software is underestimated. (12) The rate of defect repair is underestimated. (13) The size of the software is underestimated. (14)
Organizational	The organization is restructured so that different management are responsible for the project. (6) Organizational financial problems force reductions in the project budget. (7)
People	It is impossible to recruit staff with the skills required. (3) Key staff are ill and unavailable at critical times. (4) Required training for staff is not available. (5)
Requirements	Changes to requirements that require major design rework are proposed. (10) Customers fail to understand the impact of requirements changes. (11)
Technology	The database used in the system cannot process as many transactions per second as expected. (1) Reusable software components contain defects that mean they cannot be reused as planned. (2)
Tools	The code generated by software code generation tools is inefficient. (8) Software tools cannot work together in an integrated way. (9)

# Risk analysis

---

- ✧ Assess probability and seriousness of each risk.
- ✧ Probability may be very low, low, moderate, high or very high.
- ✧ Risk consequences might be catastrophic, serious, tolerable or insignificant.

# Risk types and examples

---

Risk	Probability	Effects
Organizational financial problems force reductions in the project budget (7).	Low	Catastrophic
It is impossible to recruit staff with the skills required for the project (3).	High	Catastrophic
Key staff are ill at critical times in the project (4).	Moderate	Serious
Faults in reusable software components have to be repaired before these components are reused. (2).	Moderate	Serious
Changes to requirements that require major design rework are proposed (10).	Moderate	Serious
The organization is restructured so that different management are responsible for the project (6).	High	Serious
The database used in the system cannot process as many transactions per second as expected (1).	Moderate	Serious

# Risk types and examples

Risk	Probability	Effects
The time required to develop the software is underestimated (12).	High	Serious
Software tools cannot be integrated (9).	High	Tolerable
Customers fail to understand the impact of requirements changes (11).	Moderate	Tolerable
Required training for staff is not available (5).	Moderate	Tolerable
The rate of defect repair is underestimated (13).	Moderate	Tolerable
The size of the software is underestimated (14).	High	Tolerable
Code generated by code generation tools is inefficient (8).	Moderate	Insignificant

# Risk planning

---

- ✧ Consider each risk and develop a strategy to manage that risk.
- ✧ Avoidance strategies
  - The probability that the risk will arise is reduced;
- ✧ Minimization strategies
  - The impact of the risk on the project or product will be reduced;
- ✧ Contingency plans
  - If the risk arises, contingency plans are plans to deal with that risk;

## What-if questions

---

- ✧ What if several engineers are ill at the same time?
- ✧ What if an economic downturn leads to budget cuts of 20% for the project?
- ✧ What if the performance of open-source software is inadequate and the only expert on that open source software leaves?
- ✧ What if the company that supplies and maintains software components goes out of business?
- ✧ What if the customer fails to deliver the revised requirements as predicted?

# Strategies to help manage risk

---

Risk	Strategy
Organizational financial problems	Prepare a briefing document for senior management showing how the project is making a very important contribution to the goals of the business and presenting reasons why cuts to the project budget would not be cost-effective.
Recruitment problems	Alert customer to potential difficulties and the possibility of delays; investigate buying-in components.
Staff illness	Reorganize team so that there is more overlap of work and people therefore understand each other's jobs.
Defective components	Replace potentially defective components with bought-in components of known reliability.
Requirements changes	Derive traceability information to assess requirements change impact; maximize information hiding in the design.

# Strategies to help manage risk

---

Risk	Strategy
Organizational restructuring	Prepare a briefing document for senior management showing how the project is making a very important contribution to the goals of the business.
Database performance	Investigate the possibility of buying a higher-performance database.
Underestimated development time	Investigate buying-in components; investigate use of a program generator.

# Risk monitoring

---

- ✧ Assess each identified risks regularly to decide whether or not it is becoming less or more probable.
- ✧ Also assess whether the effects of the risk have changed.
- ✧ Each key risk should be discussed at management progress meetings.

# Risk indicators

---

Risk type	Potential indicators
Estimation	Failure to meet agreed schedule; failure to clear reported defects.
Organizational	Organizational gossip; lack of action by senior management.
People	Poor staff morale; poor relationships amongst team members; high staff turnover.
Requirements	Many requirements change requests; customer complaints.
Technology	Late delivery of hardware or support software; many reported technology problems.
Tools	Reluctance by team members to use tools; complaints about CASE tools; demands for higher-powered workstations.

---

# Managing people

# Managing people

---

- ✧ People are an organisation's most important assets.
- ✧ The tasks of a manager are essentially people-oriented.  
Unless there is some understanding of people,  
management will be unsuccessful.
- ✧ Poor people management is an important contributor to  
project failure.

# People management factors

---

## ✧ Consistency

- Team members should all be treated in a comparable way without favourites or discrimination.

## ✧ Respect

- Different team members have different skills and these differences should be respected.

## ✧ Inclusion

- Involve all team members and make sure that people's views are considered.

## ✧ Honesty

- You should always be honest about what is going well and what is going badly in a project.

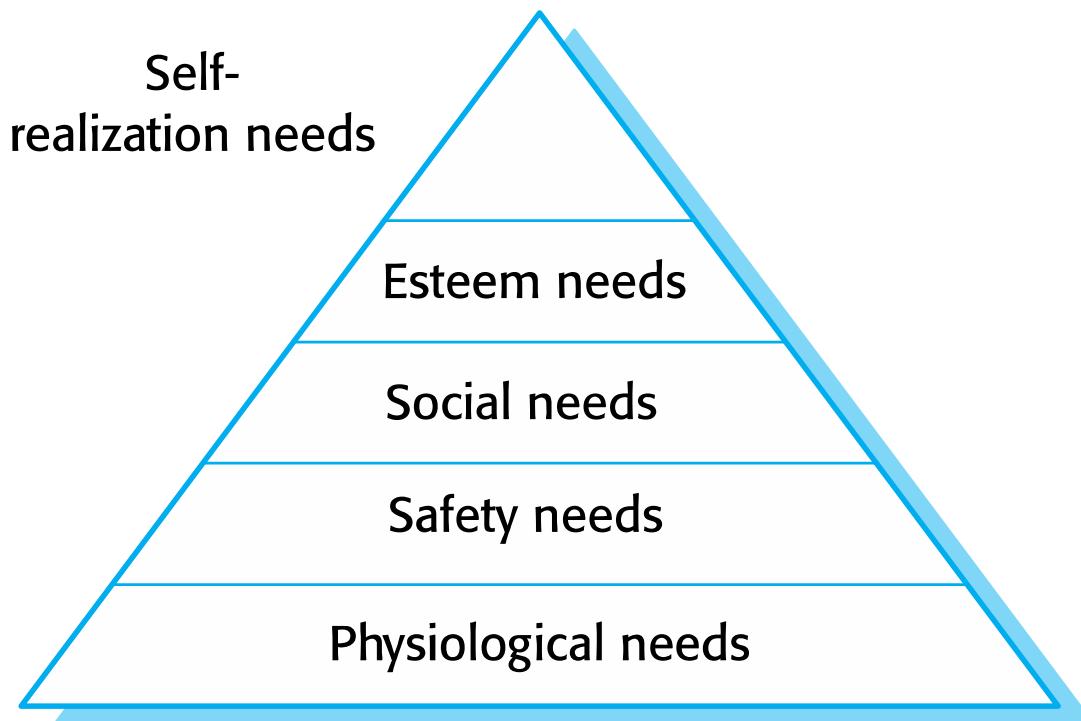
# Motivating people

---

- ✧ An important role of a manager is to motivate the people working on a project.
- ✧ Motivation means organizing the work and the working environment to encourage people to work effectively.
  - If people are not motivated, they will not be interested in the work they are doing. They will work slowly, be more likely to make mistakes and will not contribute to the broader goals of the team or the organization.
- ✧ Motivation is a complex issue but it appears that there are different types of motivation based on:
  - Basic needs (e.g. food, sleep, etc.);
  - Personal needs (e.g. respect, self-esteem);
  - Social needs (e.g. to be accepted as part of a group).

# Human needs hierarchy

---



# Need satisfaction

---

- ✧ In software development groups, basic physiological and safety needs are not an issue.
- ✧ Social
  - Provide communal facilities;
  - Allow informal communications e.g. via social networking
- ✧ Esteem
  - Recognition of achievements;
  - Appropriate rewards.
- ✧ Self-realization
  - Training - people want to learn more;
  - Responsibility.

# Case study: Individual motivation

---

Alice is a software project manager working in a company that develops alarm systems. This company wishes to enter the growing market of assistive technology to help elderly and disabled people live independently. Alice has been asked to lead a team of 6 developers than can develop new products based around the company's alarm technology.

Alice's assistive technology project starts well. Good working relationships develop within the team and creative new ideas are developed. The team decides to develop a peer-to-peer messaging system using digital televisions linked to the alarm network for communications. However, some months into the project, Alice notices that Dorothy, a hardware design expert, starts coming into work late, the quality of her work deteriorates and, increasingly, that she does not appear to be communicating with other members of the team.

Alice talks about the problem informally with other team members to try to find out if Dorothy's personal circumstances have changed, and if this might be affecting her work. They don't know of anything, so Alice decides to talk with Dorothy to try to understand the problem.

# Case study: Individual motivation

---

After some initial denials that there is a problem, Dorothy admits that she has lost interest in the job. She expected that she would be able to develop and use her hardware interfacing skills. However, because of the product direction that has been chosen, she has little opportunity for this. Basically, she is working as a C programmer with other team members.

Although she admits that the work is challenging, she is concerned that she is not developing her interfacing skills. She is worried that finding a job that involves hardware interfacing will be difficult after this project. Because she does not want to upset the team by revealing that she is thinking about the next project, she has decided that it is best to minimize conversation with them.

## Comments on case study

---

- ✧ If you don't sort out the problem of unacceptable work, the other group members will become dissatisfied and feel that they are doing an unfair share of the work.
- ✧ Personal difficulties affect motivation because people can't concentrate on their work. They need time and support to resolve these issues, although you have to make clear that they still have a responsibility to their employer.
- ✧ Alice gives Dorothy more design autonomy and organizes training courses in software engineering that will give her more opportunities after her current project has finished.

# Personality types

---

- ✧ The needs hierarchy is almost certainly an oversimplification of motivation in practice.
- ✧ Motivation should also take into account different personality types:
  - Task-oriented people, who are motivated by the work they do. In software engineering.
  - Interaction-oriented people, who are motivated by the presence and actions of co-workers.
  - Self-oriented people, who are principally motivated by personal success and recognition.

# Personality types

---

- ✧ Task-oriented.
  - The motivation for doing the work is the work itself;
- ✧ Self-oriented.
  - The work is a means to an end which is the achievement of individual goals - e.g. to get rich, to play tennis, to travel etc.;
- ✧ Interaction-oriented
  - The principal motivation is the presence and actions of co-workers. People go to work because they like to go to work.

# Motivation balance

---

- ✧ Individual motivations are made up of elements of each class.
- ✧ The balance can change depending on personal circumstances and external events.
- ✧ However, people are not just motivated by personal factors but also by being part of a group and culture.
- ✧ People go to work because they are motivated by the people that they work with.

# Teamwork

# Teamwork

---

- ✧ Most software engineering is a group activity
  - The development schedule for most non-trivial software projects is such that they cannot be completed by one person working alone.
- ✧ A good group is cohesive and has a team spirit. The people involved are motivated by the success of the group as well as by their own personal goals.
- ✧ Group interaction is a key determinant of group performance.
- ✧ Flexibility in group composition is limited
  - Managers must do the best they can with available people.

# Group cohesiveness

---

- ✧ In a cohesive group, members consider the group to be more important than any individual in it.
- ✧ The advantages of a cohesive group are:
  - Group quality standards can be developed by the group members.
  - Team members learn from each other and get to know each other's work; Inhibitions caused by ignorance are reduced.
  - Knowledge is shared. Continuity can be maintained if a group member leaves.
  - Refactoring and continual improvement is encouraged. Group members work collectively to deliver high quality results and fix problems, irrespective of the individuals who originally created the design or program.

# Team spirit

---

Alice, an experienced project manager, understands the importance of creating a cohesive group. As they are developing a new product, she takes the opportunity of involving all group members in the product specification and design by getting them to discuss possible technology with elderly members of their families. She also encourages them to bring these family members to meet other members of the development group.

Alice also arranges monthly lunches for everyone in the group. These lunches are an opportunity for all team members to meet informally, talk around issues of concern, and get to know each other. At the lunch, Alice tells the group what she knows about organizational news, policies, strategies, and so forth. Each team member then briefly summarizes what they have been doing and the group discusses a general topic, such as new product ideas from elderly relatives.

Every few months, Alice organizes an ‘away day’ for the group where the team spends two days on ‘technology updating’. Each team member prepares an update on a relevant technology and presents it to the group. This is an off-site meeting in a good hotel and plenty of time is scheduled for discussion and social interaction.

# The effectiveness of a team

---

## ✧ The people in the group

- You need a mix of people in a project group as software development involves diverse activities such as negotiating with clients, programming, testing and documentation.

## ✧ The group organization

- A group should be organized so that individuals can contribute to the best of their abilities and tasks can be completed as expected.

## ✧ Technical and managerial communications

- Good communications between group members, and between the software engineering team and other project stakeholders, is essential.

# Selecting group members

---

- ✧ A manager or team leader's job is to create a cohesive group and organize their group so that they can work together effectively.
- ✧ This involves creating a group with the right balance of technical skills and personalities, and organizing that group so that the members work together effectively.

# Assembling a team

---

- ✧ May not be possible to appoint the ideal people to work on a project
  - Project budget may not allow for the use of highly-paid staff;
  - Staff with the appropriate experience may not be available;
  - An organisation may wish to develop employee skills on a software project.
- ✧ Managers have to work within these constraints especially when there are shortages of trained staff.

# Group composition

---

- ✧ Group composed of members who share the same motivation can be problematic
  - Task-oriented - everyone wants to do their own thing;
  - Self-oriented - everyone wants to be the boss;
  - Interaction-oriented - too much chatting, not enough work.
- ✧ An effective group has a balance of all types.
- ✧ This can be difficult to achieve software engineers are often task-oriented.
- ✧ Interaction-oriented people are very important as they can detect and defuse tensions that arise.

# Group composition

---

In creating a group for assistive technology development, Alice is aware of the importance of selecting members with complementary personalities. When interviewing potential group members, she tried to assess whether they were task-oriented, self-oriented, or interaction-oriented. She felt that she was primarily a self-oriented type because she considered the project to be a way of getting noticed by senior management and possibly promoted. She therefore looked for one or perhaps two interaction-oriented personalities, with task-oriented individuals to complete the team. The final assessment that she arrived at was:

- Alice—self-oriented
- Brian—task-oriented
- Bob—task-oriented
- Carol—interaction-oriented
- Dorothy—self-oriented
- Ed—interaction-oriented
- Fred—task-oriented

# Group organization

---

- ✧ The way that a group is organized affects the decisions that are made by that group, the ways that information is exchanged and the interactions between the development group and external project stakeholders.
  - Key questions include:
    - Should the project manager be the technical leader of the group?
    - Who will be involved in making critical technical decisions, and how will these be made?
    - How will interactions with external stakeholders and senior company management be handled?
    - How can groups integrate people who are not co-located?
    - How can knowledge be shared across the group?

# Group organization

---

- ✧ Small software engineering groups are usually organised informally without a rigid structure.
- ✧ For large projects, there may be a hierarchical structure where different groups are responsible for different sub-projects.
- ✧ Agile development is always based around an informal group on the principle that formal structure inhibits information exchange

## Informal groups

---

- ✧ The group acts as a whole and comes to a consensus on decisions affecting the system.
- ✧ The group leader serves as the external interface of the group but does not allocate specific work items.
- ✧ Rather, work is discussed by the group as a whole and tasks are allocated according to ability and experience.
- ✧ This approach is successful for groups where all members are experienced and competent.

# Group communications

---

- ✧ Good communications are essential for effective group working.
- ✧ Information must be exchanged on the status of work, design decisions and changes to previous decisions.
- ✧ Good communications also strengthens group cohesion as it promotes understanding.

# Group communications

---

## ✧ Group size

- The larger the group, the harder it is for people to communicate with other group members.

## ✧ Group structure

- Communication is better in informally structured groups than in hierarchically structured groups.

## ✧ Group composition

- Communication is better when there are different personality types in a group and when groups are mixed rather than single sex.

## ✧ The physical work environment

- Good workplace organisation can help encourage communications.

# Key points

---

- ✧ Good project management is essential if software engineering projects are to be developed on schedule and within budget.
- ✧ Software management is distinct from other engineering management. Software is intangible. Projects may be novel or innovative with no body of experience to guide their management. Software processes are not as mature as traditional engineering processes.
- ✧ Risk management involves identifying and assessing project risks to establish the probability that they will occur and the consequences for the project if that risk does arise. You should make plans to avoid, manage or deal with likely risks if or when they arise.

# Key points

---

- ✧ People management involves choosing the right people to work on a project and organizing the team and its working environment.
- ✧ People are motivated by interaction with other people, the recognition of management and their peers, and by being given opportunities for personal development.
- ✧ Software development groups should be fairly small and cohesive. The key factors that influence the effectiveness of a group are the people in that group, the way that it is organized and the communication between group members.
- ✧ Communications within a group are influenced by factors such as the status of group members, the size of the group, the gender composition of the group, personalities and available communication channels.

# Chapter 31

---

## ■ Project Management Concepts

*Slide Set to accompany*

*Software Engineering: A Practitioner's Approach, 8/e*  
**by Roger S. Pressman and Bruce R. Maxim**

Slides copyright © 1996, 2001, 2005, 2009, 2014 by Roger S. Pressman

***For non-profit educational use only***

May be reproduced ONLY for student use at the university level when used in conjunction with *Software Engineering: A Practitioner's Approach, 8/e*. Any other reproduction or use is prohibited without the express written permission of the author.

All copyright information MUST appear if these slides are posted on a website for student use.

# The Four P's

---

- **People** — the most important element of a successful project
- **Product** — the software to be built
- **Process** — the set of framework activities and software engineering tasks to get the job done
- **Project** — all work required to make the product a reality

# Stakeholders

---

- *Senior managers* who define the business issues that often have significant influence on the project.
- *Project (technical) managers* who must plan, motivate, organize, and control the practitioners who do software work.
- *Practitioners* who deliver the technical skills that are necessary to engineer a product or application.
- *Customers* who specify the requirements for the software to be engineered and other stakeholders who have a peripheral interest in the outcome.
- *End-users* who interact with the software once it is released for production use.

# Software Teams

---



# Team Leader

---

## ■ The MOI Model

- **Motivation.** The ability to encourage (by “push or pull”) technical people to produce to their best ability.
- **Organization.** The ability to mold existing processes (or invent new ones) that will enable the initial concept to be translated into a final product.
- **Ideas or innovation.** The ability to encourage people to create and feel creative even when they must work within bounds established for a particular software product or application.

# Software Teams

---

***The following factors must be considered when selecting a software project team structure ...***

- the difficulty of the problem to be solved
- the size of the resultant program(s) in lines of code or function points
- the time that the team will stay together (team lifetime)
- the degree to which the problem can be modularized
- the required quality and reliability of the system to be built
- the rigidity of the delivery date
- the degree of sociability (communication) required for the project

# Organizational Paradigms

---

- **closed paradigm**—structures a team along a traditional hierarchy of authority
- **random paradigm**—structures a team loosely and depends on individual initiative of the team members
- **open paradigm**—attempts to structure a team in a manner that achieves some of the controls associated with the closed paradigm but also much of the innovation that occurs when using the random paradigm
- **synchronous paradigm**—relies on the natural compartmentalization of a problem and organizes team members to work on pieces of the problem with little active communication among themselves

*suggested by Constantine [Con93]*

# Avoid Team “Toxicity”

---

- A frenzied work atmosphere in which team members waste energy and lose focus on the objectives of the work to be performed.
- High frustration caused by personal, business, or technological factors that cause friction among team members.
- “Fragmented or poorly coordinated procedures” or a poorly defined or improperly chosen process model that becomes a roadblock to accomplishment.
- Unclear definition of roles resulting in a lack of accountability and resultant finger-pointing.
- “Continuous and repeated exposure to failure” that leads to a loss of confidence and a lowering of morale.

# Agile Teams

---

- Team members must have trust in one another.
- The distribution of skills must be appropriate to the problem.
- Mavericks may have to be excluded from the team, if team cohesiveness is to be maintained.
- Team is “self-organizing”
  - An adaptive team structure
  - Uses elements of Constantine’s random, open, and synchronous paradigms
  - Significant autonomy

# Team Coordination & Communication

---

- *Formal, impersonal approaches* include software engineering documents and work products (including source code), technical memos, project milestones, schedules, and project control tools (Chapter 23), change requests and related documentation, error tracking reports, and repository data (see Chapter 26).
- *Formal, interpersonal procedures* focus on quality assurance activities (Chapter 25) applied to software engineering work products. These include status review meetings and design and code inspections.
- *Informal, interpersonal procedures* include group meetings for information dissemination and problem solving and “collocation of requirements and development staff.”
- *Electronic communication* encompasses electronic mail, electronic bulletin boards, and by extension, video-based conferencing systems.
- *Interpersonal networking* includes informal discussions with team members and those outside the project who may have experience or insight that can assist team members.

# The Product Scope

---

## ■ Scope

- **Context.** How does the software to be built fit into a larger system, product, or business context and what constraints are imposed as a result of the context?
- **Information objectives.** What customer-visible data objects (Chapter 8) are produced as output from the software? What data objects are required for input?
- **Function and performance.** What function does the software perform to transform input data into output? Are any special performance characteristics to be addressed?

## ■ Software project scope must be unambiguous and understandable at the management and technical levels.

# Problem Decomposition

---

- Sometimes called *partitioning* or *problem elaboration*
- Once scope is defined ...
  - It is decomposed into constituent functions
  - It is decomposed into user-visible data objects  
*or*
  - It is decomposed into a set of problem classes
- Decomposition process continues until all functions or problem classes have been defined

# The Process

---

- Once a process framework has been established
  - Consider project characteristics
  - Determine the degree of rigor required
  - Define a task set for each software engineering activity
    - Task set =
      - Software engineering tasks
      - Work products
      - Quality assurance points
      - Milestones

# Melding the Problem and the Process

COMMON PROCESS FRAMEWORK ACTIVITIES	Planning	Design	Implementation	Testing	Deployment	Evolution
Software Engineering Tools						
Product Functions						
Text Input						
Editing and formatting						
Automatic copy edit						
Page layout capability						
Automatic indexing and TOC						
File management						
Document production						

# The Project

---

- *Projects get into trouble when ...*
  - Software people don't understand their customer's needs.
  - The product scope is poorly defined.
  - Changes are managed poorly.
  - The chosen technology changes.
  - Business needs change [or are ill-defined].
  - Deadlines are unrealistic.
  - Users are resistant.
  - Sponsorship is lost [or was never properly obtained].
  - The project team lacks people with appropriate skills.
  - Managers [and practitioners] avoid best practices and lessons learned.

# Common-Sense Approach to Projects

---

- *Start on the right foot.* This is accomplished by working hard (very hard) to understand the problem that is to be solved and then setting realistic objectives and expectations.
- *Maintain momentum.* The project manager must provide incentives to keep turnover of personnel to an absolute minimum, the team should emphasize quality in every task it performs, and senior management should do everything possible to stay out of the team's way.
- *Track progress.* For a software project, progress is tracked as work products (e.g., models, source code, sets of test cases) are produced and approved (using formal technical reviews) as part of a quality assurance activity.
- *Make smart decisions.* In essence, the decisions of the project manager and the software team should be to "keep it simple."
- *Conduct a postmortem analysis.* Establish a consistent mechanism for extracting lessons learned for each project.

# To Get to the Essence of a Project

---

- Why is the system being developed?
- What will be done?
- When will it be accomplished?
- Who is responsible?
- Where are they organizationally located?
- How will the job be done technically and managerially?
- How much of each resource (e.g., people, software, tools, database) will be needed?

*Barry Boehm [Boe96]*

# Critical Practices

---

- Formal risk management
- Empirical cost and schedule estimation
- Metrics-based project management
- Earned value tracking
- Defect tracking against quality targets
- People aware project management

# Chapter 23 – Project planning

# Topics covered

---

- ✧ Software pricing
- ✧ Plan-driven development
- ✧ Project scheduling
- ✧ Agile planning
- ✧ Estimation techniques
- ✧ COCOMO cost modeling

# Project planning

---

- ✧ Project planning involves breaking down the work into parts and assign these to project team members, anticipate problems that might arise and prepare tentative solutions to those problems.
- ✧ The project plan, which is created at the start of a project, is used to communicate how the work will be done to the project team and customers, and to help assess progress on the project.

# Planning stages

---

- ✧ At the proposal stage, when you are bidding for a contract to develop or provide a software system.
- ✧ During the project startup phase, when you have to plan who will work on the project, how the project will be broken down into increments, how resources will be allocated across your company, etc.
- ✧ Periodically throughout the project, when you modify your plan in the light of experience gained and information from monitoring the progress of the work.

# Proposal planning

---

- ✧ Planning may be necessary with only outline software requirements.
- ✧ The aim of planning at this stage is to provide information that will be used in setting a price for the system to customers.
- ✧ Project pricing involves estimating how much the software will cost to develop, taking factors such as staff costs, hardware costs, software costs, etc. into account

# Project startup planning

---

- ✧ At this stage, you know more about the system requirements but do not have design or implementation information
- ✧ Create a plan with enough detail to make decisions about the project budget and staffing.
  - This plan is the basis for project resource allocation
- ✧ The startup plan should also define project monitoring mechanisms
- ✧ A startup plan is still needed for agile development to allow resources to be allocated to the project

# Development planning

---

- ✧ The project plan should be regularly amended as the project progresses and you know more about the software and its development
- ✧ The project schedule, cost-estimate and risks have to be regularly revised

---

# **Software pricing**

# Software pricing

---

- ✧ Estimates are made to discover the cost, to the developer, of producing a software system.
  - You take into account, hardware, software, travel, training and effort costs.
- ✧ There is not a simple relationship between the development cost and the price charged to the customer.
- ✧ Broader organisational, economic, political and business considerations influence the price charged.

# Factors affecting software pricing

---

Factor	Description
Contractual terms	A customer may be willing to allow the developer to retain ownership of the source code and reuse it in other projects. The price charged may then be less than if the software source code is handed over to the customer.
Cost estimate uncertainty	If an organization is unsure of its cost estimate, it may increase its price by a contingency over and above its normal profit.
Financial health	Developers in financial difficulty may lower their price to gain a contract. It is better to make a smaller than normal profit or break even than to go out of business. Cash flow is more important than profit in difficult economic times.

# Factors affecting software pricing

---

Factor	Description
Market opportunity	A development organization may quote a low price because it wishes to move into a new segment of the software market. Accepting a low profit on one project may give the organization the opportunity to make a greater profit later. The experience gained may also help it develop new products.
Requirements volatility	If the requirements are likely to change, an organization may lower its price to win a contract. After the contract is awarded, high prices can be charged for changes to the requirements.

# Pricing strategies

---

## ✧ Under pricing

- A company may underprice a system in order to gain a contract that allows them to retain staff for future opportunities
- A company may underprice a system to gain access to a new market area

## ✧ Increased pricing

- The price may be increased when a buyer wishes a fixed-price contract and so the seller increases the price to allow for unexpected risks

## Pricing to win

---

- ✧ The software is priced according to what the software developer believes the buyer is willing to pay
- ✧ If this is less than the development costs, the software functionality may be reduced accordingly with a view to extra functionality being added in a later release
- ✧ Additional costs may be added as the requirements change and these may be priced at a higher level to make up the shortfall in the original price

# **Plan-driven development**

# Plan-driven development

---

- ✧ Plan-driven or plan-based development is an approach to software engineering where the development process is planned in detail.
  - Plan-driven development is based on engineering project management techniques and is the ‘traditional’ way of managing large software development projects.
- ✧ A project plan is created that records the work to be done, who will do it, the development schedule and the work products.
- ✧ Managers use the plan to support project decision making and as a way of measuring progress.

# Plan-driven development – pros and cons

---

- ✧ The arguments in favor of a plan-driven approach are that early planning allows organizational issues (availability of staff, other projects, etc.) to be closely taken into account, and that potential problems and dependencies are discovered before the project starts, rather than once the project is underway.
- ✧ The principal argument against plan-driven development is that many early decisions have to be revised because of changes to the environment in which the software is to be developed and used.

# Project plans

---

- ✧ In a plan-driven development project, a project plan sets out the resources available to the project, the work breakdown and a schedule for carrying out the work.
- ✧ Plan sections
  - Introduction
  - Project organization
  - Risk analysis
  - Hardware and software resource requirements
  - Work breakdown
  - Project schedule
  - Monitoring and reporting mechanisms

# Project plan supplements

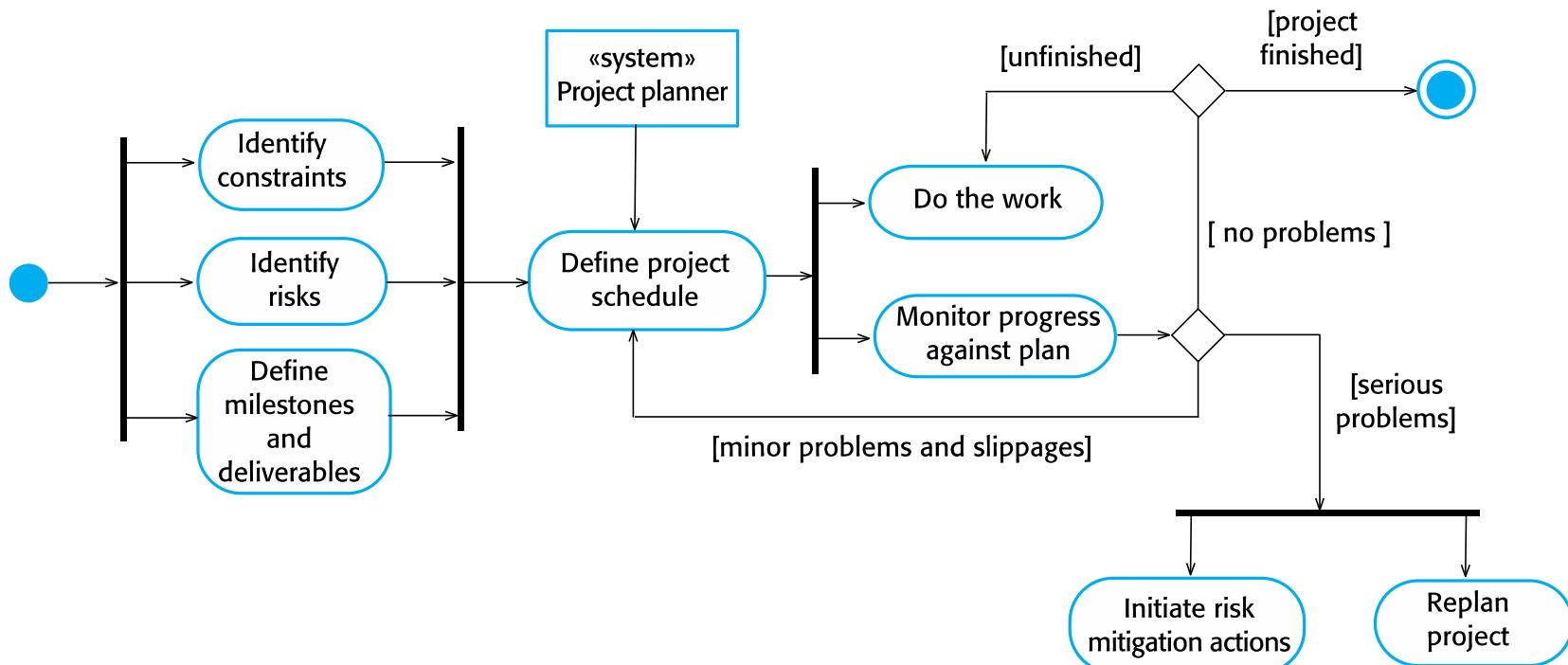
Plan	Description
Configuration management plan	Describes the configuration management procedures and structures to be used.
Deployment plan	Describes how the software and associated hardware (if required) will be deployed in the customer's environment. This should include a plan for migrating data from existing systems.
Maintenance plan	Predicts the maintenance requirements, costs, and effort.
Quality plan	Describes the quality procedures and standards that will be used in a project.
Validation plan	Describes the approach, resources, and schedule used for system validation.

# The planning process

---

- ✧ Project planning is an iterative process that starts when you create an initial project plan during the project startup phase.
- ✧ Plan changes are inevitable.
  - As more information about the system and the project team becomes available during the project, you should regularly revise the plan to reflect requirements, schedule and risk changes.
  - Changing business goals also leads to changes in project plans. As business goals change, this could affect all projects, which may then have to be re-planned.

# The project planning process



# Planning assumptions

---

- ✧ You should make realistic rather than optimistic assumptions when you are defining a project plan.
- ✧ Problems of some description always arise during a project, and these lead to project delays.
- ✧ Your initial assumptions and scheduling should therefore take unexpected problems into account.
- ✧ You should include contingency in your plan so that if things go wrong, then your delivery schedule is not seriously disrupted.

# Risk mitigation

---

- ✧ If there are serious problems with the development work that are likely to lead to significant delays, you need to initiate risk mitigation actions to reduce the risks of project failure.
- ✧ In conjunction with these actions, you also have to re-plan the project.
- ✧ This may involve renegotiating the project constraints and deliverables with the customer. A new schedule of when work should be completed also has to be established and agreed with the customer.

---

# **Project scheduling**

# Project scheduling

---

- ✧ Project scheduling is the process of deciding how the work in a project will be organized as separate tasks, and when and how these tasks will be executed.
- ✧ You estimate the calendar time needed to complete each task, the effort required and who will work on the tasks that have been identified.
- ✧ You also have to estimate the resources needed to complete each task, such as the disk space required on a server, the time required on specialized hardware, such as a simulator, and what the travel budget will be.

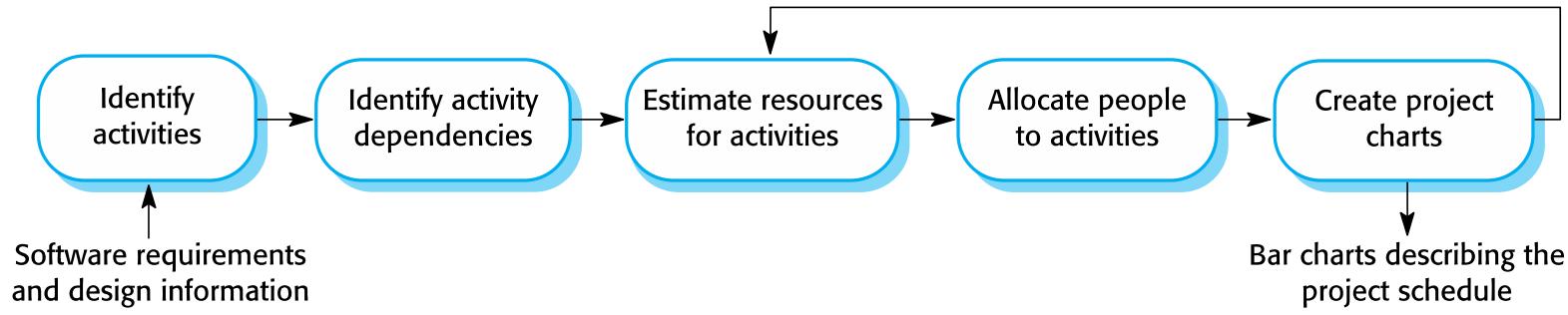
# Project scheduling activities

---

- ✧ Split project into tasks and estimate time and resources required to complete each task.
- ✧ Organize tasks concurrently to make optimal use of workforce.
- ✧ Minimize task dependencies to avoid delays caused by one task waiting for another to complete.
- ✧ Dependent on project managers intuition and experience.

# The project scheduling process

---



# Scheduling problems

---

- ✧ Estimating the difficulty of problems and hence the cost of developing a solution is hard.
- ✧ Productivity is not proportional to the number of people working on a task.
- ✧ Adding people to a late project makes it later because of communication overheads.
- ✧ The unexpected always happens. Always allow contingency in planning.

# Schedule presentation

---

- ✧ Graphical notations are normally used to illustrate the project schedule.
- ✧ These show the project breakdown into tasks. Tasks should not be too small. They should take about a week or two.
- ✧ Calendar-based
  - Bar charts are the most commonly used representation for project schedules. They show the schedule as activities or resources against time.
- ✧ Activity networks
  - Show task dependencies

# Project activities

---

❖ Project activities (tasks) are the basic planning element.  
Each activity has:

- a duration in calendar days or months,
- an effort estimate, which shows the number of person-days or person-months to complete the work,
- a deadline by which the activity should be complete,
- a defined end-point, which might be a document, the holding of a review meeting, the successful execution of all tests, etc.

# Milestones and deliverables

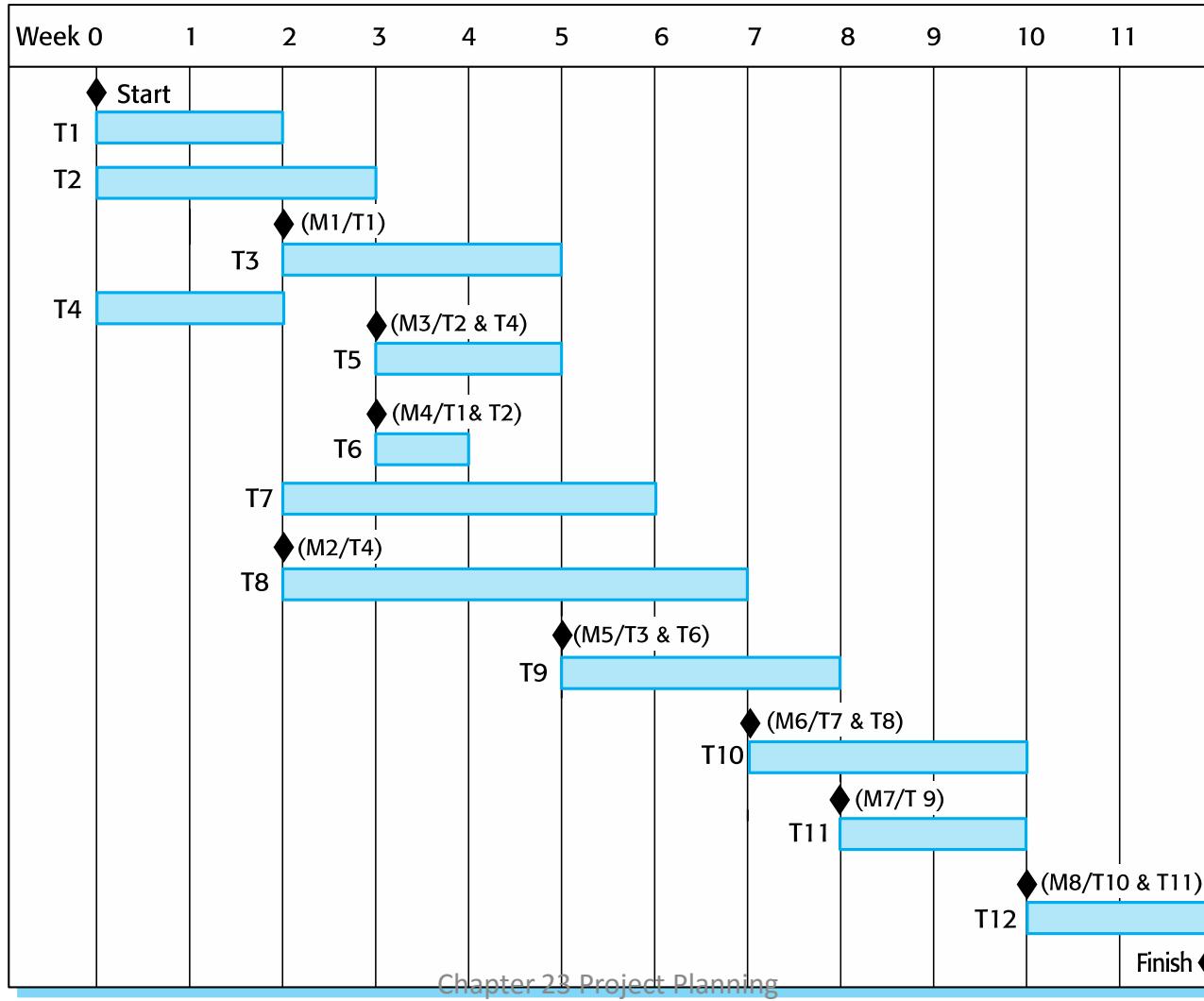
---

- ✧ Milestones are points in the schedule against which you can assess progress, for example, the handover of the system for testing.
- ✧ Deliverables are work products that are delivered to the customer, e.g. a requirements document for the system.

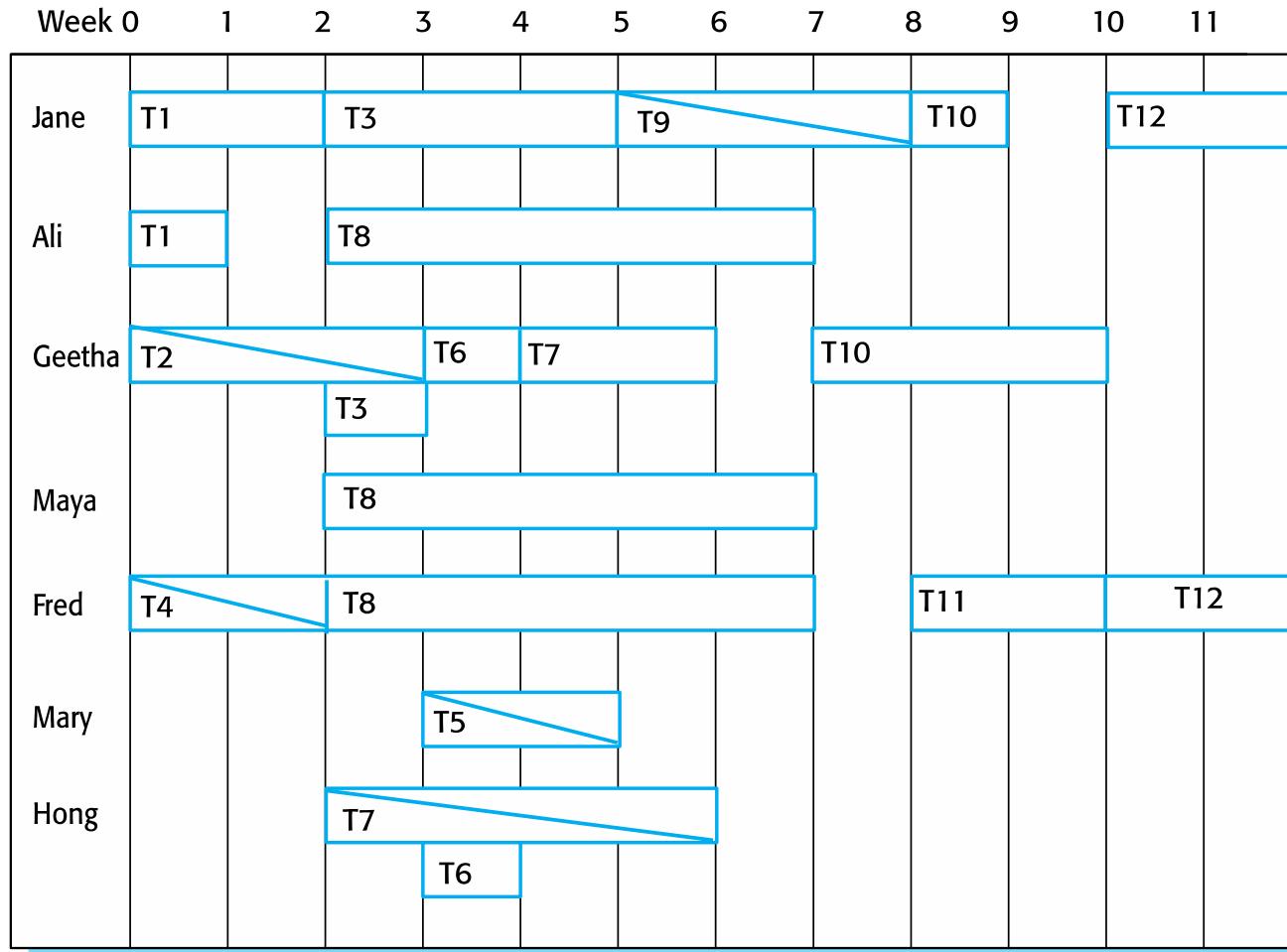
# Tasks, durations, and dependencies

Task	Effort (person-days)	Duration (days)	Dependencies
T1	15	10	
T2	8	15	
T3	20	15	T1 (M1)
T4	5	10	
T5	5	10	T2, T4 (M3)
T6	10	5	T1, T2 (M4)
T7	25	20	T1 (M1)
T8	75	25	T4 (M2)
T9	10	15	T3, T6 (M5)
T10	20	15	T7, T8 (M6)
T11	10	10	T9 (M7)
T12	20	10	T10, T11 (M8)

# Activity bar chart



# Staff allocation chart



# Agile planning

# Agile planning

---

- ✧ Agile methods of software development are iterative approaches where the software is developed and delivered to customers in increments.
- ✧ Unlike plan-driven approaches, the functionality of these increments is not planned in advance but is decided during the development.
  - The decision on what to include in an increment depends on progress and on the customer's priorities.
- ✧ The customer's priorities and requirements change so it makes sense to have a flexible plan that can accommodate these changes.

# Agile planning stages

---

- ✧ Release planning, which looks ahead for several months and decides on the features that should be included in a release of a system.
- ✧ Iteration planning, which has a shorter term outlook, and focuses on planning the next increment of a system. This is typically 2-4 weeks of work for the team.

# Approaches to agile planning

---

- ✧ Planning in Scrum
  - Covered in Chapter 3
- ✧ Based on managing a project backlog (things to be done) with daily reviews of progress and problems
- ✧ The planning game
  - Developed originally as part of Extreme Programming (XP)
  - Dependent on user stories as a measure of progress in the project

# Story-based planning

---

- ✧ The planning game is based on user stories that reflect the features that should be included in the system.
- ✧ The project team read and discuss the stories and rank them in order of the amount of time they think it will take to implement the story.
- ✧ Stories are assigned ‘effort points’ reflecting their size and difficulty of implementation
- ✧ The number of effort points implemented per day is measured giving an estimate of the team’s ‘velocity’
- ✧ This allows the total effort required to implement the system to be estimated

# The planning game

---



# Release and iteration planning

---

- ✧ Release planning involves selecting and refining the stories that will reflect the features to be implemented in a release of a system and the order in which the stories should be implemented.
- ✧ Stories to be implemented in each iteration are chosen, with the number of stories reflecting the time to deliver an iteration (usually 2 or 3 weeks).
- ✧ The team's velocity is used to guide the choice of stories so that they can be delivered within an iteration.

# Task allocation

---

- ✧ During the task planning stage, the developers break down stories into development tasks.
  - A development task should take 4–16 hours.
  - All of the tasks that must be completed to implement all of the stories in that iteration are listed.
  - The individual developers then sign up for the specific tasks that they will implement.
- ✧ Benefits of this approach:
  - The whole team gets an overview of the tasks to be completed in an iteration.
  - Developers have a sense of ownership in these tasks and this is likely to motivate them to complete the task.

# Software delivery

---

- ✧ A software increment is always delivered at the end of each project iteration.
- ✧ If the features to be included in the increment cannot be completed in the time allowed, the scope of the work is reduced.
- ✧ The delivery schedule is never extended.

# Agile planning difficulties

---

- ✧ Agile planning is reliant on customer involvement and availability.
- ✧ This can be difficult to arrange, as customer representatives sometimes have to prioritize other work and are not available for the planning game.
- ✧ Furthermore, some customers may be more familiar with traditional project plans and may find it difficult to engage in an agile planning process.

# Agile planning applicability

---

- ✧ Agile planning works well with small, stable development teams that can get together and discuss the stories to be implemented.
- ✧ However, where teams are large and/or geographically distributed, or when team membership changes frequently, it is practically impossible for everyone to be involved in the collaborative planning that is essential for agile project management.

---

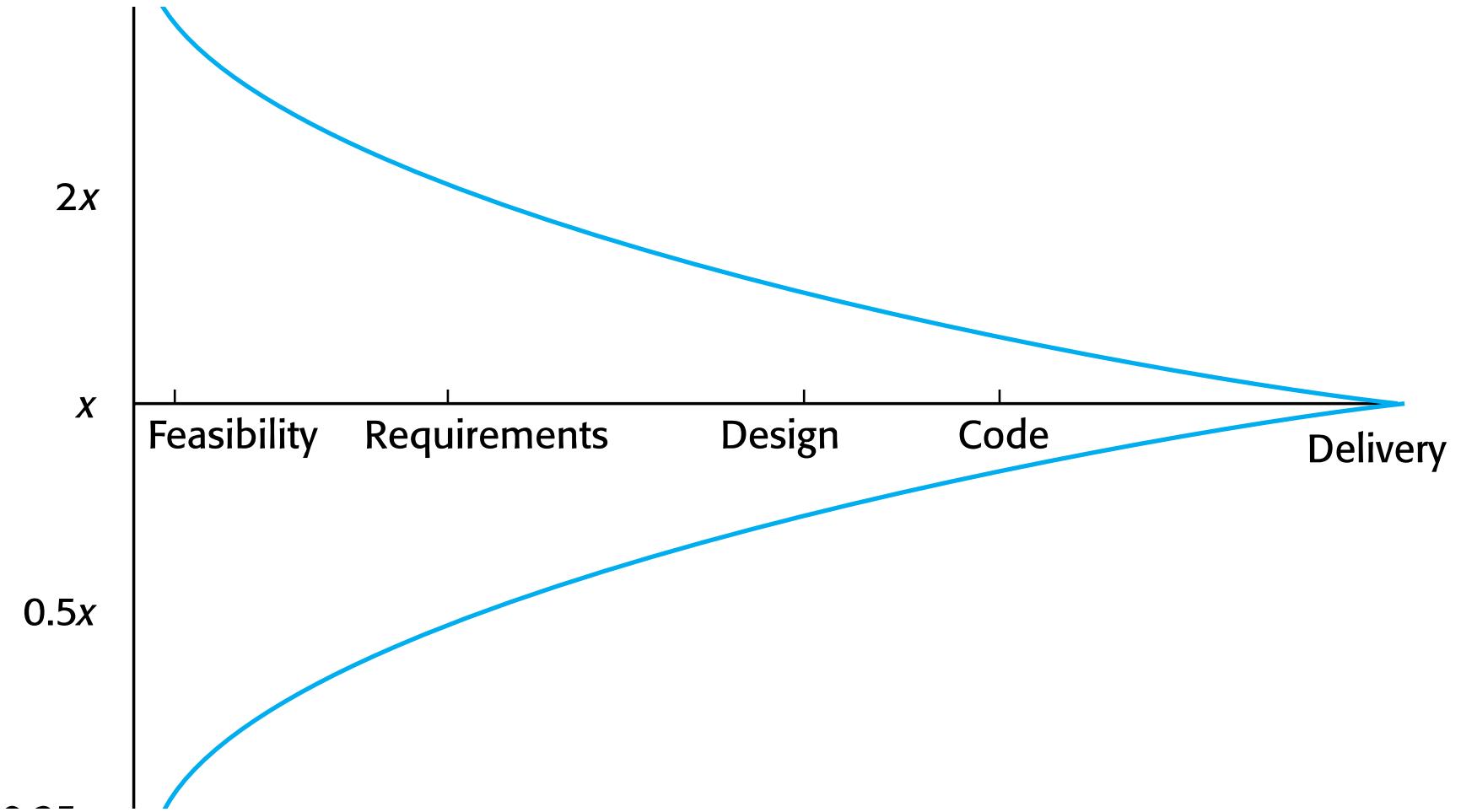
# **Estimation techniques**

# Estimation techniques

---

- ✧ Organizations need to make software effort and cost estimates. There are two types of technique that can be used to do this:
  - *Experience-based techniques* The estimate of future effort requirements is based on the manager's experience of past projects and the application domain. Essentially, the manager makes an informed judgment of what the effort requirements are likely to be.
  - *Algorithmic cost modeling* In this approach, a formulaic approach is used to compute the project effort based on estimates of product attributes, such as size, and process characteristics, such as experience of staff involved.

# Estimate uncertainty



# Experience-based approaches

---

- ✧ Experience-based techniques rely on judgments based on experience of past projects and the effort expended in these projects on software development activities.
- ✧ Typically, you identify the deliverables to be produced in a project and the different software components or systems that are to be developed.
- ✧ You document these in a spreadsheet, estimate them individually and compute the total effort required.
- ✧ It usually helps to get a group of people involved in the effort estimation and to ask each member of the group to explain their estimate.

# Problem with experience-based approaches

---

- ✧ The difficulty with experience-based techniques is that a new software project may not have much in common with previous projects.
- ✧ Software development changes very quickly and a project will often use unfamiliar techniques such as web services, application system configuration or HTML5.
- ✧ If you have not worked with these techniques, your previous experience may not help you to estimate the effort required, making it more difficult to produce accurate costs and schedule estimates.

# Algorithmic cost modelling

---

- ✧ Cost is estimated as a mathematical function of product, project and process attributes whose values are estimated by project managers:
  - $\text{Effort} = A \cdot \text{Size}^B \cdot M$
  - A is an organisation-dependent constant, B reflects the disproportionate effort for large projects and M is a multiplier reflecting product, process and people attributes.
- ✧ The most commonly used product attribute for cost estimation is code size.
- ✧ Most models are similar but they use different values for A, B and M.

# Estimation accuracy

---

- ✧ The size of a software system can only be known accurately when it is finished.
- ✧ Several factors influence the final size
  - Use of reused systems and components;
  - Programming language;
  - Distribution of system.
- ✧ As the development process progresses then the size estimate becomes more accurate.
- ✧ The estimates of the factors contributing to B and M are subjective and vary according to the judgment of the estimator.

# Effectiveness of algorithmic models

---

- ✧ Algorithmic cost models are a systematic way to estimate the effort required to develop a system. However, these models are complex and difficult to use.
- ✧ There are many attributes and considerable scope for uncertainty in estimating their values.
- ✧ This complexity means that the practical application of algorithmic cost modeling has been limited to a relatively small number of large companies, mostly working in defense and aerospace systems engineering.

# COCOMO cost modeling

# COCOMO cost modeling

---

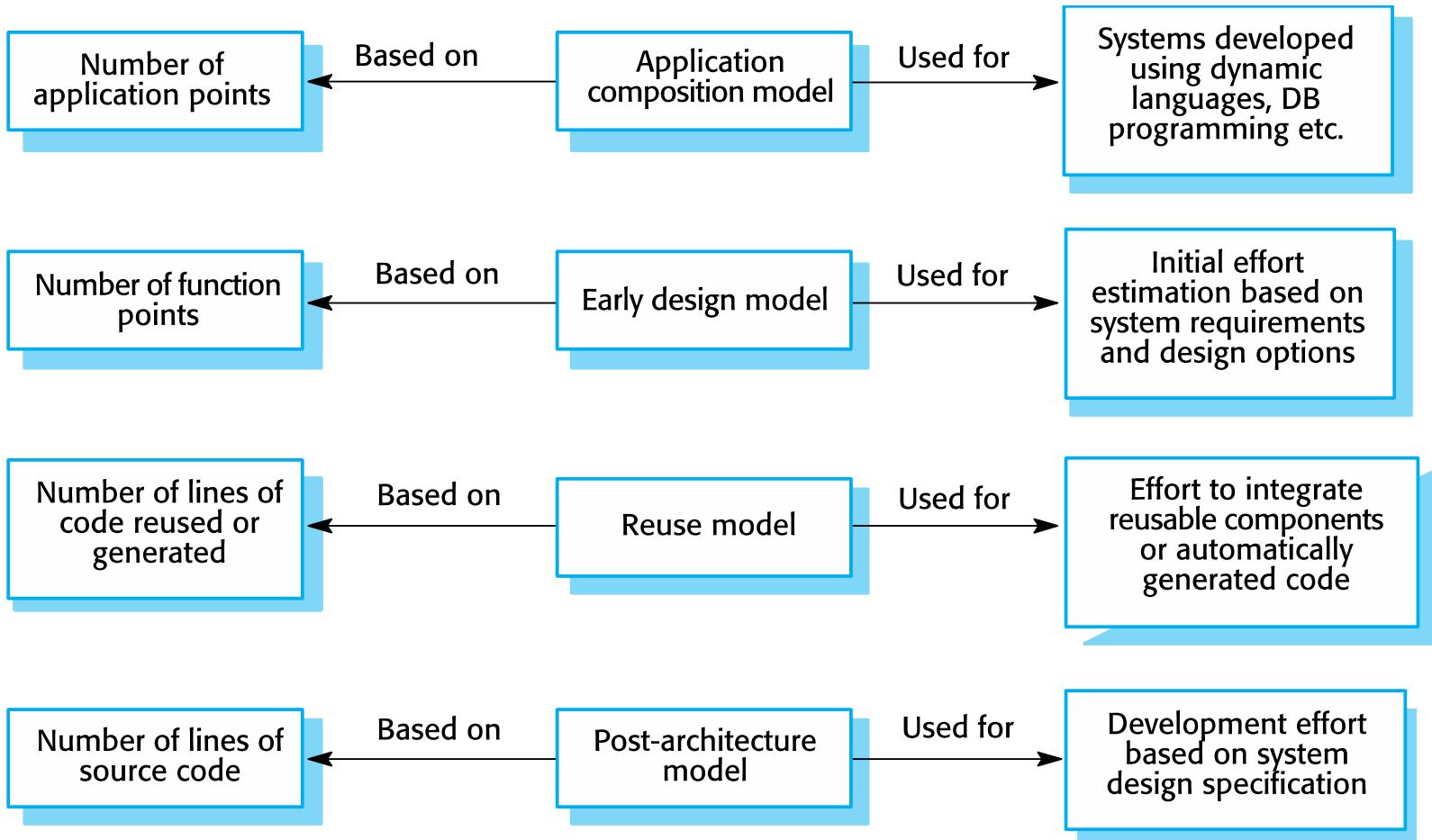
- ✧ An empirical model based on project experience.
- ✧ Well-documented, ‘independent’ model which is not tied to a specific software vendor.
- ✧ Long history from initial version published in 1981 (COCOMO-81) through various instantiations to COCOMO 2.
- ✧ COCOMO 2 takes into account different approaches to software development, reuse, etc.

# COCOMO 2 models

---

- ✧ COCOMO 2 incorporates a range of sub-models that produce increasingly detailed software estimates.
- ✧ The sub-models in COCOMO 2 are:
  - **Application composition model.** Used when software is composed from existing parts.
  - **Early design model.** Used when requirements are available but design has not yet started.
  - **Reuse model.** Used to compute the effort of integrating reusable components.
  - **Post-architecture model.** Used once the system architecture has been designed and more information about the system is available.

# COCOMO estimation models



# Application composition model

---

- ✧ Supports prototyping projects and projects where there is extensive reuse.
- ✧ Based on standard estimates of developer productivity in application (object) points/month.
- ✧ Takes software tool use into account.
- ✧ Formula is
  - $PM = ( NAP \cdot (1 - \%reuse/100) ) / PROD$
  - PM is the effort in person-months, NAP is the number of application points and PROD is the productivity.

# Application-point productivity

---

Developer's experience and capability	Very low	Low	Nominal	High	Very high
ICASE maturity and capability	Very low	Low	Nominal	High	Very high
PROD (NAP/month)	4	7	13	25	50

## Early design model

---

- ✧ Estimates can be made after the requirements have been agreed.
- ✧ Based on a standard formula for algorithmic models
- ✧  $PM = A^{\wedge} \text{Size}^B \wedge M$  where
  - $M = PERS^{\wedge} RCPX^{\wedge} RUSE^{\wedge} PDIF^{\wedge} PREX^{\wedge} FCIL^{\wedge} SCED;$
  - $A = 2.94$  in initial calibration,
  - Size in KLOC,
  - $B$  varies from 1.1 to 1.24 depending on novelty of the project, development flexibility, risk management approaches and the process maturity.

# Multipliers

---

- ✧ Multipliers reflect the capability of the developers, the non-functional requirements, the familiarity with the development platform, etc.
  - RCPX - product reliability and complexity;
  - RUSE - the reuse required;
  - PDIF - platform difficulty;
  - PREX - personnel experience;
  - PERS - personnel capability;
  - SCED - required schedule;
  - FCIL - the team support facilities.

# The reuse model

---

- ✧ Takes into account black-box code that is reused without change and code that has to be adapted to integrate it with new code.
- ✧ There are two versions:
  - Black-box reuse where code is not modified. An effort estimate (PM) is computed.
  - White-box reuse where code is modified. A size estimate equivalent to the number of lines of new source code is computed. This then adjusts the size estimate for new code.

# Reuse model estimates 1

---

- ✧ For generated code:
- ✧ 
$$PM = (\text{ASLOC} * \text{AT}/100)/\text{ATPROD}$$
  - ASLOC is the number of lines of generated code
  - AT is the percentage of code automatically generated.
  - ATPROD is the productivity of engineers in integrating this code.

## Reuse model estimates 2

---

- ✧ When code has to be understood and integrated:
- ✧  $\text{ESLOC} = \text{ASLOC} * (1-\text{AT}/100) * \text{AAM}$ .
  - ASLOC and AT as before.
  - AAM is the adaptation adjustment multiplier computed from the costs of changing the reused code, the costs of understanding how to integrate the code and the costs of reuse decision making.

## Post-architecture level

---

- ✧ Uses the same formula as the early design model but with 17 rather than 7 associated multipliers.
- ✧ The code size is estimated as:
  - Number of lines of new code to be developed;
  - Estimate of equivalent number of lines of new code computed using the reuse model;
  - An estimate of the number of lines of code that have to be modified according to requirements changes.

# The exponent term

---

- ✧ This depends on 5 scale factors (see next slide). Their sum/100 is added to 1.01
- ✧ A company takes on a project in a new domain. The client has not defined the process to be used and has not allowed time for risk analysis. The company has a CMM level 2 rating.
  - Precedenteness - new project (4)
  - Development flexibility - no client involvement - Very high (1)
  - Architecture/risk resolution - No risk analysis - V. Low .(5)
  - Team cohesion - new team - nominal (3)
  - Process maturity - some control - nominal (3)
- ✧ Scale factor is therefore 1.17.

# Scale factors used in the exponent computation in the post-architecture model

Scale factor	Explanation
Architecture/risk resolution	Reflects the extent of risk analysis carried out. Very low means little analysis; extra-high means a complete and thorough risk analysis.
Development flexibility	Reflects the degree of flexibility in the development process. Very low means a prescribed process is used; extra-high means that the client sets only general goals.
Precedentedness	Reflects the previous experience of the organization with this type of project. Very low means no previous experience; extra-high means that the organization is completely familiar with this application domain.
Process maturity	Reflects the process maturity of the organization. The computation of this value depends on the CMM Maturity Questionnaire, but an estimate can be achieved by subtracting the CMM process maturity level from 5.
Team cohesion	Reflects how well the development team knows each other and work together. Very low means very difficult interactions; extra-high means an integrated and effective team with no communication problems.

# Multipliers

---

## ✧ Product attributes

- Concerned with required characteristics of the software product being developed.

## ✧ Computer attributes

- Constraints imposed on the software by the hardware platform.

## ✧ Personnel attributes

- Multipliers that take the experience and capabilities of the people working on the project into account.

## ✧ Project attributes

- Concerned with the particular characteristics of the software development project.

# The effect of cost drivers on effort estimates

---

<b>Exponent value</b>	<b>1.17</b>
System size (including factors for reuse and requirements volatility)	128,000 DSI
<b>Initial COCOMO estimate without cost drivers</b>	<b>730 person-months</b>
Reliability	Very high, multiplier = 1.39
Complexity	Very high, multiplier = 1.3
Memory constraint	High, multiplier = 1.21
Tool use	Low, multiplier = 1.12
Schedule	Accelerated, multiplier = 1.29
<b>Adjusted COCOMO estimate</b>	<b>2,306 person-months</b>

# The effect of cost drivers on effort estimates

---

Exponent value	1.17
Reliability	Very low, multiplier = 0.75
Complexity	Very low, multiplier = 0.75
Memory constraint	None, multiplier = 1
Tool use	Very high, multiplier = 0.72
Schedule	Normal, multiplier = 1
<b>Adjusted COCOMO estimate</b>	<b>295 person-months</b>

# Project duration and staffing

---

- ✧ As well as effort estimation, managers must estimate the calendar time required to complete a project and when staff will be required.
- ✧ Calendar time can be estimated using a COCOMO 2 formula
  - $TDEV = 3^{\wedge} (PM)^{(0.33+0.2*(B-1.01))}$
  - PM is the effort computation and B is the exponent computed as discussed above (B is 1 for the early prototyping model). This computation predicts the nominal schedule for the project.
- ✧ The time required is independent of the number of people working on the project.

# Staffing requirements

---

- ✧ Staff required can't be computed by diving the development time by the required schedule.
- ✧ The number of people working on a project varies depending on the phase of the project.
- ✧ The more people who work on the project, the more total effort is usually required.
- ✧ A very rapid build-up of people often correlates with schedule slippage.

# Key points

---

- ✧ The price charged for a system does not just depend on its estimated development costs and the profit required by the development company. Organizational factors may mean that the price is increased to compensate for increased risk or decreased to gain competitive advantage.
- ✧ Software is often priced to gain a contract and the functionality of the system is then adjusted to meet the estimated price.
- ✧ Plan-driven development is organized around a complete project plan that defines the project activities, the planned effort, the activity schedule and who is responsible for each activity.

# Key points

---

- ✧ Project scheduling involves the creation of various graphical representations of part of the project plan. Bar charts, which show the activity duration and staffing timelines, are the most commonly used schedule representations.
- ✧ A project milestone is a predictable outcome of an activity or set of activities. At each milestone, a formal report of progress should be presented to management. A deliverable is a work product that is delivered to the project customer.
- ✧ The agile planning game involves the whole team in project planning. The plan is developed incrementally and, if problems arise, it is adjusted so that software functionality is reduced instead of delaying the delivery of an increment.

# Key points

---

- ✧ Estimation techniques for software may be experience-based, where managers judge the effort required, or algorithmic, where the effort required is computed from other estimated project parameters.
- ✧ The COCOMO II costing model is a mature algorithmic cost model that takes project, product, hardware and personnel attributes into account when formulating a cost estimate.

# Chapter 33

---

## ■ Estimation for Software Projects

*Slide Set to accompany*

*Software Engineering: A Practitioner's Approach, 8/e*  
**by Roger S. Pressman and Bruce R. Maxim**

Slides copyright © 1996, 2001, 2005, 2009, 2014 by Roger S. Pressman

***For non-profit educational use only***

May be reproduced ONLY for student use at the university level when used in conjunction with *Software Engineering: A Practitioner's Approach, 8/e*. Any other reproduction or use is prohibited without the express written permission of the author.

All copyright information MUST appear if these slides are posted on a website for student use.

# Software Project Planning

The overall goal of project planning is to establish a pragmatic strategy for controlling, tracking, and monitoring a complex technical project.

Why?

*So the end result gets done on time, with quality!*

# Project Planning Task Set-I

---

- Establish project scope
- Determine feasibility
- Analyze risks
  - Risk analysis is considered in detail in Chapter 25.
- Define required resources
  - Determine require human resources
  - Define reusable software resources
  - Identify environmental resources

# Project Planning Task Set-II

---

- Estimate cost and effort
  - Decompose the problem
  - Develop two or more estimates using size, function points, process tasks or use-cases
  - Reconcile the estimates
- Develop a project schedule
  - Scheduling is considered in detail in Chapter 34.
    - Establish a meaningful task set
    - Define a task network
    - Use scheduling tools to develop a timeline chart
    - Define schedule tracking mechanisms

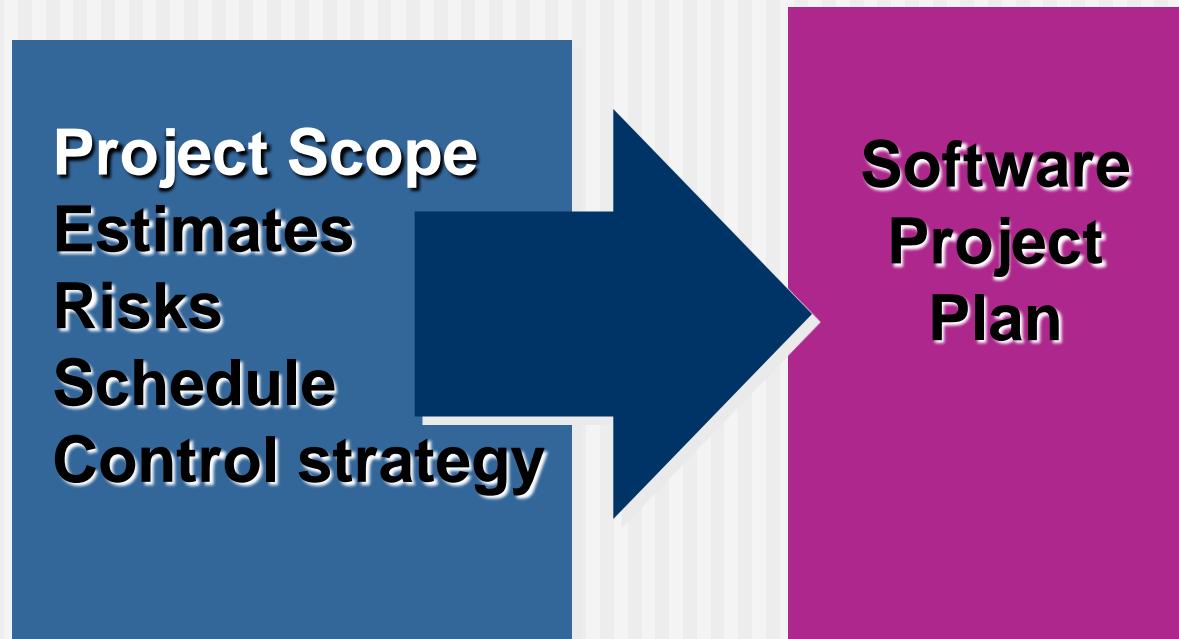
# Estimation

---

- Estimation of resources, cost, and schedule for a software engineering effort requires
  - experience
  - access to good historical information (metrics)
  - the courage to commit to quantitative predictions when qualitative information is all that exists
- Estimation carries inherent risk and this risk leads to uncertainty

# Write it Down!

---



# To Understand Scope ...

---

- Understand the customers needs
- understand the business context
- understand the project boundaries
- understand the customer's motivation
- understand the likely paths for change
- understand that ...

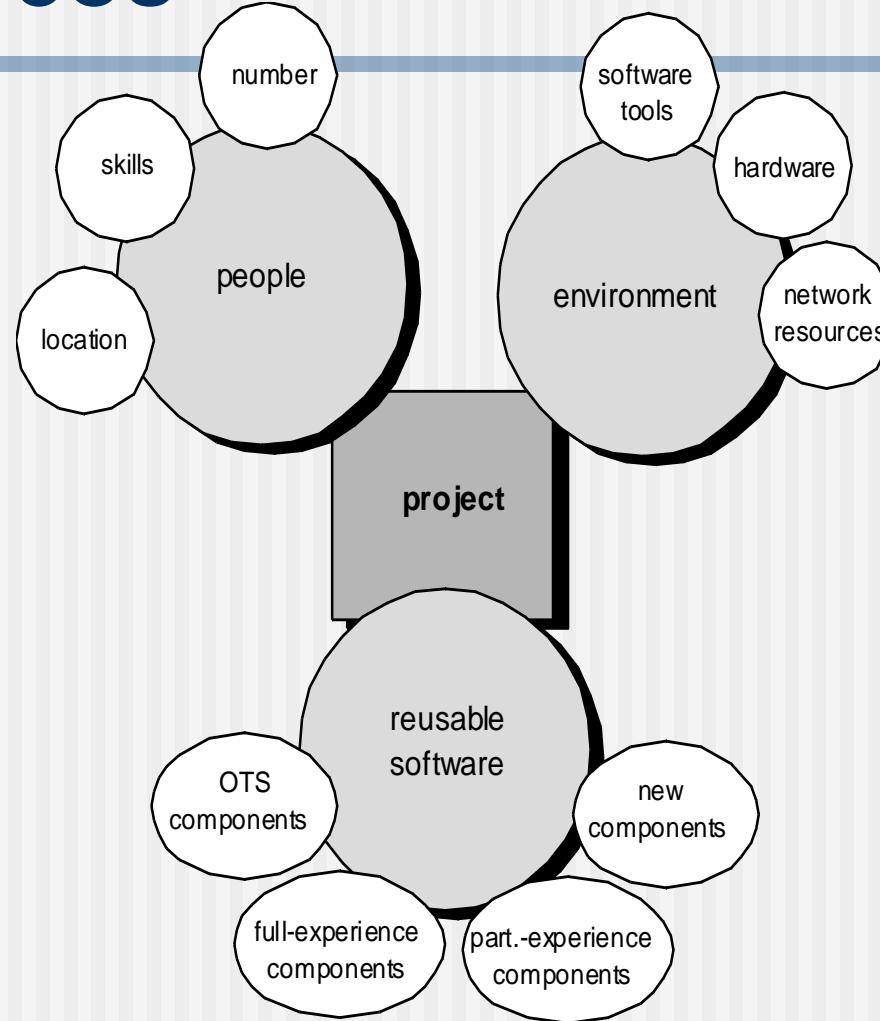
***Even when you understand,  
nothing is guaranteed!***

# What is Scope?

---

- *Software scope* describes
  - the functions and features that are to be delivered to end-users
  - the data that are input and output
  - the “content” that is presented to users as a consequence of using the software
  - the performance, constraints, interfaces, and reliability that *bound* the system.
- Scope is defined using one of two techniques:
  - A narrative description of software scope is developed after communication with all stakeholders.
  - A set of use-cases is developed by end-users.

# Resources



These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 8/e* (McGraw-Hill 2014). Slides copyright 2014 by Roger Pressman.

# Project Estimation

---



- Project scope must be understood
- Elaboration (decomposition) is necessary
- Historical metrics are very helpful
- At least two different techniques should be used
- Uncertainty is inherent in the process

# Estimation Techniques

---

- Past (similar) project experience
- Conventional estimation techniques
  - task breakdown and effort estimates
  - size (e.g., FP) estimates
- Empirical models
- Automated tools

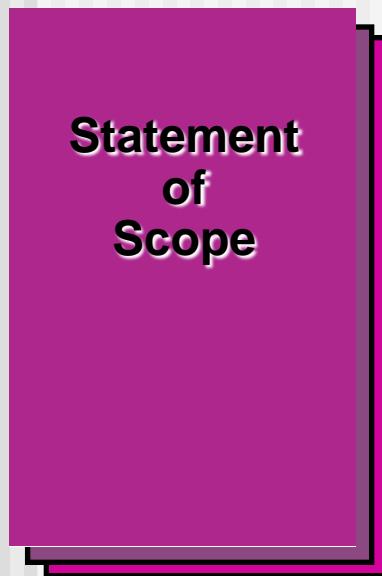


# Estimation Accuracy

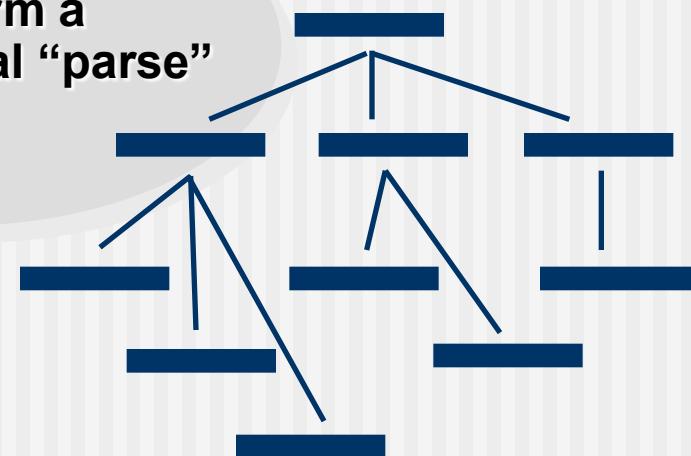
---

- Predicated on ...
  - the degree to which the planner has properly estimated the size of the product to be built
  - the ability to translate the size estimate into human effort, calendar time, and dollars (a function of the availability of reliable software metrics from past projects)
  - the degree to which the project plan reflects the abilities of the software team
  - the stability of product requirements and the environment that supports the software engineering effort.

# Functional Decomposition



Perform a Grammatical “parse”



functional  
decomposition

# Conventional Methods: LOC/FP Approach

---

- compute LOC/FP using estimates of information domain values
- use historical data to build estimates for the project

# Example: LOC Approach

Function	Estimated LOC
user interface and control facilities (UICF)	2,300
two-dimensional geometric analysis (2D GA)	5,300
three-dimensional geometric analysis (3D GA)	6,800
database management (DBM)	3,380
computer graphics display facilities (CGDF)	4,980
peripheral control (PC)	2,100
design analysis modules (DAM)	8,400
<i>estimated lines of code</i>	<b>33,200</b>

Average productivity for systems of this type = 620 LOC/pm.

Burdened labor rate = \$8000 per month, the cost per line of code is approximately \$13.

Based on the LOC estimate and the historical productivity data, the total estimated project cost is **\$431,000 and the estimated effort is 54 person-months.**

# Example: FP Approach

Information Domain Value	opt.	likely	pess.	est. count	weight	FP-count
number of inputs	20	24	30	24	4	97
number of outputs	12	18	22	18	5	78
number of inquiries	16	22	28	22	5	88
number of files	4	4	8	4	10	42
number of external interfaces	2	2	3	2	7	18
count-total						321

The estimated number of FP is derived:

$$FP_{estimated} = \text{count-total} \times [0.65 + 0.01 \times 3 \times S(F_i)]$$

$$FP_{estimated} = 375$$

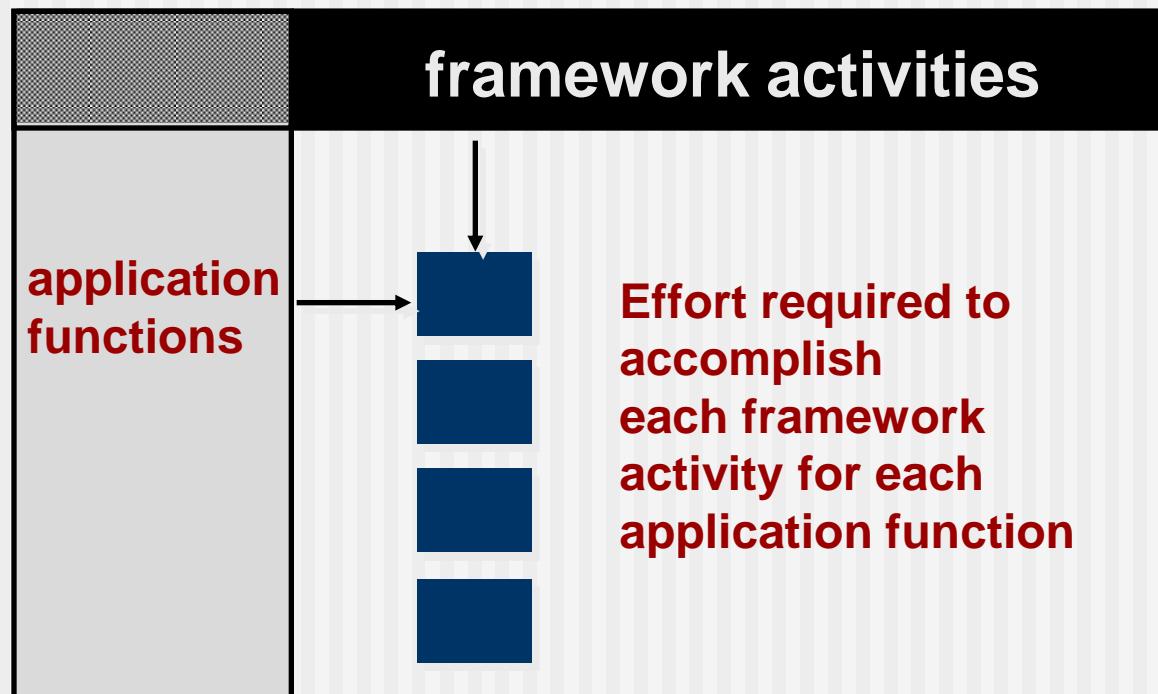
organizational average productivity = 6.5 FP/pm.

burdened labor rate = \$8000 per month, approximately \$1230/FP.

Based on the FP estimate and the historical productivity data, **total estimated project cost is \$461,000 and estimated effort is 58 person-months.**

# Process-Based Estimation

Obtained from “process framework”



# Process-Based Estimation Example

Activity →	CC	Planning	Risk Analysis	Engineering		Construction Release		CE	Totals
Task →				analysis	design	code	test		
Function									
UICF				0.50	2.50	0.40	5.00	n/a	8.40
2DGA				0.75	4.00	0.60	2.00	n/a	7.35
3DGA				0.50	4.00	1.00	3.00	n/a	8.50
CGDF				0.50	3.00	1.00	1.50	n/a	6.00
DSM				0.50	3.00	0.75	1.50	n/a	5.75
PCF				0.25	2.00	0.50	1.50	n/a	4.25
DAM				0.50	2.00	0.50	2.00	n/a	5.00
Totals	0.25	0.25	0.25	3.50	20.50	4.50	16.50		46.00
% effort	1%	1%	1%	8%	45%	10%	36%		

CC = customer communication CE = customer evaluation

Based on an average burdened labor rate of \$8,000 per month, **the total estimated project cost is \$368,000 and the estimated effort is 46 person-months.**

# Tool-Based Estimation

---

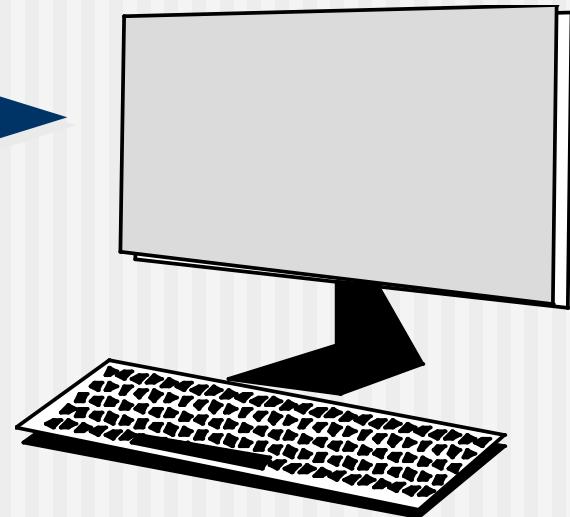
**project characteristics**



**calibration factors**



**LOC/FP data**



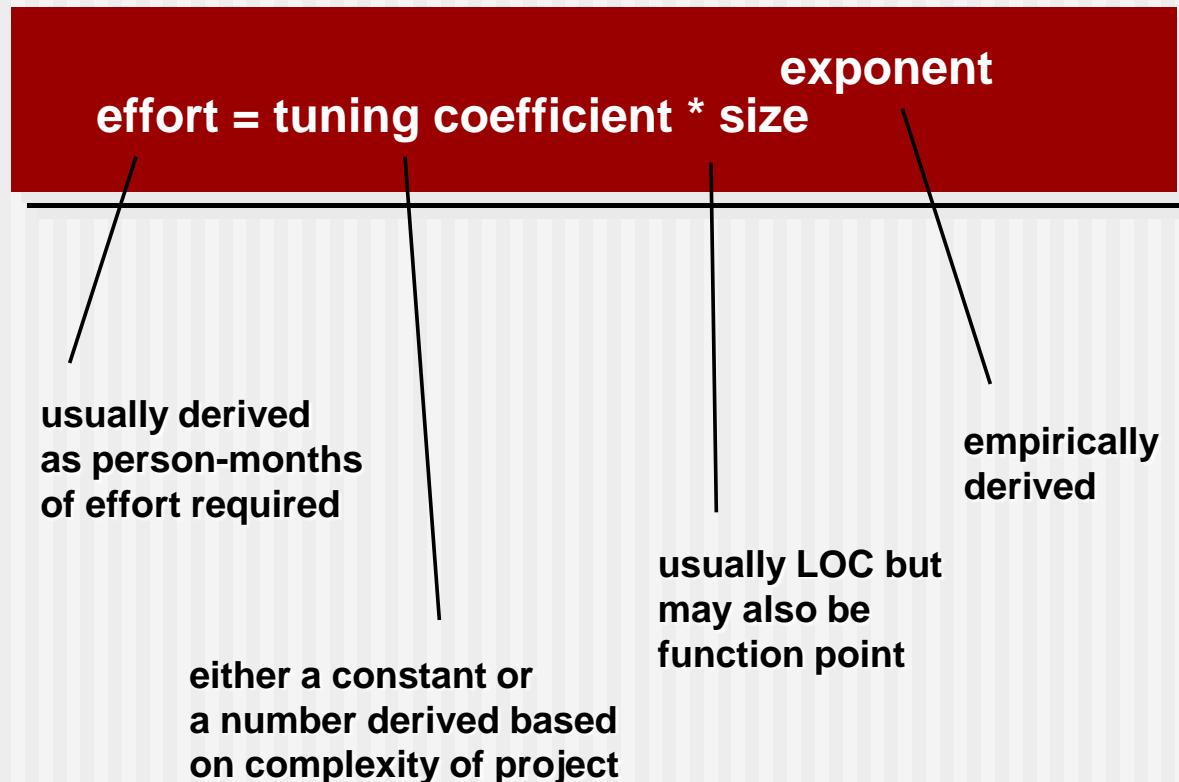
# Estimation with Use-Cases

	use cases	scenarios	pages	scenarios	pages	LOC	LOC estimate
User interface subsystem	6	10	6	12	5	560	3,366
Engineering subsystem group	10	20	8	16	8	3100	31,233
Infrastructure subsystem group	5	6	5	10	6	1650	7,970
Total LOC estimate				Ê	Ê	Ê	Ê
				Ê	Ê	Ê	42,568

Using 620 LOC/pm as the average productivity for systems of this type and a burdened labor rate of \$8000 per month, the cost per line of code is approximately \$13. Based on the use-case estimate and the historical productivity data, **the total estimated project cost is \$552,000 and the estimated effort is 68 person-months.**

# Empirical Estimation Models

*General form:*



# COCOMO-II

---

- COCOMO II is actually a hierarchy of estimation models that address the following areas:
  - *Application composition model.* Used during the early stages of software engineering, when prototyping of user interfaces, consideration of software and system interaction, assessment of performance, and evaluation of technology maturity are paramount.
  - *Early design stage model.* Used once requirements have been stabilized and basic software architecture has been established.
  - *Post-architecture-stage model.* Used during the construction of the software.

# The Software Equation

---

*A dynamic multivariable model*

$$E = [LOC \times B^{0.333}/P]^3 \times (1/t^4)$$

where

E = effort in person-months or person-years

t = project duration in months or years

B = “special skills factor”

P = “productivity parameter”

# Estimation for OO Projects-I

---

- Develop estimates using effort decomposition, FP analysis, and any other method that is applicable for conventional applications.
- Using object-oriented requirements modeling (Chapter 6), develop use-cases and determine a count.
- From the analysis model, determine the number of key classes (called analysis classes in Chapter 6).
- Categorize the type of interface for the application and develop a multiplier for support classes:

<b>Interface type</b>	<b>Multiplier</b>
■ No GUI	2.0
■ Text-based user interface	2.25
■ GUI	2.5
■ Complex GUI	3.0

# Estimation for OO Projects-II

---

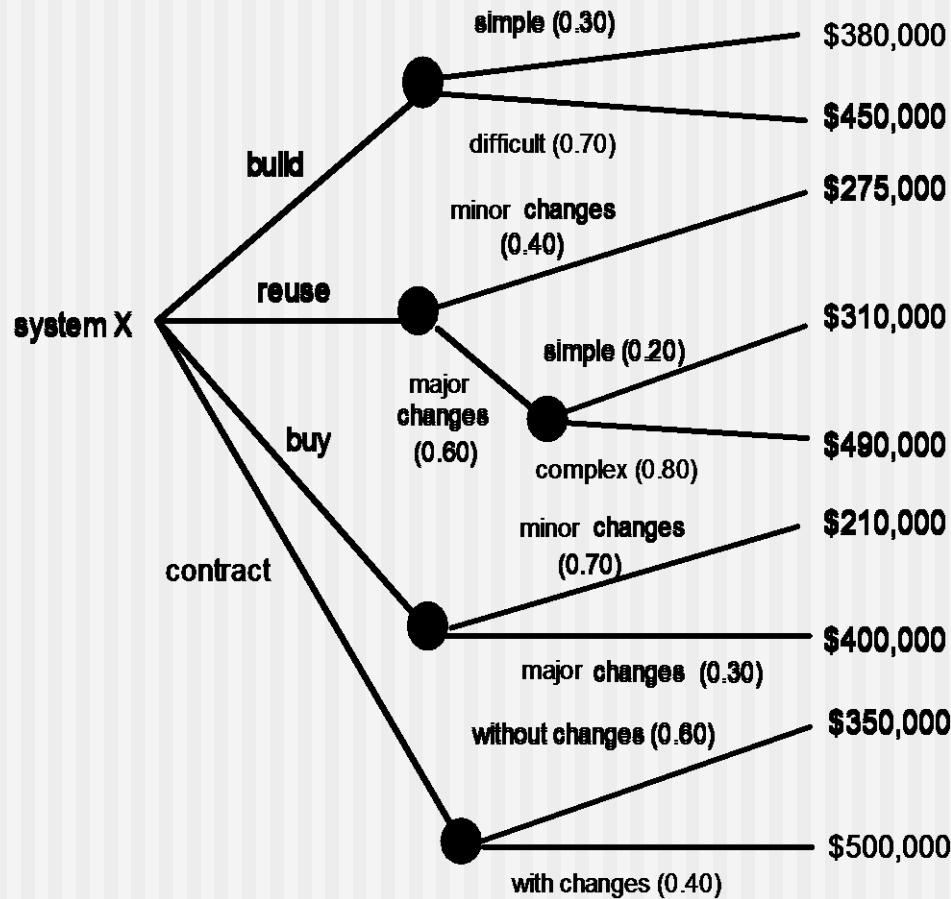
- Multiply the number of key classes (step 3) by the multiplier to obtain an estimate for the number of support classes.
- Multiply the total number of classes (key + support) by the average number of work-units per class. Lorenz and Kidd suggest 15 to 20 person-days per class.
- Cross check the class-based estimate by multiplying the average number of work-units per use-case

# Estimation for Agile Projects

---

- Each user scenario (a mini-use-case) is considered separately for estimation purposes.
- The scenario is decomposed into the set of software engineering tasks that will be required to develop it.
- Each task is estimated separately. Note: estimation can be based on historical data, an empirical model, or “experience.”
  - Alternatively, the ‘volume’ of the scenario can be estimated in LOC, FP or some other volume-oriented measure (e.g., use-case count).
- Estimates for each task are summed to create an estimate for the scenario.
  - Alternatively, the volume estimate for the scenario is translated into effort using historical data.
- The effort estimates for all scenarios that are to be implemented for a given software increment are summed to develop the effort estimate for the increment.

# The Make-Buy Decision



# Computing Expected Cost

---

**expected cost =**

$$\sum_i (\text{path probability}_i \times \text{estimated path cost}_i)$$

*For example, the expected cost to build is:*

$$\begin{aligned}\text{expected cost}_{\text{build}} &= 0.30 (\$380K) + 0.70 (\$450K) \\ &= \$429 K\end{aligned}$$

*similarly,*

$$\text{expected cost}_{\text{reuse}} = \$382K$$

$$\text{expected cost}_{\text{buy}} = \$267K$$

$$\text{expected cost}_{\text{contr}} = \$410K$$

# Chapter 34

---

## ■ Project Scheduling

*Slide Set to accompany*

*Software Engineering: A Practitioner's Approach, 8/e*  
**by Roger S. Pressman and Bruce R. Maxim**

Slides copyright © 1996, 2001, 2005, 2009, 2014 by Roger S. Pressman

***For non-profit educational use only***

May be reproduced ONLY for student use at the university level when used in conjunction with *Software Engineering: A Practitioner's Approach, 8/e*. Any other reproduction or use is prohibited without the express written permission of the author.

All copyright information MUST appear if these slides are posted on a website for student use.

# Why Are Projects Late?

---

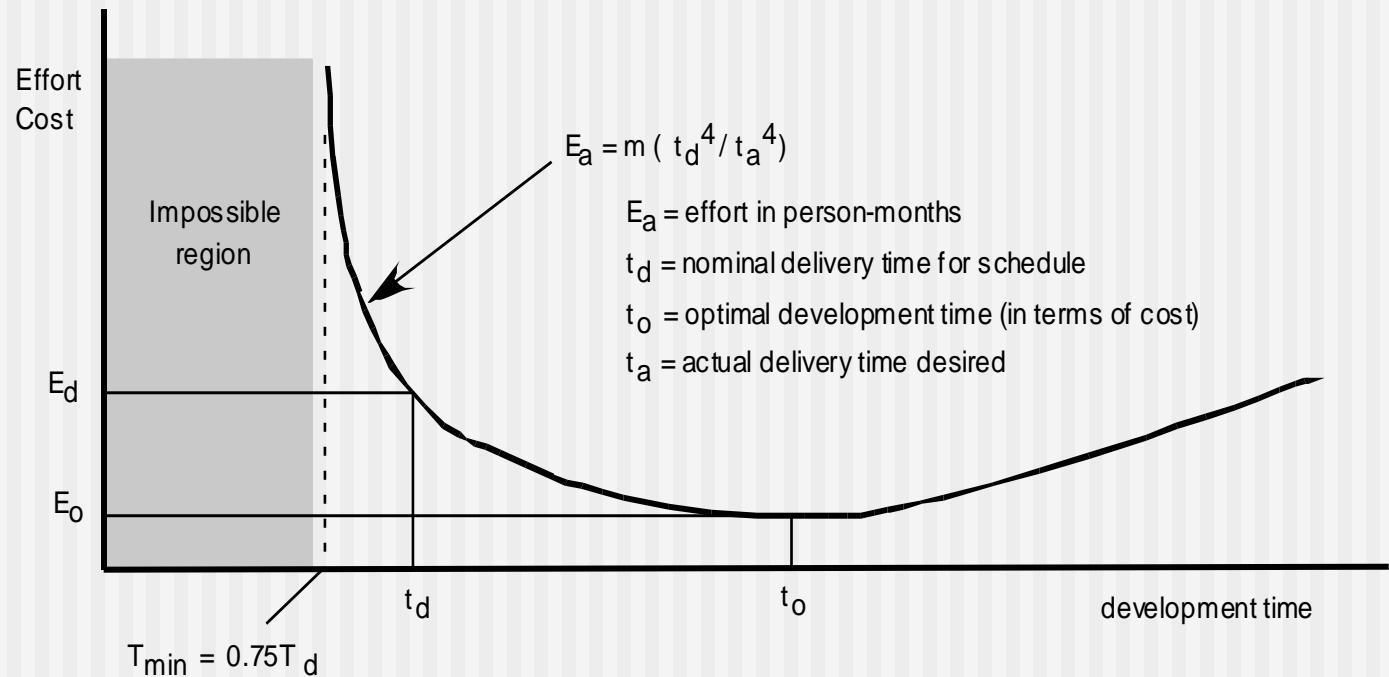
- an **unrealistic deadline** established by someone outside the software development group
- **changing customer requirements** that are not reflected in schedule changes;
- an **honest underestimate** of the amount of effort and/or the number of resources that will be required to do the job;
- **predictable and/or unpredictable risks** that were not considered when the project commenced;
- **technical difficulties** that could not have been foreseen in advance;
- **human difficulties** that could not have been foreseen in advance;
- **miscommunication** among project staff that results in delays;
- a failure by project management to recognize that **the project is falling behind schedule** and **a lack of action to correct the problem**

# Scheduling Principles

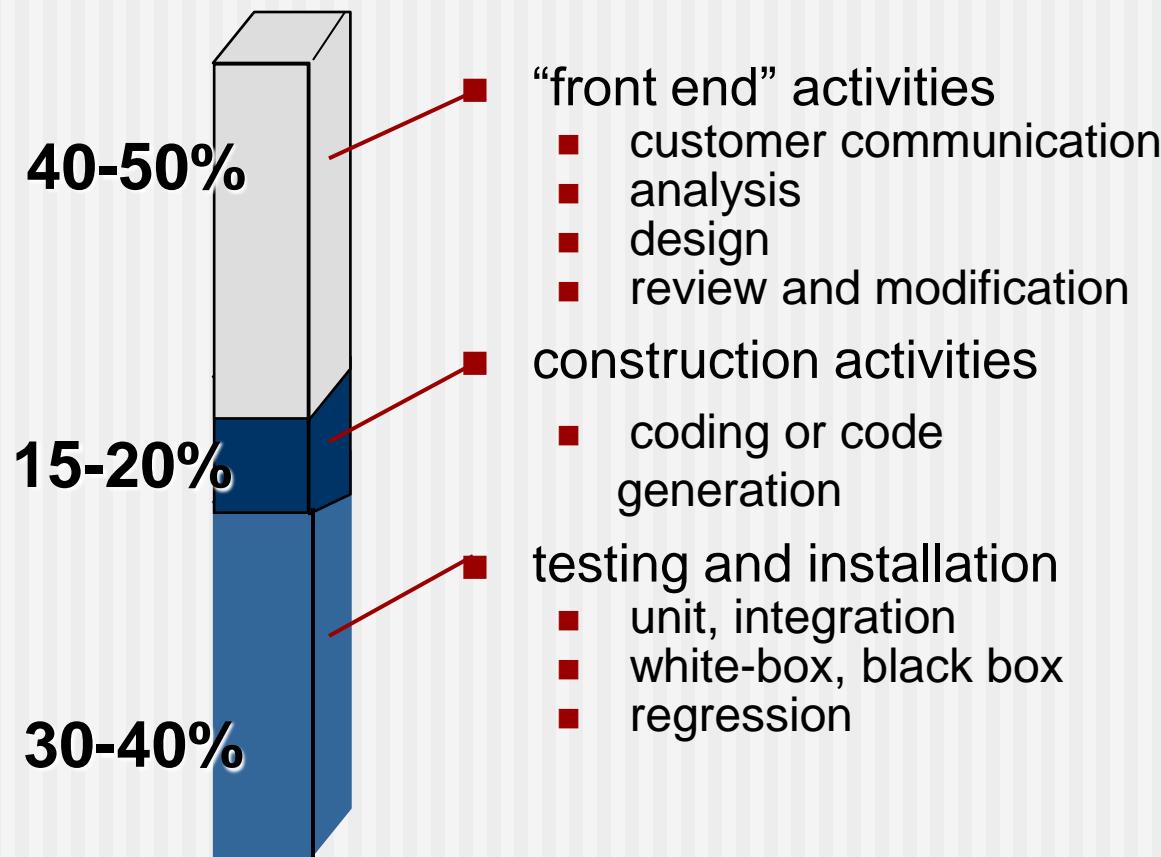
---

- compartmentalization—define distinct tasks
- interdependency—indicate task interrelationship
- effort validation—be sure resources are available
- defined responsibilities—people must be assigned
- defined outcomes—each task must have an output
- defined milestones—review for quality

# Effort and Delivery Time



# Effort Allocation

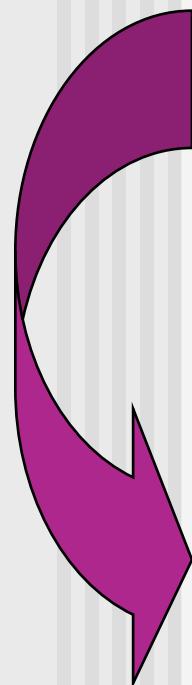


# Defining Task Sets

---

- determine type of project
- assess the degree of rigor required
- identify adaptation criteria
- select appropriate software engineering tasks

# Task Set Refinement

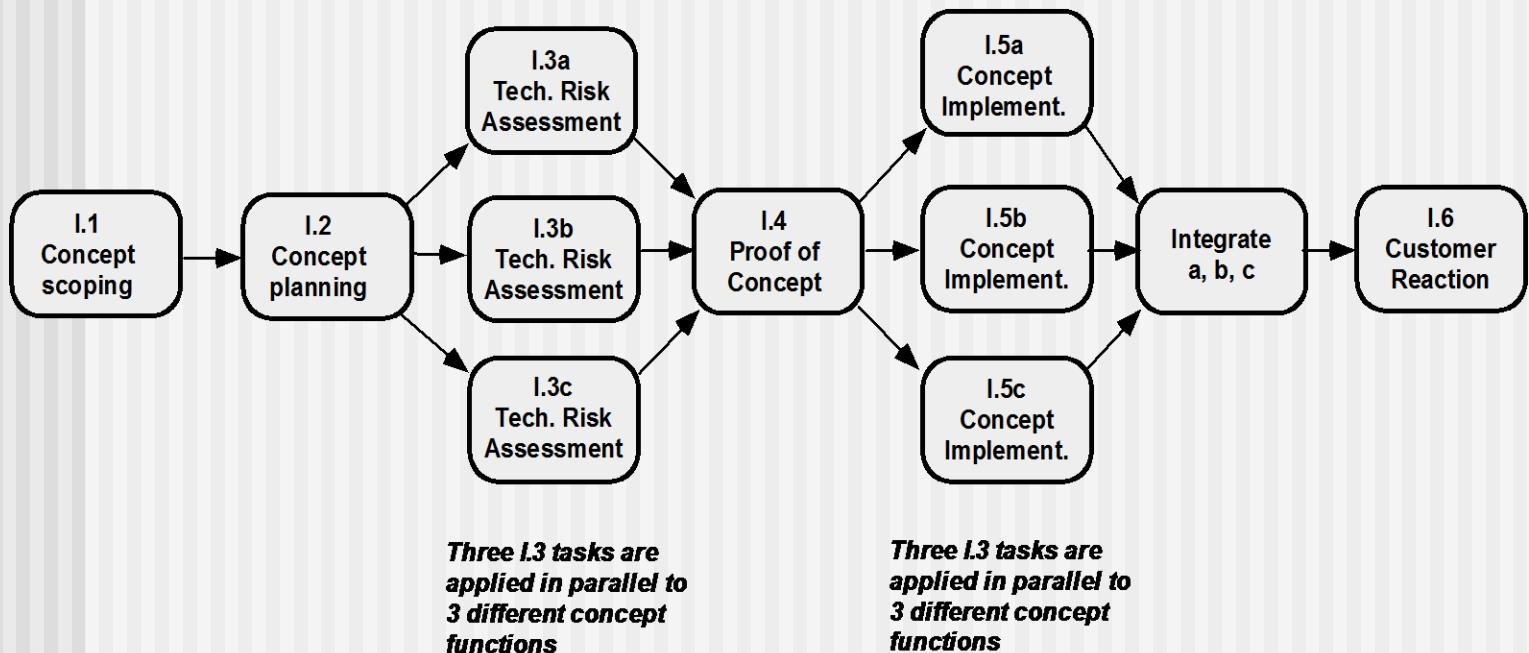


## 1.1 **Concept scoping** determines the overall scope of the project.

- Task definition: Task 1.1 Concept Scoping
  - 1.1.1 Identify need, benefits and potential customers;
  - 1.1.2 Define desired output/control and input events that drive the application;  
Begin Task 1.1.2
    - 1.1.2.1 FTR: Review written description of need  
FTR indicates that a formal technical review (Chapter 26) is to be conducted.
    - 1.1.2.2 Derive a list of customer visible outputs/inputs
    - 1.1.2.3 FTR: Review outputs/inputs with customer and revise as required;  
endtask Task 1.1.2
  - 1.1.3 Define the functionality/behavior for each major function;  
Begin Task 1.1.3
    - 1.1.3.1 FTR: Review output and input data objects derived in task 1.1.2;
    - 1.1.3.2 Derive a model of functions/behaviors;
    - 1.1.3.3 FTR: Review functions/behaviors with customer and revise as required;  
endtask Task 1.1.3
  - 1.1.4 Isolate those elements of the technology to be implemented in software;
  - 1.1.5 Research availability of existing software;
  - 1.1.6 Define technical feasibility;
  - 1.1.7 Make quick estimate of size;
  - 1.1.8 Create a Scope Definition;  
endTask definition: Task 1.1

**is refined to**

# Define a Task Network

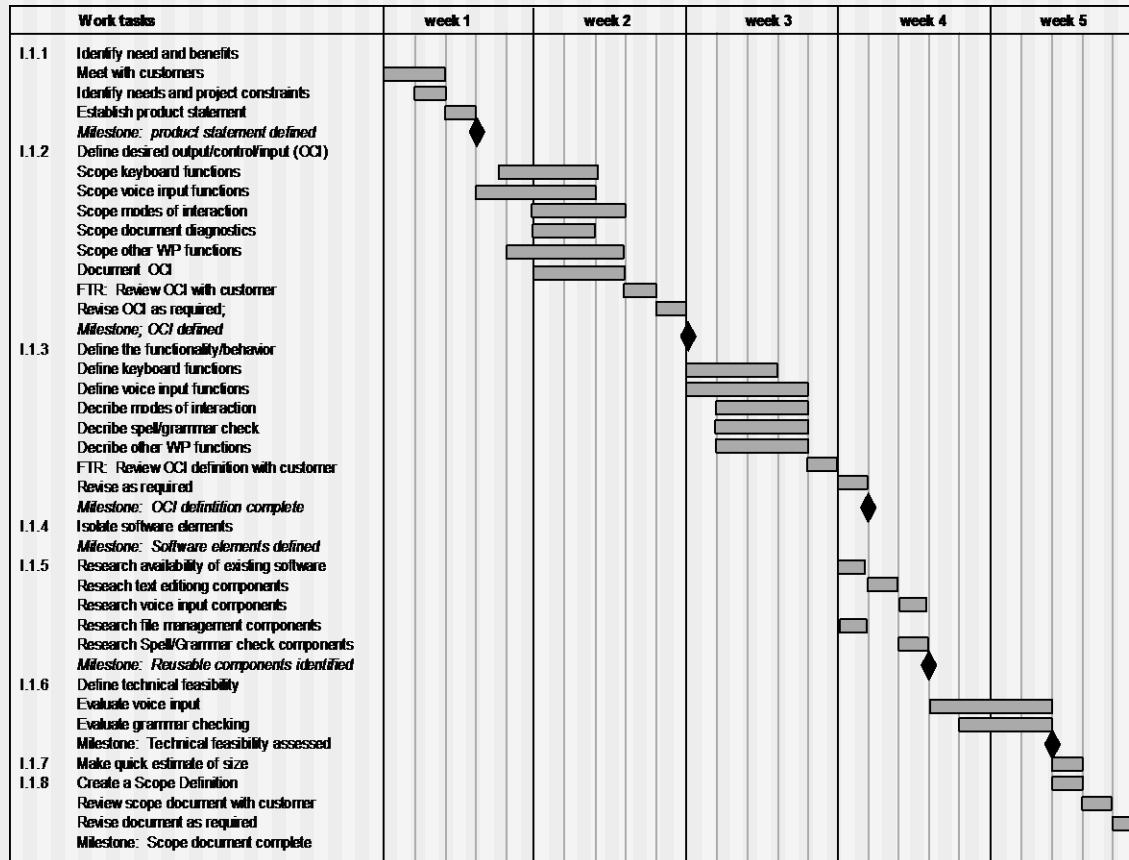


# Timeline Charts

---

Tasks	Week 1	Week 2	Week 3	Week 4		Week n
Task 1						
Task 2						
Task 3						
Task 4						
Task 5						
Task 6						
Task 7						
Task 8						
Task 9						
Task 10						
Task 11						
Task 12						

# Use Automated Tools to Derive a Timeline Chart



# Schedule Tracking

---

- conduct periodic project status meetings in which each team member reports progress and problems.
- evaluate the results of all reviews conducted throughout the software engineering process.
- determine whether formal project milestones (the diamonds shown in Figure 34.3) have been accomplished by the scheduled date.
- compare actual start-date to planned start-date for each project task listed in the resource table (Figure 34.4).
- meet informally with practitioners to obtain their subjective assessment of progress to date and problems on the horizon.
- use earned value analysis (Section 34.6) to assess progress quantitatively.

# Progress on an OO Project-I

---

- *Technical milestone: OO analysis completed*

- All classes and the class hierarchy have been defined and reviewed.
- Class attributes and operations associated with a class have been defined and reviewed.
- Class relationships (Chapter 10) have been established and reviewed.
- A behavioral model (Chapter 11) has been created and reviewed.
- Reusable classes have been noted.

- *Technical milestone: OO design completed*

- The set of subsystems (Chapter 12) has been defined and reviewed.
- Classes are allocated to subsystems and reviewed.
- Task allocation has been established and reviewed.
- Responsibilities and collaborations (Chapter 12) have been identified.
- Attributes and operations have been designed and reviewed.
- The communication model has been created and reviewed.

# Progress on an OO Project-II

---

- *Technical milestone: OO programming completed*
  - Each new class has been implemented in code from the design model.
  - Extracted classes (from a reuse library) have been implemented.
  - Prototype or increment has been built.
- *Technical milestone: OO testing*
  - The correctness and completeness of OO analysis and design models has been reviewed.
  - A class-responsibility-collaboration network (Chapter 10) has been developed and reviewed.
  - Test cases are designed and class-level tests (Chapter 24) have been conducted for each class.
  - Test cases are designed and cluster testing (Chapter 24) is completed and the classes are integrated.
  - System level tests have been completed.

# Earned Value Analysis (EVA)

- Earned value
  - is a measure of progress
  - enables us to assess the “percent of completeness” of a project using quantitative analysis rather than rely on a gut feeling
  - “provides accurate and reliable readings of performance from as early as 15 percent into the project.” [Fle98]

# Computing Earned Value-I

---

- The *budgeted cost of work scheduled* (BCWS) is determined for each work task represented in the schedule.
  - $BCWS_i$  is the effort planned for work task  $i$ .
  - To determine progress at a given point along the project schedule, the value of BCWS is the sum of the  $BCWS_i$  values for all work tasks that should have been completed by that point in time on the project schedule.
- The BCWS values for all work tasks are summed to derive the *budget at completion*, BAC. Hence,

$$BAC = \sum (BCWS_k) \text{ for all tasks } k$$

# Computing Earned Value-II

---

- Next, the value for *budgeted cost of work performed* (BCWP) is computed.
  - The value for BCWP is the sum of the BCWS values for all work tasks that have actually been completed by a point in time on the project schedule.
- “the distinction between the BCWS and the BCWP is that the former represents the budget of the activities that were planned to be completed and the latter represents the budget of the activities that actually were completed.” [Wil99]
- Given values for BCWS, BAC, and BCWP, important progress indicators can be computed:
  - Schedule performance index,  $SPI = BCWP/BCWS$
  - Schedule variance,  $SV = BCWP - BCWS$
  - SPI is an indication of the efficiency with which the project is utilizing scheduled resources.

# Computing Earned Value-III

---

- Percent scheduled for completion =  $BCWS/BAC$ 
  - provides an indication of the percentage of work that should have been completed by time  $t$ .
- Percent complete =  $BCWP/BAC$ 
  - provides a quantitative indication of the percent of completeness of the project at a given point in time,  $t$ .
- *Actual cost of work performed, ACWP*, is the sum of the effort actually expended on work tasks that have been completed by a point in time on the project schedule. It is then possible to compute
  - Cost performance index,  $CPI = BCWP/ACWP$
  - Cost variance,  $CV = BCWP - ACWP$

# **Chapter 24 - Quality Management**

# Topics covered

---

- ✧ Software quality
- ✧ Software standards
- ✧ Reviews and inspections
- ✧ Quality management and agile development
- ✧ Software measurement

# Software quality management

---

- ✧ Concerned with ensuring that the required level of quality is achieved in a software product.
- ✧ Three principal concerns:
  - At the organizational level, quality management is concerned with establishing a framework of organizational processes and standards that will lead to high-quality software.
  - At the project level, quality management involves the application of specific quality processes and checking that these planned processes have been followed.
  - At the project level, quality management is also concerned with establishing a quality plan for a project. The quality plan should set out the quality goals for the project and define what processes and standards are to be used.

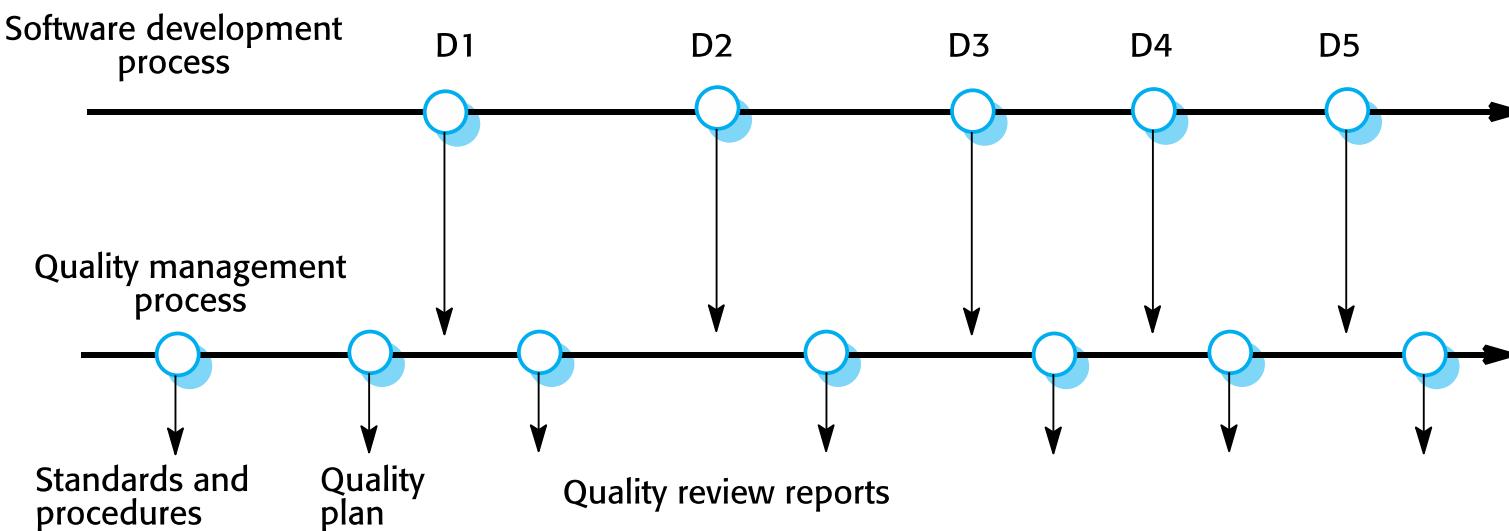
# Quality management activities

---

- ✧ Quality management provides an independent check on the software development process.
- ✧ The quality management process checks the project deliverables to ensure that they are consistent with organizational standards and goals
- ✧ The quality team should be independent from the development team so that they can take an objective view of the software. This allows them to report on software quality without being influenced by software development issues.

# Quality management and software development

---



# Quality planning

---

- ✧ A quality plan sets out the desired product qualities and how these are assessed and defines the most significant quality attributes.
- ✧ The quality plan should define the quality assessment process.
- ✧ It should set out which organisational standards should be applied and, where necessary, define new standards to be used.

# Quality plans

---

## ✧ Quality plan structure

- Product introduction;
- Product plans;
- Process descriptions;
- Quality goals;
- Risks and risk management.

## ✧ Quality plans should be short, succinct documents

- If they are too long, no-one will read them.

# Scope of quality management

---

- ✧ Quality management is particularly important for large, complex systems. The quality documentation is a record of progress and supports continuity of development as the development team changes.
- ✧ For smaller systems, quality management needs less documentation and should focus on establishing a quality culture.
- ✧ Techniques have to evolve when agile development is used.

---

# **Software quality**

# Software quality

---

- ✧ Quality, simplistically, means that a product should meet its specification.
- ✧ This is problematical for software systems
  - There is a tension between customer quality requirements (efficiency, reliability, etc.) and developer quality requirements (maintainability, reusability, etc.);
  - Some quality requirements are difficult to specify in an unambiguous way;
  - Software specifications are usually incomplete and often inconsistent.
- ✧ The focus may be ‘fitness for purpose’ rather than specification conformance.

# Software fitness for purpose

---

- ✧ Has the software been properly tested?
- ✧ Is the software sufficiently dependable to be put into use?
- ✧ Is the performance of the software acceptable for normal use?
- ✧ Is the software usable?
- ✧ Is the software well-structured and understandable?
- ✧ Have programming and documentation standards been followed in the development process?

# Non-functional characteristics

---

- ✧ The subjective quality of a software system is largely based on its non-functional characteristics.
- ✧ This reflects practical user experience – if the software's functionality is not what is expected, then users will often just work around this and find other ways to do what they want to do.
- ✧ However, if the software is unreliable or too slow, then it is practically impossible for them to achieve their goals.

# Software quality attributes

---

Safety	Understandability	Portability
Security	Testability	Usability
Reliability	Adaptability	Reusability
Resilience	Modularity	Efficiency
Robustness	Complexity	Learnability

# Quality conflicts

---

- ✧ It is not possible for any system to be optimized for all of these attributes – for example, improving robustness may lead to loss of performance.
- ✧ The quality plan should therefore define the most important quality attributes for the software that is being developed.
- ✧ The plan should also include a definition of the quality assessment process, an agreed way of assessing whether some quality, such as maintainability or robustness, is present in the product.

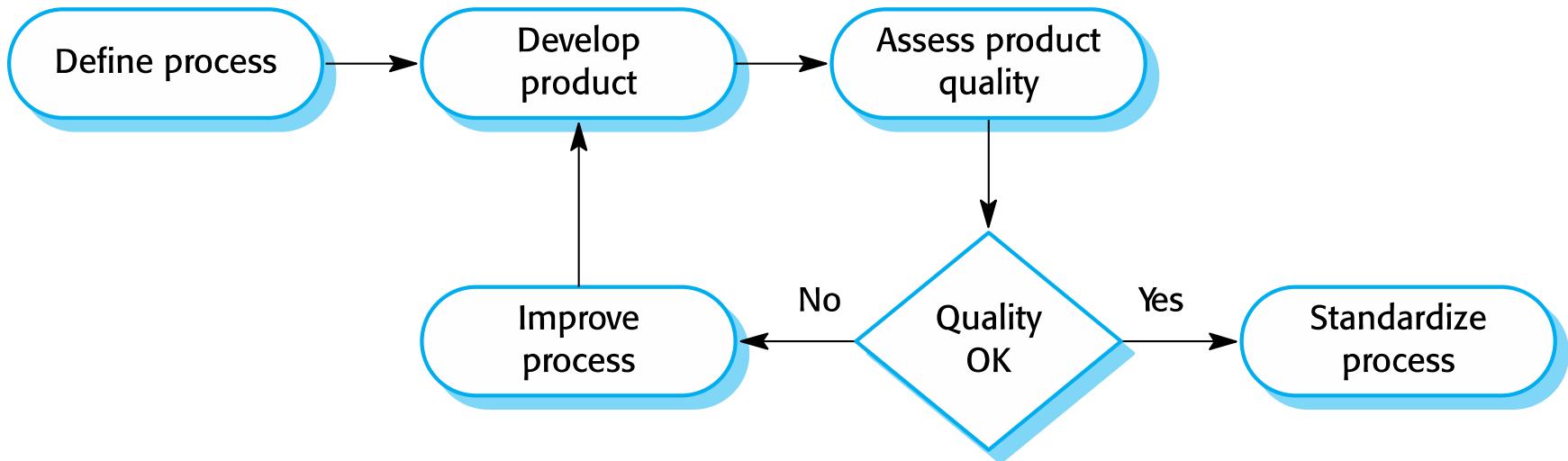
# Process and product quality

---

- ✧ The quality of a developed product is influenced by the quality of the production process.
- ✧ This is important in software development as some product quality attributes are hard to assess.
- ✧ However, there is a very complex and poorly understood relationship between software processes and product quality.
  - The application of individual skills and experience is particularly important in software development;
  - External factors such as the novelty of an application or the need for an accelerated development schedule may impair product quality.

# Process-based quality

---



# Quality culture

---

- ✧ Quality managers should aim to develop a ‘quality culture’ where everyone responsible for software development is committed to achieving a high level of product quality.
- ✧ They should encourage teams to take responsibility for the quality of their work and to develop new approaches to quality improvement.
- ✧ They should support people who are interested in the intangible aspects of quality and encourage professional behavior in all team members.

---

# **Software standards**

# Software standards

---

- ✧ Standards define the required attributes of a product or process. They play an important role in quality management.
- ✧ Standards may be international, national, organizational or project standards.

# Importance of standards

---

- ✧ Encapsulation of best practice- avoids repetition of past mistakes.
- ✧ They are a framework for defining what quality means in a particular setting i.e. that organization's view of quality.
- ✧ They provide continuity - new staff can understand the organisation by understanding the standards that are used.

# Product and process standards

---

## ✧ *Product standards*

- Apply to the software product being developed. They include document standards, such as the structure of requirements documents, documentation standards, such as a standard comment header for an object class definition, and coding standards, which define how a programming language should be used.

## ✧ *Process standards*

- These define the processes that should be followed during software development. Process standards may include definitions of specification, design and validation processes, process support tools and a description of the documents that should be written during these processes.

# Product and process standards

---

Product standards	Process standards
Design review form	Design review conduct
Requirements document structure	Submission of new code for system building
Method header format	Version release process
Java programming style	Project plan approval process
Project plan format	Change control process
Change request form	Test recording process

# Problems with standards

---

- ✧ They may not be seen as relevant and up-to-date by software engineers.
- ✧ They often involve too much bureaucratic form filling.
- ✧ If they are unsupported by software tools, tedious form filling work is often involved to maintain the documentation associated with the standards.

# Standards development

---

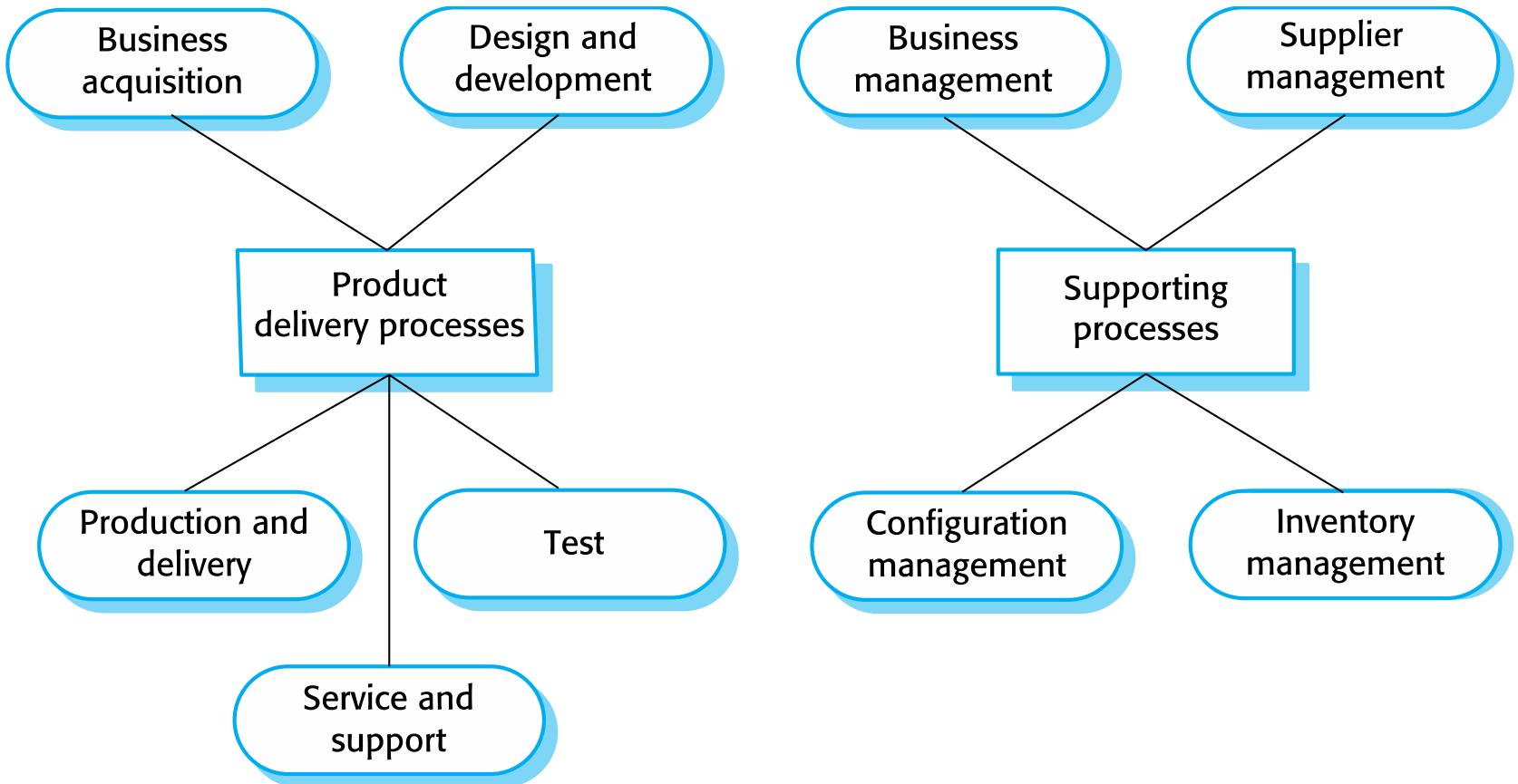
- ✧ Involve practitioners in development. Engineers should understand the rationale underlying a standard.
- ✧ Review standards and their usage regularly. Standards can quickly become outdated and this reduces their credibility amongst practitioners.
- ✧ Detailed standards should have specialized tool support. Excessive clerical work is the most significant complaint against standards.
  - Web-based forms are not good enough.

# ISO 9001 standards framework

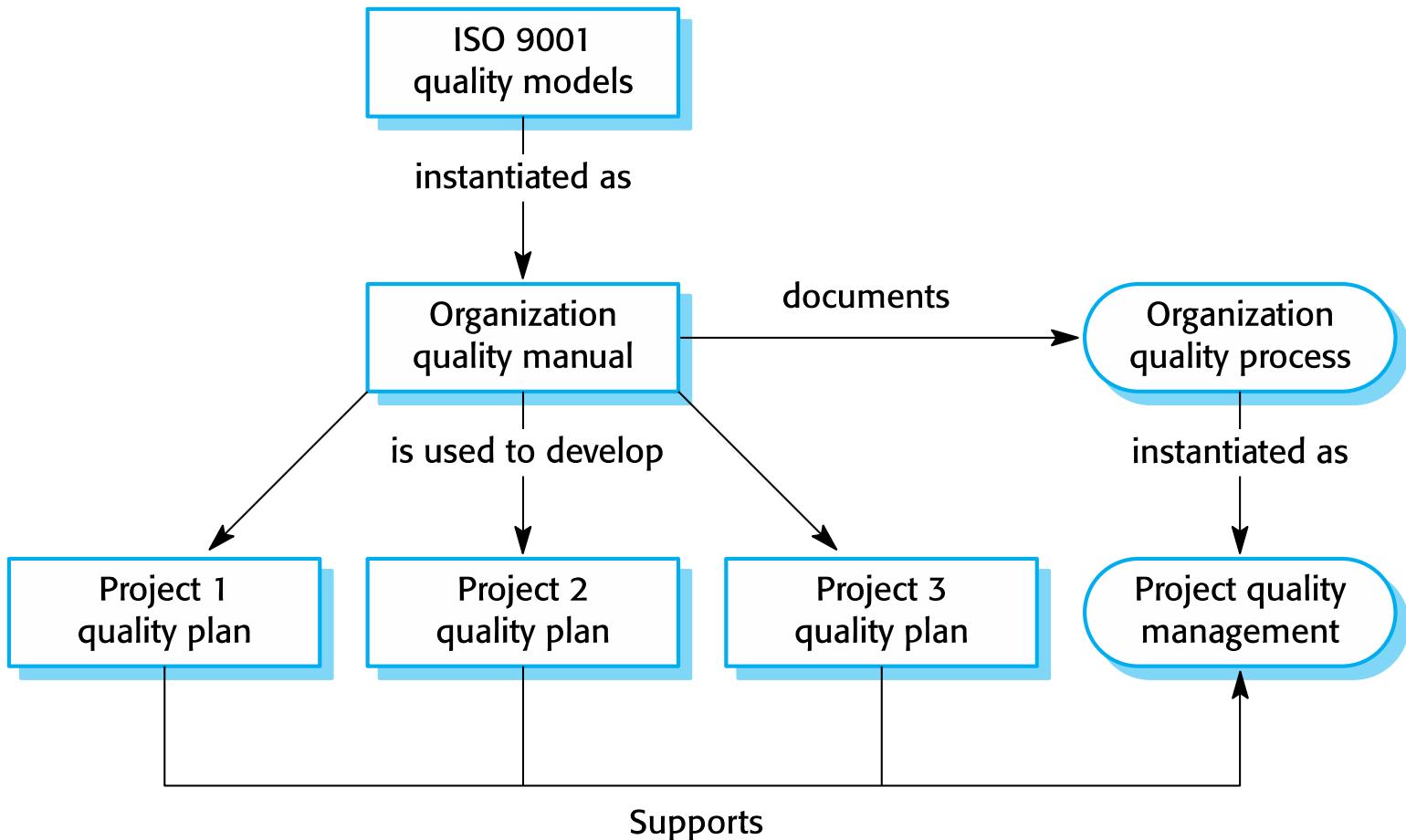
---

- ✧ An international set of standards that can be used as a basis for developing quality management systems.
- ✧ ISO 9001, the most general of these standards, applies to organizations that design, develop and maintain products, including software.
- ✧ The ISO 9001 standard is a framework for developing software standards.
  - It sets out general quality principles, describes quality processes in general and lays out the organizational standards and procedures that should be defined. These should be documented in an organizational quality manual.

# ISO 9001 core processes



# ISO 9001 and quality management



# ISO 9001 certification

---

- ✧ Quality standards and procedures should be documented in an organisational quality manual.
- ✧ An external body may certify that an organisation's quality manual conforms to ISO 9000 standards.
- ✧ Some customers require suppliers to be ISO 9000 certified although the need for flexibility here is increasingly recognised.

# Software quality and ISO9001

---

- ✧ The ISO 9001 certification is inadequate because it defines quality to be the conformance to standards.
- ✧ It takes no account of quality as experienced by users of the software. For example, a company could define test coverage standards specifying that all methods in objects must be called at least once.
- ✧ Unfortunately, this standard can be met by incomplete software testing that does not include tests with different method parameters. So long as the defined testing procedures are followed and test records maintained, the company could be ISO 9001 certified.

# Reviews and inspections

# Reviews and inspections

---

- ✧ A group examines part or all of a process or system and its documentation to find potential problems.
- ✧ Software or documents may be 'signed off' at a review which signifies that progress to the next development stage has been approved by management.
- ✧ There are different types of review with different objectives
  - Inspections for defect removal (product);
  - Reviews for progress assessment (product and process);
  - Quality reviews (product and standards).

# Quality reviews

---

- ✧ A group of people carefully examine part or all of a software system and its associated documentation.
- ✧ Code, designs, specifications, test plans, standards, etc. can all be reviewed.
- ✧ Software or documents may be 'signed off' at a review which signifies that progress to the next development stage has been approved by management.

# Phases in the review process

---

## ✧ Pre-review activities

- Pre-review activities are concerned with review planning and review preparation

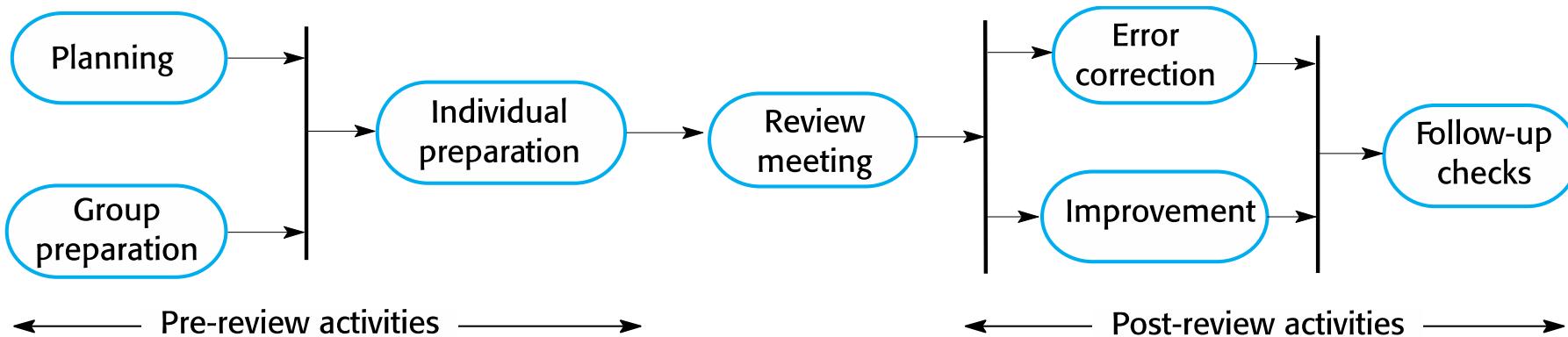
## ✧ The review meeting

- During the review meeting, an author of the document or program being reviewed should ‘walk through’ the document with the review team.

## ✧ Post-review activities

- These address the problems and issues that have been raised during the review meeting.

# The software review process



## Distributed reviews

---

- ✧ The processes suggested for reviews assume that the review team has a face-to-face meeting to discuss the software or documents that they are reviewing.
- ✧ However, project teams are now often distributed, sometimes across countries or continents, so it is impractical for team members to meet face to face.
- ✧ Remote reviewing can be supported using shared documents where each review team member can annotate the document with their comments.

# Program inspections

---

- ✧ These are peer reviews where engineers examine the source of a system with the aim of discovering anomalies and defects.
- ✧ Inspections do not require execution of a system so may be used before implementation.
- ✧ They may be applied to any representation of the system (requirements, design, configuration data, test data, etc.).
- ✧ They have been shown to be an effective technique for discovering program errors.

# Inspection checklists

---

- ✧ Checklist of common errors should be used to drive the inspection.
- ✧ Error checklists are programming language dependent and reflect the characteristic errors that are likely to arise in the language.
- ✧ In general, the 'weaker' the type checking, the larger the checklist.
- ✧ Examples: Initialisation, Constant naming, loop termination, array bounds, etc.

# An inspection checklist (a)

Fault class	Inspection check
Data faults	<ul style="list-style-type: none"><li>• Are all program variables initialized before their values are used?</li><li>• Have all constants been named?</li><li>• Should the upper bound of arrays be equal to the size of the array or Size -1?</li><li>• If character strings are used, is a delimiter explicitly assigned?</li><li>• Is there any possibility of buffer overflow?</li></ul>
Control faults	<ul style="list-style-type: none"><li>• For each conditional statement, is the condition correct?</li><li>• Is each loop certain to terminate?</li><li>• Are compound statements correctly bracketed?</li><li>• In case statements, are all possible cases accounted for?</li><li>• If a break is required after each case in case statements, has it been included?</li></ul>
Input/output faults	<ul style="list-style-type: none"><li>• Are all input variables used?</li><li>• Are all output variables assigned a value before they are output?</li><li>• Can unexpected inputs cause corruption?</li></ul>

# An inspection checklist (b)

Fault class	Inspection check
Interface faults	<ul style="list-style-type: none"><li>• Do all function and method calls have the correct number of parameters?</li><li>• Do formal and actual parameter types match?</li><li>• Are the parameters in the right order?</li><li>• If components access shared memory, do they have the same model of the shared memory structure?</li></ul>
Storage management faults	<ul style="list-style-type: none"><li>• If a linked structure is modified, have all links been correctly reassigned?</li><li>• If dynamic storage is used, has space been allocated correctly?</li><li>• Is space explicitly deallocated after it is no longer required?</li></ul>
Exception management faults	<ul style="list-style-type: none"><li>• Have all possible error conditions been taken into account?</li></ul>

---

# **Quality management and agile development**

# Quality management and agile development

---

- ✧ Quality management in agile development is informal rather than document-based.
- ✧ It relies on establishing a quality culture, where all team members feel responsible for software quality and take actions to ensure that quality is maintained.
- ✧ The agile community is fundamentally opposed to what it sees as the bureaucratic overheads of standards-based approaches and quality processes as embodied in ISO 9001.

# Shared good practice

---

## ✧ *Check before check-in*

- Programmers are responsible for organizing their own code reviews with other team members before the code is checked in to the build system.

## ✧ *Never break the build*

- Team members should not check in code that causes the system to fail. Developers have to test their code changes against the whole system and be confident that these work as expected.

## ✧ *Fix problems when you see them*

- If a programmer discovers problems or obscurities in code developed by someone else, they can fix these directly rather than referring them back to the original developer.

# Reviews and agile methods

---

- ✧ The review process in agile software development is usually informal.
- ✧ In Scrum,, there is a review meeting after each iteration of the software has been completed (a sprint review), where quality issues and problems may be discussed.
- ✧ In Extreme Programming, pair programming ensures that code is constantly being examined and reviewed by another team member.

# Pair programming

---

- ✧ This is an approach where 2 people are responsible for code development and work together to achieve this.
- ✧ Code developed by an individual is therefore constantly being examined and reviewed by another team member.
- ✧ Pair programming leads to a deep knowledge of a program, as both programmers have to understand the program in detail to continue development.
- ✧ This depth of knowledge is difficult to achieve in inspection processes and pair programming can find bugs that would not be discovered in formal inspections.

# Pair programming weaknesses

---

## ✧ *Mutual misunderstandings*

- Both members of a pair may make the same mistake in understanding the system requirements. Discussions may reinforce these errors.

## ✧ *Pair reputation*

- Pairs may be reluctant to look for errors because they do not want to slow down the progress of the project.

## ✧ *Working relationships*

- The pair's ability to discover defects is likely to be compromised by their close working relationship that often leads to reluctance to criticize work partners.

# Agile QM and large systems

---

- ✧ When a large system is being developed for an external customer, agile approaches to quality management with minimal documentation may be impractical.
  - If the customer is a large company, it may have its own quality management processes and may expect the software development company to report on progress in a way that is compatible with them.
  - Where there are several geographically distributed teams involved in development, perhaps from different companies, then informal communications may be impractical.
  - For long-lifetime systems, the team involved in development will changeWithout documentation, new team members may find it impossible to understand development.

# Software measurement

# Software measurement

---

- ✧ Software measurement is concerned with deriving a numeric value for an attribute of a software product or process.
- ✧ This allows for objective comparisons between techniques and processes.
- ✧ Although some companies have introduced measurement programmes, most organisations still don't make systematic use of software measurement.
- ✧ There are few established standards in this area.

# Software metric

---

- ✧ Any type of measurement which relates to a software system, process or related documentation
  - Lines of code in a program, the Fog index, number of person-days required to develop a component.
- ✧ Allow the software and the software process to be quantified.
- ✧ May be used to predict product attributes or to control the software process.
- ✧ Product metrics can be used for general predictions or to identify anomalous components.

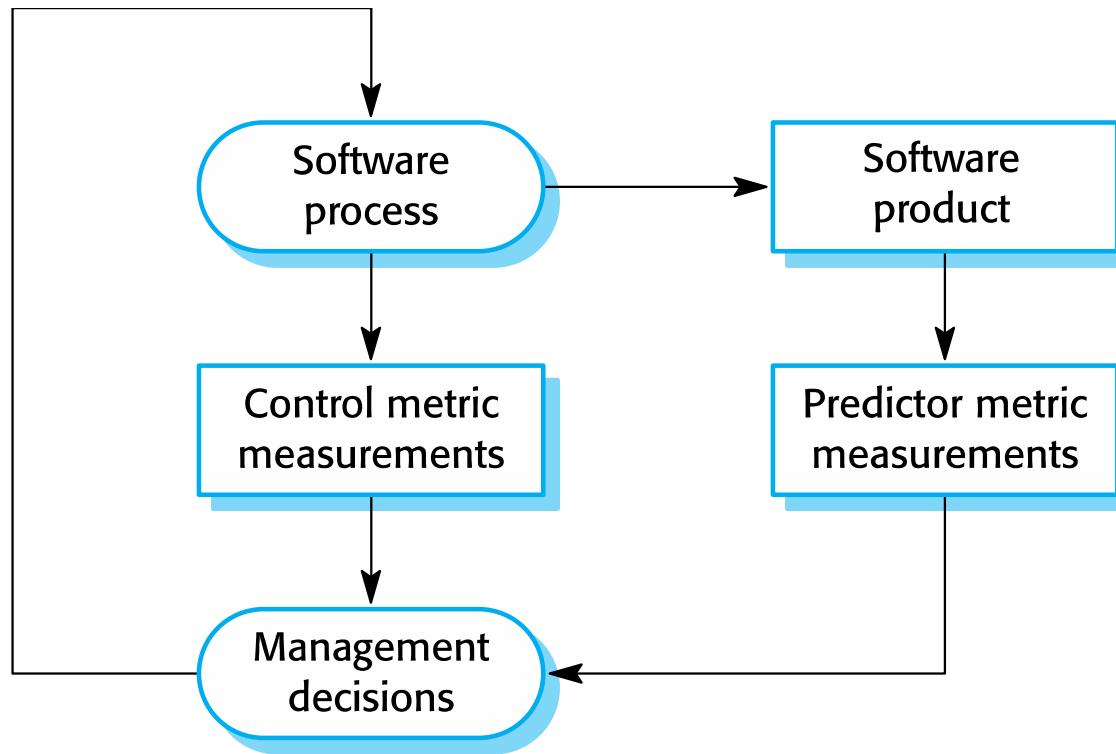
# Types of process metric

---

- ✧ *The time taken for a particular process to be completed*
  - This can be the total time devoted to the process, calendar time, the time spent on the process by particular engineers, and so on.
- ✧ *The resources required for a particular process*
  - Resources might include total effort in person-days, travel costs or computer resources.
- ✧ *The number of occurrences of a particular event*
  - Examples of events that might be monitored include the number of defects discovered during code inspection, the number of requirements changes requested, the number of bug reports in a delivered system and the average number of lines of code modified in response to a requirements change.

# Predictor and control measurements

---



# Use of measurements

---

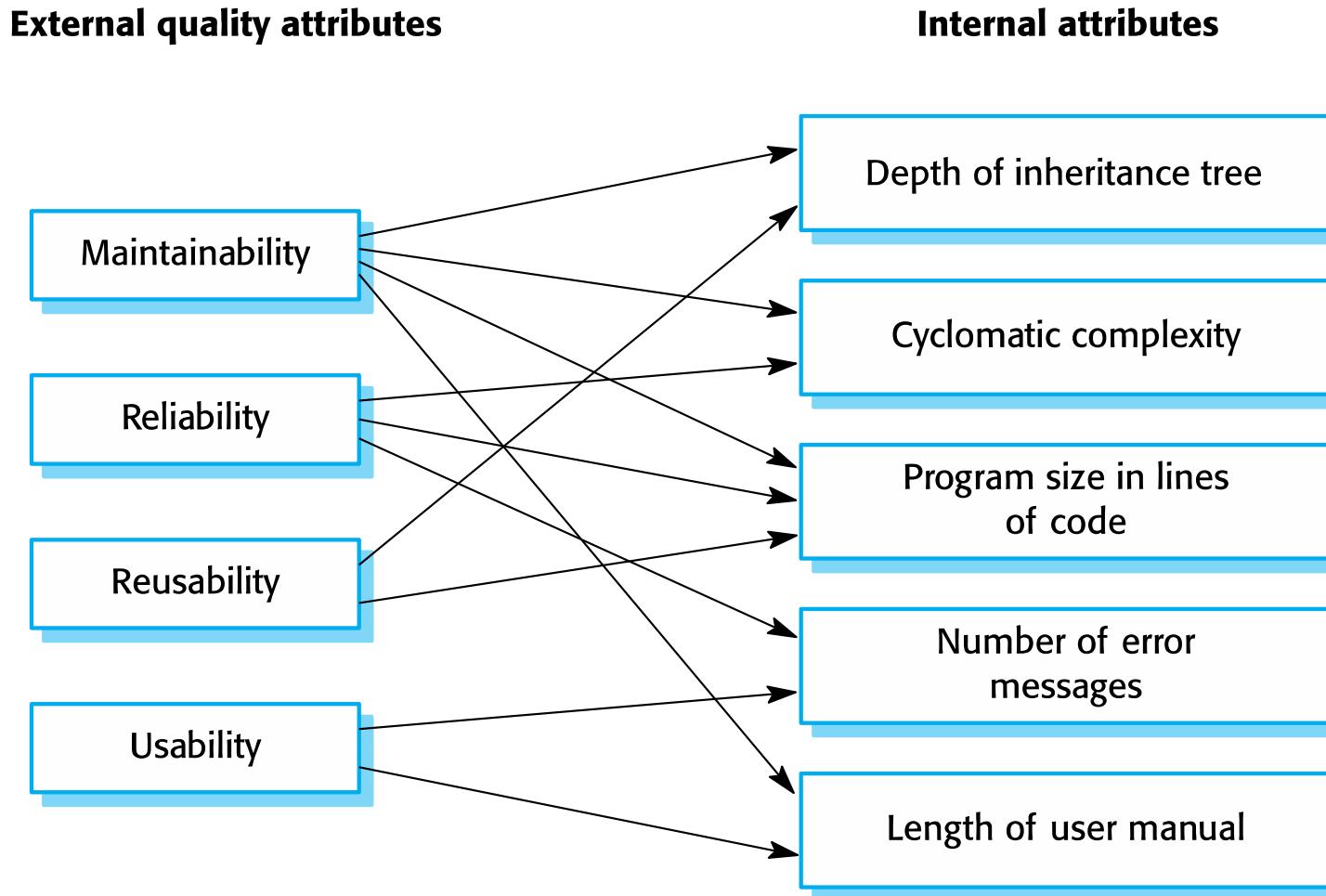
- ✧ To assign a value to system quality attributes
  - By measuring the characteristics of system components, such as their cyclomatic complexity, and then aggregating these measurements, you can assess system quality attributes, such as maintainability.
- ✧ To identify the system components whose quality is sub-standard
  - Measurements can identify individual components with characteristics that deviate from the norm. For example, you can measure components to discover those with the highest complexity. These are most likely to contain bugs because the complexity makes them harder to understand.

# Metrics assumptions

---

- ✧ A software property can be measured accurately.
- ✧ The relationship exists between what we can measure and what we want to know. We can only measure internal attributes but are often more interested in external software attributes.
- ✧ This relationship has been formalised and validated.
- ✧ It may be difficult to relate what can be measured to desirable external quality attributes.

# Relationships between internal and external software



# Problems with measurement in industry

---

- ✧ It is impossible to quantify the return on investment of introducing an organizational metrics program.
- ✧ There are no standards for software metrics or standardized processes for measurement and analysis.
- ✧ In many companies, software processes are not standardized and are poorly defined and controlled.
- ✧ Most work on software measurement has focused on code-based metrics and plan-driven development processes. However, more and more software is now developed by configuring ERP systems or COTS.
- ✧ Introducing measurement adds additional overhead to processes.

# Empirical software engineering

---

- ✧ Software measurement and metrics are the basis of empirical software engineering.
- ✧ This is a research area in which experiments on software systems and the collection of data about real projects has been used to form and validate hypotheses about software engineering methods and techniques.
- ✧ Research on empirical software engineering, this has not had a significant impact on software engineering practice.
- ✧ It is difficult to relate generic research to a project that is different from the research study.

# Product metrics

---

- ✧ A quality metric should be a predictor of product quality.
- ✧ Classes of product metric
  - Dynamic metrics which are collected by measurements made of a program in execution;
  - Static metrics which are collected by measurements made of the system representations;
  - Dynamic metrics help assess efficiency and reliability
  - Static metrics help assess complexity, understandability and maintainability.

# Dynamic and static metrics

---

- ✧ Dynamic metrics are closely related to software quality attributes
  - It is relatively easy to measure the response time of a system (performance attribute) or the number of failures (reliability attribute).
- ✧ Static metrics have an indirect relationship with quality attributes
  - You need to try and derive a relationship between these metrics and properties such as complexity, understandability and maintainability.

# Static software product metrics

---

Software metric	Description
Fan-in/Fan-out	Fan-in is a measure of the number of functions or methods that call another function or method (say X). Fan-out is the number of functions that are called by function X. A high value for fan-in means that X is tightly coupled to the rest of the design and changes to X will have extensive knock-on effects. A high value for fan-out suggests that the overall complexity of X may be high because of the complexity of the control logic needed to coordinate the called components.
Length of code	This is a measure of the size of a program. Generally, the larger the size of the code of a component, the more complex and error-prone that component is likely to be. Length of code has been shown to be one of the most reliable metrics for predicting error-proneness in components.

# Static software product metrics

Software metric	Description
Cyclomatic complexity	This is a measure of the control complexity of a program. This control complexity may be related to program understandability. I discuss cyclomatic complexity in Chapter 8.
Length of identifiers	This is a measure of the average length of identifiers (names for variables, classes, methods, etc.) in a program. The longer the identifiers, the more likely they are to be meaningful and hence the more understandable the program.
Depth of conditional nesting	This is a measure of the depth of nesting of if-statements in a program. Deeply nested if-statements are hard to understand and potentially error-prone.
Fog index	This is a measure of the average length of words and sentences in documents. The higher the value of a document's Fog index, the more difficult the document is to understand.

# The CK object-oriented metrics suite

Object-oriented metric	Description
Weighted methods per class (WMC)	This is the number of methods in each class, weighted by the complexity of each method. Therefore, a simple method may have a complexity of 1, and a large and complex method a much higher value. The larger the value for this metric, the more complex the object class. Complex objects are more likely to be difficult to understand. They may not be logically cohesive, so cannot be reused effectively as superclasses in an inheritance tree.
Depth of inheritance tree (DIT)	This represents the number of discrete levels in the inheritance tree where subclasses inherit attributes and operations (methods) from superclasses. The deeper the inheritance tree, the more complex the design. Many object classes may have to be understood to understand the object classes at the leaves of the tree.
Number of children (NOC)	This is a measure of the number of immediate subclasses in a class. It measures the breadth of a class hierarchy, whereas DIT measures its depth. A high value for NOC may indicate greater reuse. It may mean that more effort should be made in validating base classes because of the number of subclasses that depend on them.

# The CK object-oriented metrics suite

Object-oriented metric	Description
Coupling between object classes (CBO)	Classes are coupled when methods in one class use methods or instance variables defined in a different class. CBO is a measure of how much coupling exists. A high value for CBO means that classes are highly dependent, and therefore it is more likely that changing one class will affect other classes in the program.
Response for a class (RFC)	RFC is a measure of the number of methods that could potentially be executed in response to a message received by an object of that class. Again, RFC is related to complexity. The higher the value for RFC, the more complex a class and hence the more likely it is that it will include errors.
Lack of cohesion in methods (LCOM)	LCOM is calculated by considering pairs of methods in a class. LCOM is the difference between the number of method pairs without shared attributes and the number of method pairs with shared attributes. The value of this metric has been widely debated and it exists in several variations. It is not clear if it really adds any additional, useful information over and above that provided by other metrics.

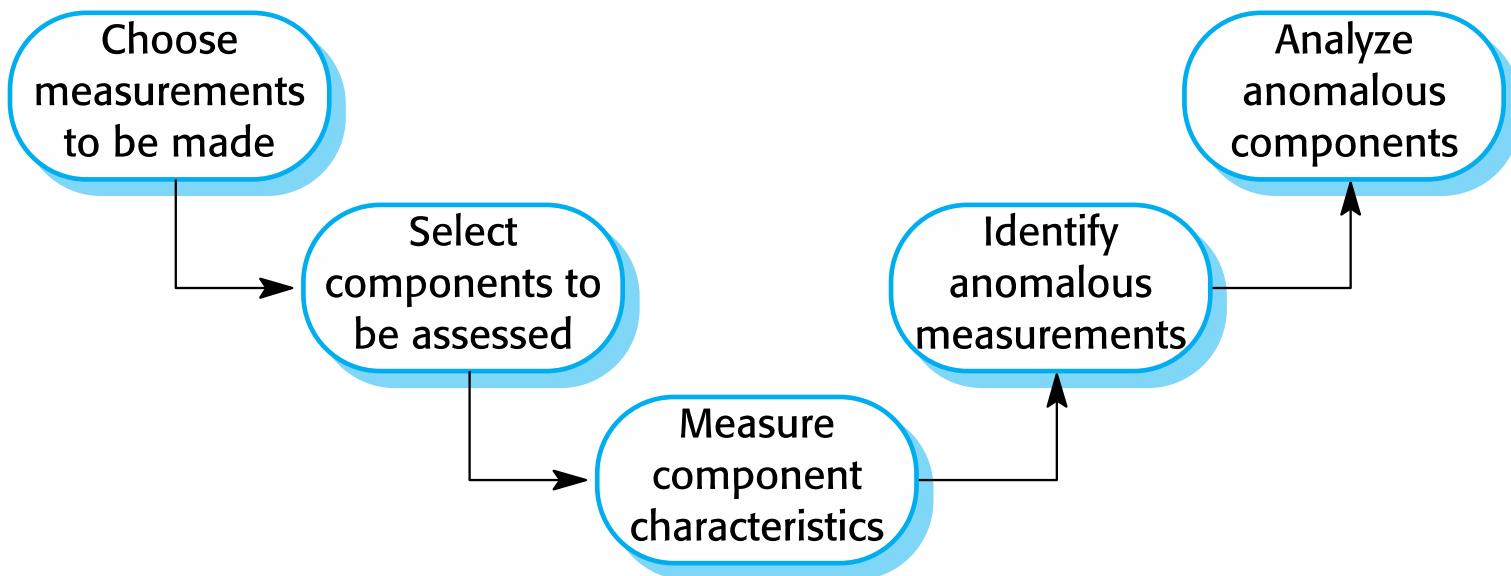
# Software component analysis

---

- ✧ System component can be analyzed separately using a range of metrics.
- ✧ The values of these metrics may then compared for different components and, perhaps, with historical measurement data collected on previous projects.
- ✧ Anomalous measurements, which deviate significantly from the norm, may imply that there are problems with the quality of these components.

# The process of product measurement

---



# Measurement ambiguity

---

- ✧ When you collect quantitative data about software and software processes, you have to analyze that data to understand its meaning.
- ✧ It is easy to misinterpret data and to make inferences that are incorrect.
- ✧ You cannot simply look at the data on its own. You must also consider the context where the data is collected.

# Measurement surprises

---

- ✧ Reducing the number of faults in a program leads to an increased number of help desk calls
  - The program is now thought of as more reliable and so has a wider more diverse market. The percentage of users who call the help desk may have decreased but the total may increase;
  - A more reliable system is used in a different way from a system where users work around the faults. This leads to more help desk calls.

## Software context

---

- ✧ Processes and products that are being measured are not insulated from their environment.
- ✧ The business environment is constantly changing and it is impossible to avoid changes to work practice just because they may make comparisons of data invalid.
- ✧ Data about human activities cannot always be taken at face value. The reasons why a measured value changes are often ambiguous. These reasons must be investigated in detail before drawing conclusions from any measurements that have been made.

# Software analytics

---

- ✧ *Software analytics is analytics on software data for managers and software engineers with the aim of empowering software development individuals and teams to gain and share insight from their data to make better decisions.*

# Software analytics enablers

---

- ✧ The automated collection of user data by software product companies when their product is used.
  - If the software fails, information about the failure and the state of the system can be sent over the Internet from the user's computer to servers run by the product developer.
- ✧ The use of open source software available on platforms such as Sourceforge and GitHub and open source repositories of software engineering data.
  - The source code of open source software is available for automated analysis and this can sometimes be linked with data in the open source repository.

## Analytics tool use

---

- ✧ Tools should be easy to use as managers are unlikely to have experience with analysis.
- ✧ •Tools should run quickly and produce concise outputs rather than large volumes of information.
- ✧ •Tools should make many measurements using as many parameters as possible. It is impossible to predict in advance what insights might emerge.
- ✧ •Tools should be interactive and allow managers and developers to explore the analyses.

# Status of software analytics

---

- ✧ Software analytics is still immature and it is too early to say what effect it will have.
- ✧ Not only are there general problems of ‘big data’ processing, our knowledge depends on collected data from large companies.
  - This is primarily from software products and it is unclear if the tools and techniques that are appropriate for products can also be used with custom software.
- ✧ Small companies are unlikely to invest in the data collection systems that are required for automated analysis so may not be able to use software analytics.

## Key points

---

- ✧ Software quality management is concerned with ensuring that software has a low number of defects and that it reaches the required standards of maintainability, reliability, portability etc. Software standards are important for quality assurance as they represent an identification of ‘best practice’. When developing software, standards provide a solid foundation for building good quality software.
- ✧ Reviews of the software process deliverables involve a team of people who check that quality standards are being followed. Reviews are the most widely used technique for assessing quality.

# Key points

---

- ✧ In a program inspection or peer review, a small team systematically checks the code. They read the code in detail and look for possible errors and omissions. The problems detected are discussed at a code review meeting.
- ✧ Agile quality management relies on establishing a quality culture where the development team works together to improve software quality.
- ✧ Software measurement can be used to gather quantitative data about software and the software process.

## Key points

---

- ✧ You may be able to use the values of the software metrics that are collected to make inferences about product and process quality.
- ✧ Product quality metrics are particularly useful for highlighting anomalous components that may have quality problems. These components should then be analyzed in more detail.
- ✧ Software analytics is the automated analysis of large volumes of software product and process data to discover relationships that may provide insights for project managers and developers.

# Chapter 25 – Configuration Management

# Topics covered

---

- ✧ Version management
- ✧ System building
- ✧ Change management
- ✧ Release management

# Configuration management

---

- ✧ Software systems are constantly changing during development and use.
- ✧ Configuration management (CM) is concerned with the policies, processes and tools for managing changing software systems.
- ✧ You need CM because it is easy to lose track of what changes and component versions have been incorporated into each system version.
- ✧ CM is essential for team projects to control changes made by different developers

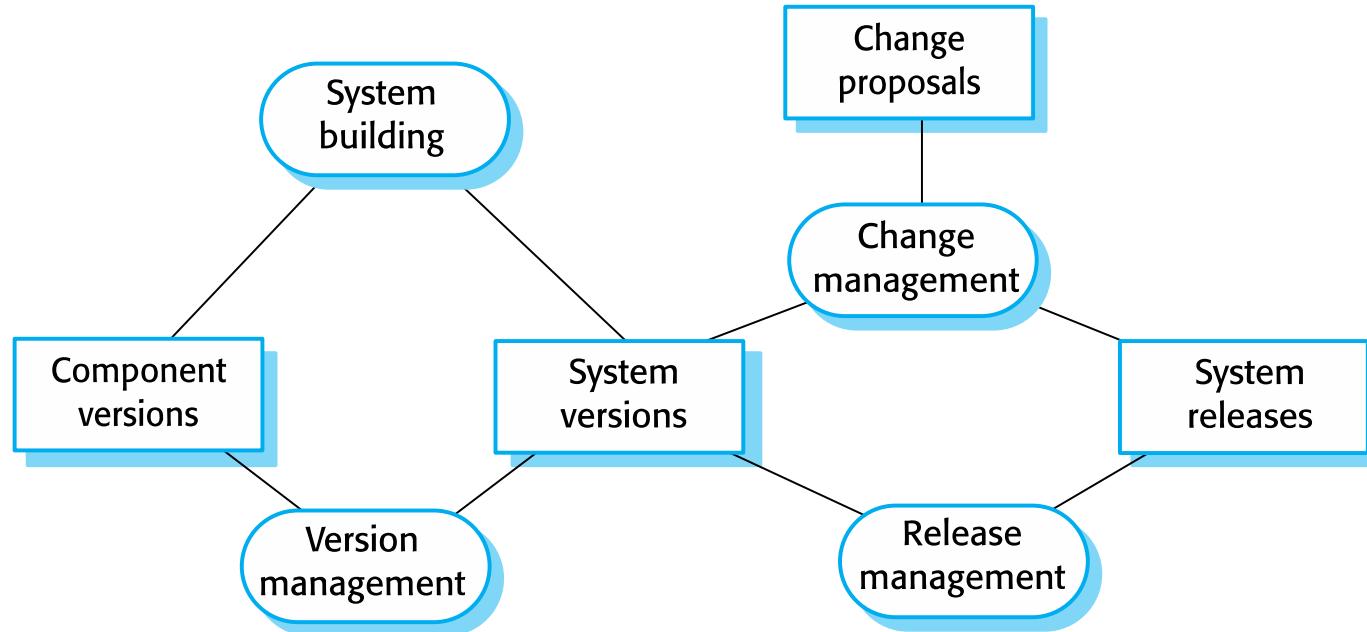
# CM activities

---

- ✧ Version management
  - Keeping track of the multiple versions of system components and ensuring that changes made to components by different developers do not interfere with each other.
- ✧ System building
  - The process of assembling program components, data and libraries, then compiling these to create an executable system.
- ✧ Change management
  - Keeping track of requests for changes to the software from customers and developers, working out the costs and impact of changes, and deciding the changes should be implemented.
- ✧ Release management
  - Preparing software for external release and keeping track of the system versions that have been released for customer use.

# Configuration management activities

---



# Agile development and CM

---

- ✧ Agile development, where components and systems are changed several times per day, is impossible without using CM tools.
- ✧ The definitive versions of components are held in a shared project repository and developers copy these into their own workspace.
- ✧ They make changes to the code then use system building tools to create a new system on their own computer for testing. Once they are happy with the changes made, they return the modified components to the project repository.

# Development phases

---

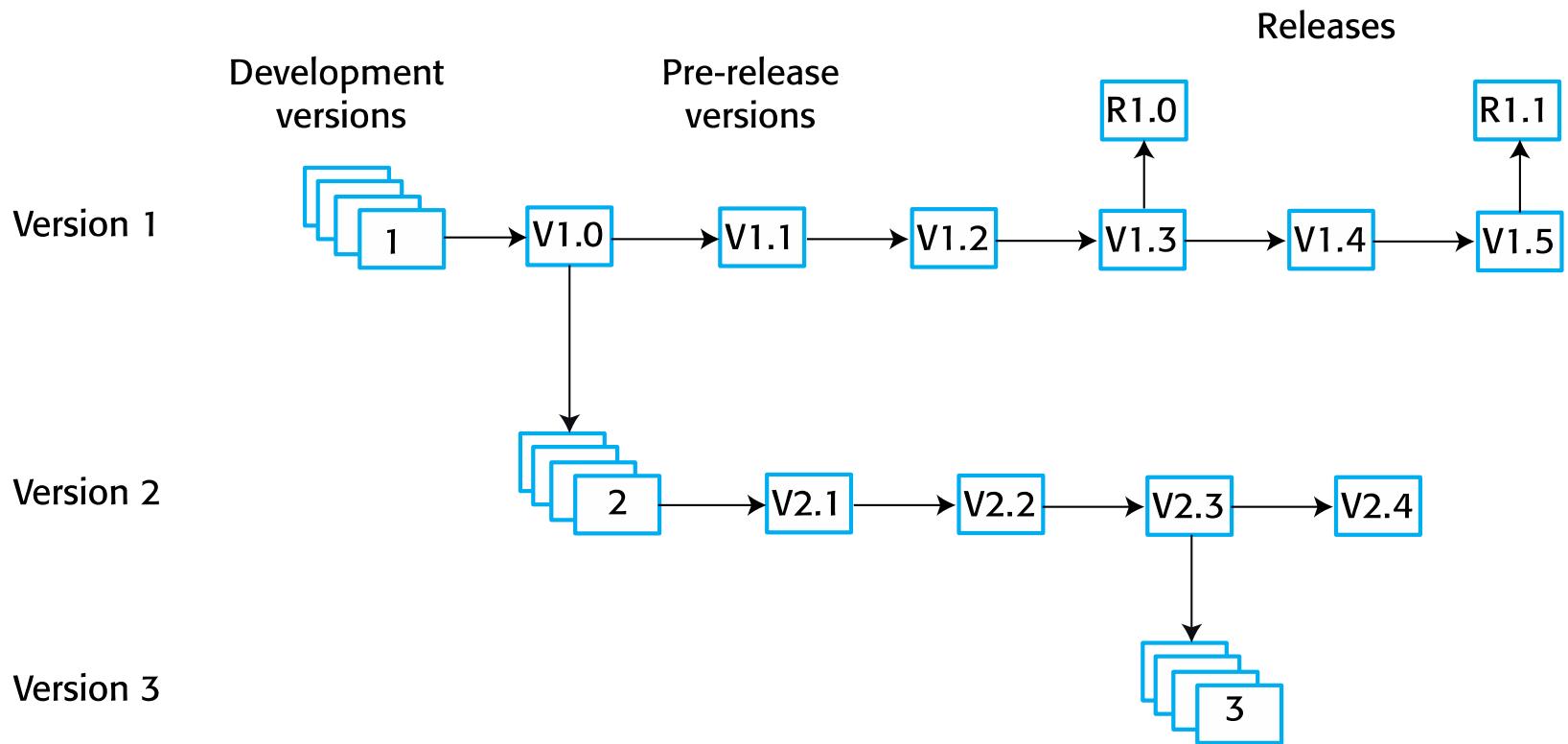
- ✧ A development phase where the development team is responsible for managing the software configuration and new functionality is being added to the software.
- ✧ A system testing phase where a version of the system is released internally for testing.
  - No new system functionality is added. Changes made are bug fixes, performance improvements and security vulnerability repairs.
- ✧ A release phase where the software is released to customers for use.
  - New versions of the released system are developed to repair bugs and vulnerabilities and to include new features.

# Multi-version systems

---

- ✧ For large systems, there is never just one ‘working’ version of a system.
- ✧ There are always several versions of the system at different stages of development.
- ✧ There may be several teams involved in the development of different system versions.

# Multi-version system development



# CM terminology

Term	Explanation
Baseline	A baseline is a collection of component versions that make up a system. Baselines are controlled, which means that the versions of the components making up the system cannot be changed. This means that it is always possible to recreate a baseline from its constituent components.
Branching	The creation of a new codeline from a version in an existing codeline. The new codeline and the existing codeline may then develop independently.
Codeline	A codeline is a set of versions of a software component and other configuration items on which that component depends.
Configuration (version) control	The process of ensuring that versions of systems and components are recorded and maintained so that changes are managed and all versions of components are identified and stored for the lifetime of the system.
Configuration item or software configuration item (SCI)	Anything associated with a software project (design, code, test data, document, etc.) that has been placed under configuration control. There are often different versions of a configuration item. Configuration items have a unique name.
Mainline	A sequence of baselines representing different versions of a system.

# CM terminology

---

Term	Explanation
Merging	The creation of a new version of a software component by merging separate versions in different codelines. These codelines may have been created by a previous branch of one of the codelines involved.
Release	A version of a system that has been released to customers (or other users in an organization) for use.
Repository	A shared database of versions of software components and meta-information about changes to these components.
System building	The creation of an executable system version by compiling and linking the appropriate versions of the components and libraries making up the system.
Version	An instance of a configuration item that differs, in some way, from other instances of that item. Versions always have a unique identifier.
Workspace	A private work area where software can be modified without affecting other developers who may be using or modifying that software.

---

# **Version management**

# Version management

---

- ✧ Version management (VM) is the process of keeping track of different versions of software components or configuration items and the systems in which these components are used.
- ✧ It also involves ensuring that changes made by different developers to these versions do not interfere with each other.
- ✧ Therefore version management can be thought of as the process of managing codelines and baselines.

# Codelines and baselines

---

- ✧ A codeline is a sequence of versions of source code with later versions in the sequence derived from earlier versions.
- ✧ Codelines normally apply to components of systems so that there are different versions of each component.
- ✧ A baseline is a definition of a specific system.
- ✧ The baseline therefore specifies the component versions that are included in the system plus a specification of the libraries used, configuration files, etc.

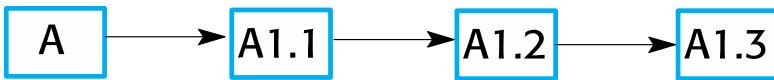
# Baselines

---

- ✧ Baselines may be specified using a configuration language, which allows you to define what components are included in a version of a particular system.
- ✧ Baselines are important because you often have to recreate a specific version of a complete system.
  - For example, a product line may be instantiated so that there are individual system versions for different customers. You may have to recreate the version delivered to a specific customer if, for example, that customer reports bugs in their system that have to be repaired.

# Codelines and baselines

Codeline (A)



Codeline (B)



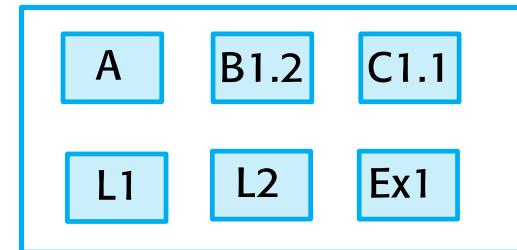
Codeline (C)



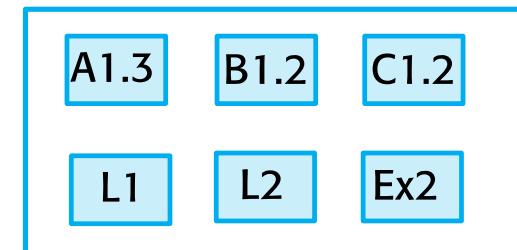
Libraries and external components



Baseline - V1



Baseline - V2



Mainline

# Version control systems

---

- ✧ Version control (VC) systems identify, store and control access to the different versions of components. There are two types of modern version control system
  - Centralized systems, where there is a single master repository that maintains all versions of the software components that are being developed. Subversion is a widely used example of a centralized VC system.
  - Distributed systems, where multiple versions of the component repository exist at the same time. Git is a widely-used example of a distributed VC system.

# Key features of version control systems

---

- ✧ Version and release identification
- ✧ Change history recording
- ✧ Support for independent development
- ✧ Project support
- ✧ Storage management

# Public repository and private workspaces

---

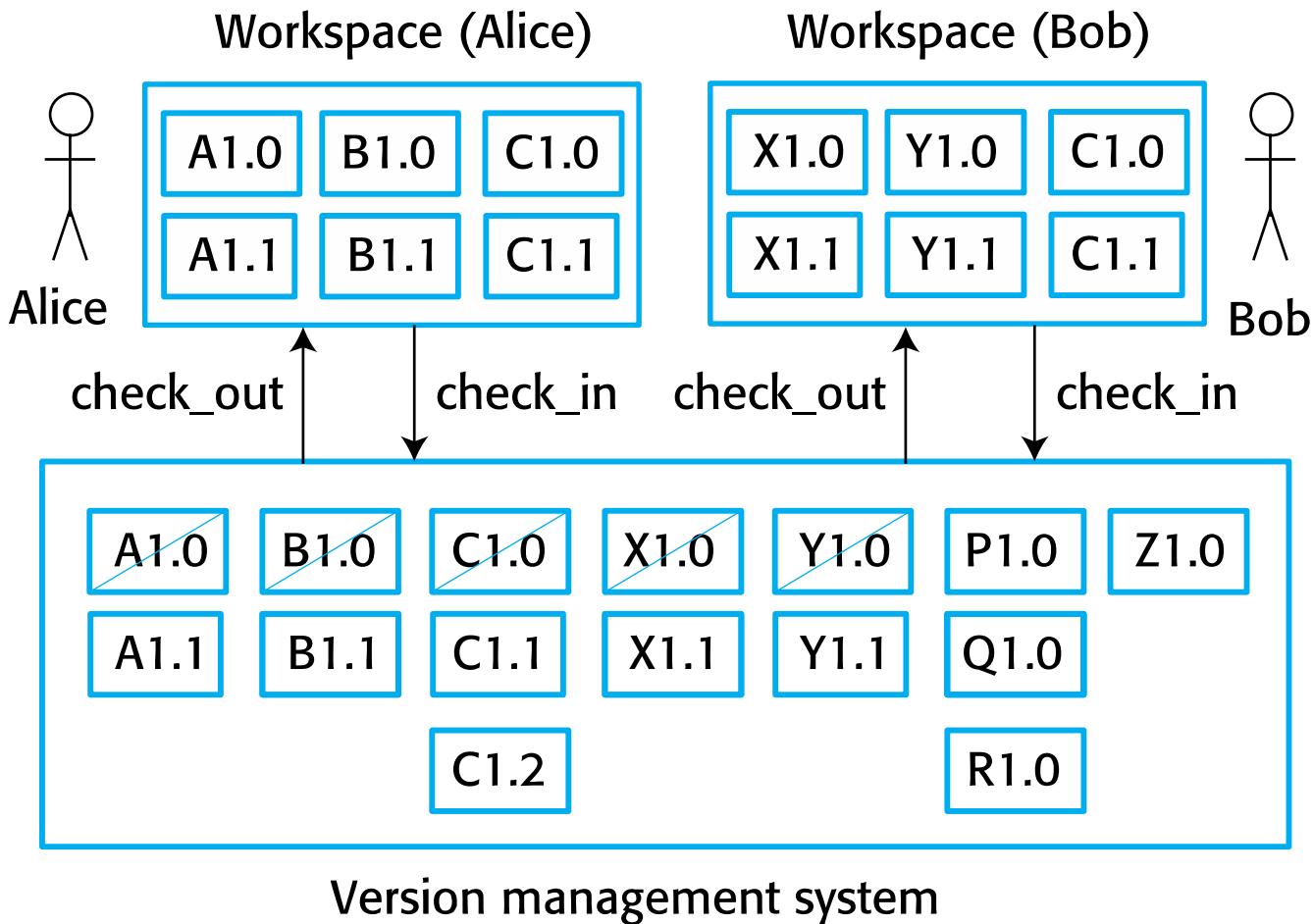
- ✧ To support independent development without interference, version control systems use the concept of a project repository and a private workspace.
- ✧ The project repository maintains the ‘master’ version of all components. It is used to create baselines for system building.
- ✧ When modifying components, developers copy (check-out) these from the repository into their workspace and work on these copies.
- ✧ When they have finished their changes, the changed components are returned (checked-in) to the repository.

# Centralized version control

---

- ✧ Developers check out components or directories of components from the project repository into their private workspace and work on these copies in their private workspace.
- ✧ When their changes are complete, they check-in the components back to the repository.
- ✧ If several people are working on a component at the same time, each check it out from the repository. If a component has been checked out, the VC system warns other users wanting to check out that component that it has been checked out by someone else.

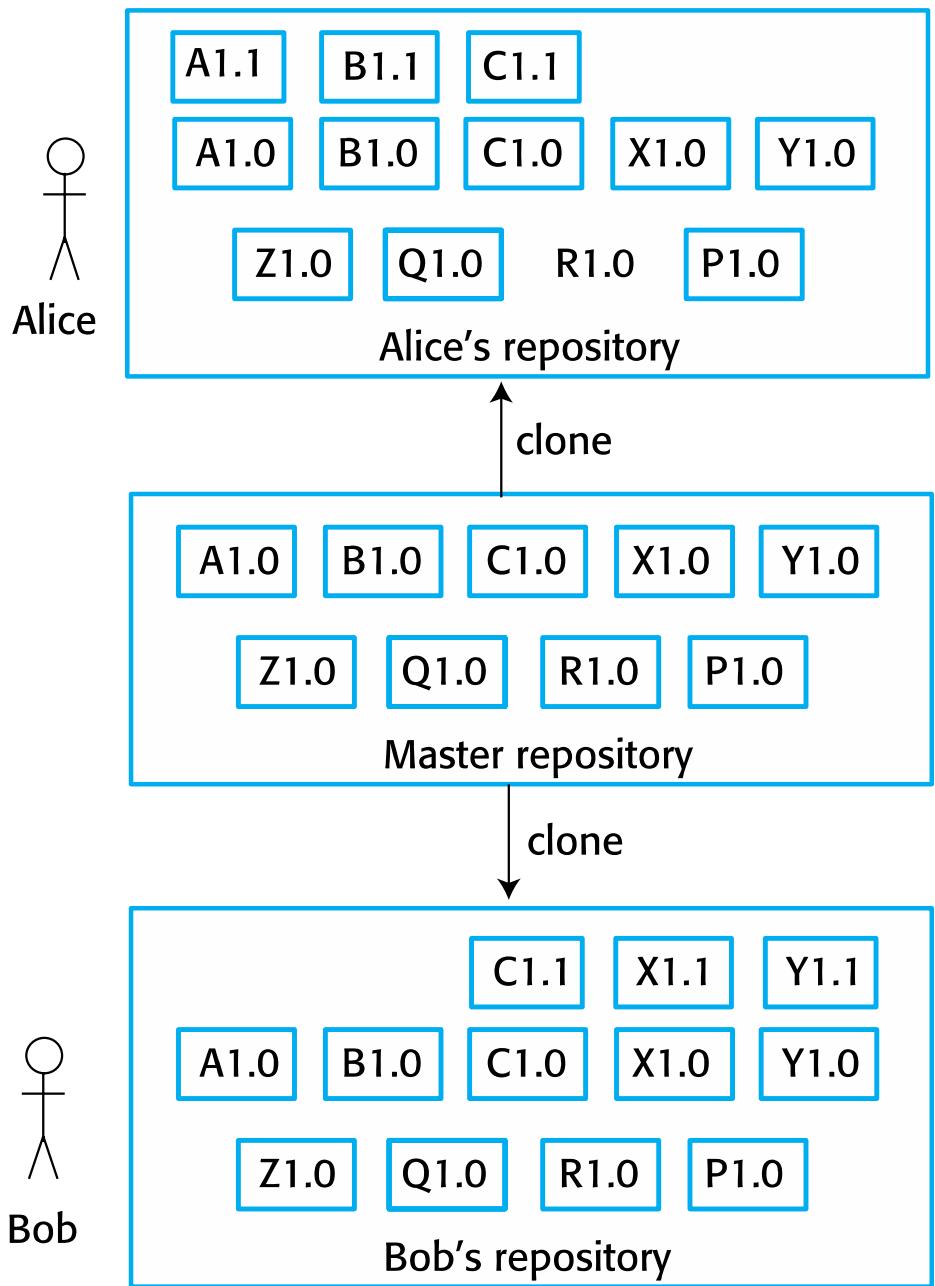
# Repository Check-in/Check-out



# Distributed version control

---

- ✧ A ‘master’ repository is created on a server that maintains the code produced by the development team.
- ✧ Instead of checking out the files that they need, a developer creates a clone of the project repository that is downloaded and installed on their computer.
- ✧ Developers work on the files required and maintain the new versions on their private repository on their own computer.
- ✧ When changes are done, they ‘commit’ these changes and update their private server repository. They may then ‘push’ these changes to the project repository.



## Repository cloning

# Benefits of distributed version control

---

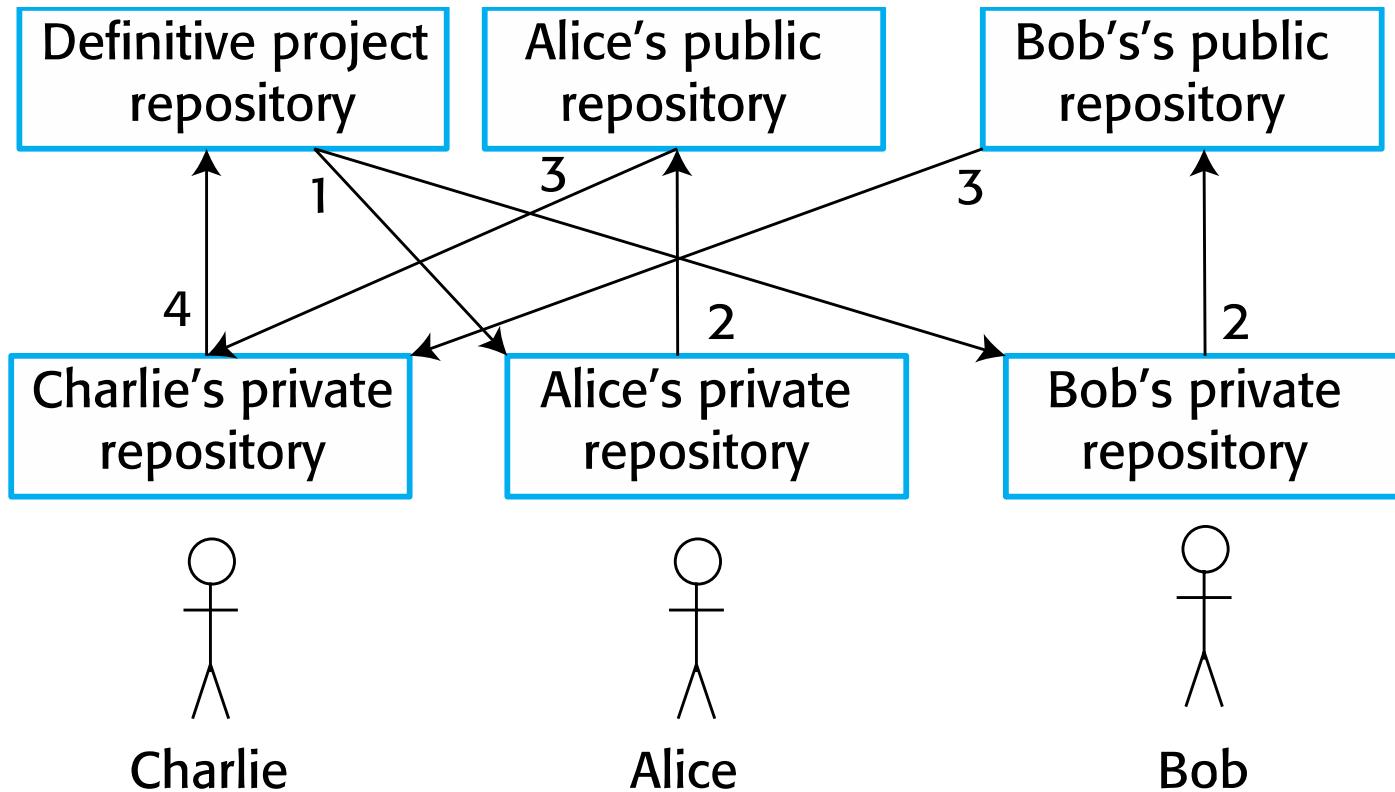
- ✧ It provides a backup mechanism for the repository.
  - If the repository is corrupted, work can continue and the project repository can be restored from local copies.
- ✧ It allows for off-line working so that developers can commit changes if they do not have a network connection.
- ✧ Project support is the default way of working.
  - Developers can compile and test the entire system on their local machines and test the changes that they have made.

# Open source development

---

- ✧ Distributed version control is essential for open source development.
  - Several people may be working simultaneously on the same system without any central coordination.
- ✧ As well as a private repository on their own computer, developers also maintain a public server repository to which they push new versions of components that they have changed.
  - It is then up to the open-source system ‘manager’ to decide when to pull these changes into the definitive system.

# Open-source development

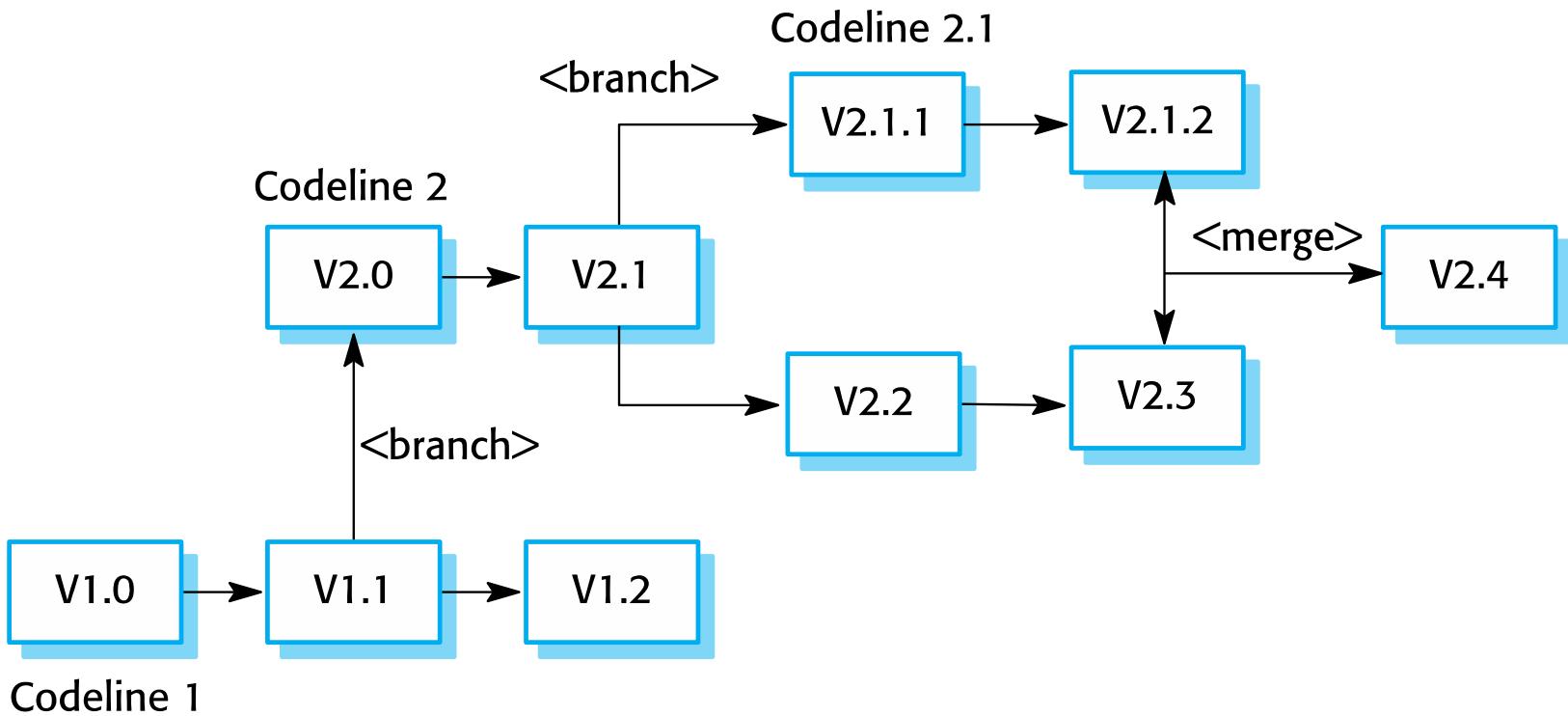


# Branching and merging

---

- ✧ Rather than a linear sequence of versions that reflect changes to the component over time, there may be several independent sequences.
  - This is normal in system development, where different developers work independently on different versions of the source code and so change it in different ways.
- ✧ At some stage, it may be necessary to merge codeline branches to create a new version of a component that includes all changes that have been made.
  - If the changes made involve different parts of the code, the component versions may be merged automatically by combining the deltas that apply to the code.

# Branching and merging

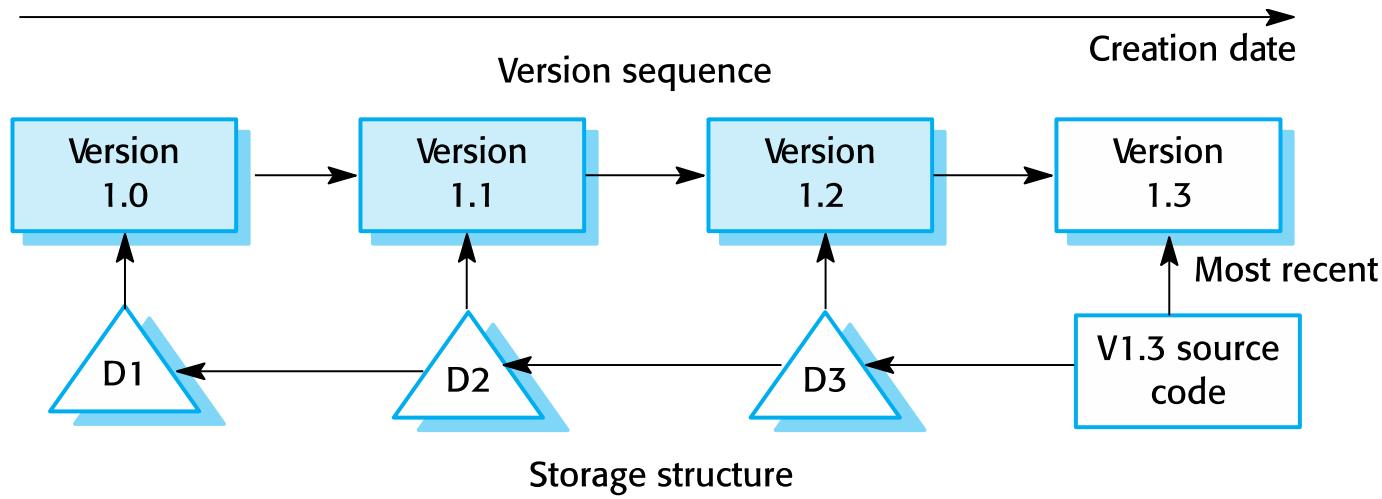


# Storage management

---

- ✧ When version control systems were first developed, storage management was one of their most important functions.
- ✧ Disk space was expensive and it was important to minimize the disk space used by the different copies of components.
- ✧ Instead of keeping a complete copy of each version, the system stores a list of differences (deltas) between one version and another.
  - By applying these to a master version (usually the most recent version), a target version can be recreated.

# Storage management using deltas



# Storage management in Git

---

- ✧ As disk storage is now relatively cheap, Git uses an alternative, faster approach.
- ✧ Git does not use deltas but applies a standard compression algorithm to stored files and their associated meta-information.
- ✧ It does not store duplicate copies of files. Retrieving a file simply involves decompressing it, with no need to apply a chain of operations.
- ✧ Git also uses the notion of packfiles where several smaller files are combined into an indexed single file.

---

# **System building**

# System building

---

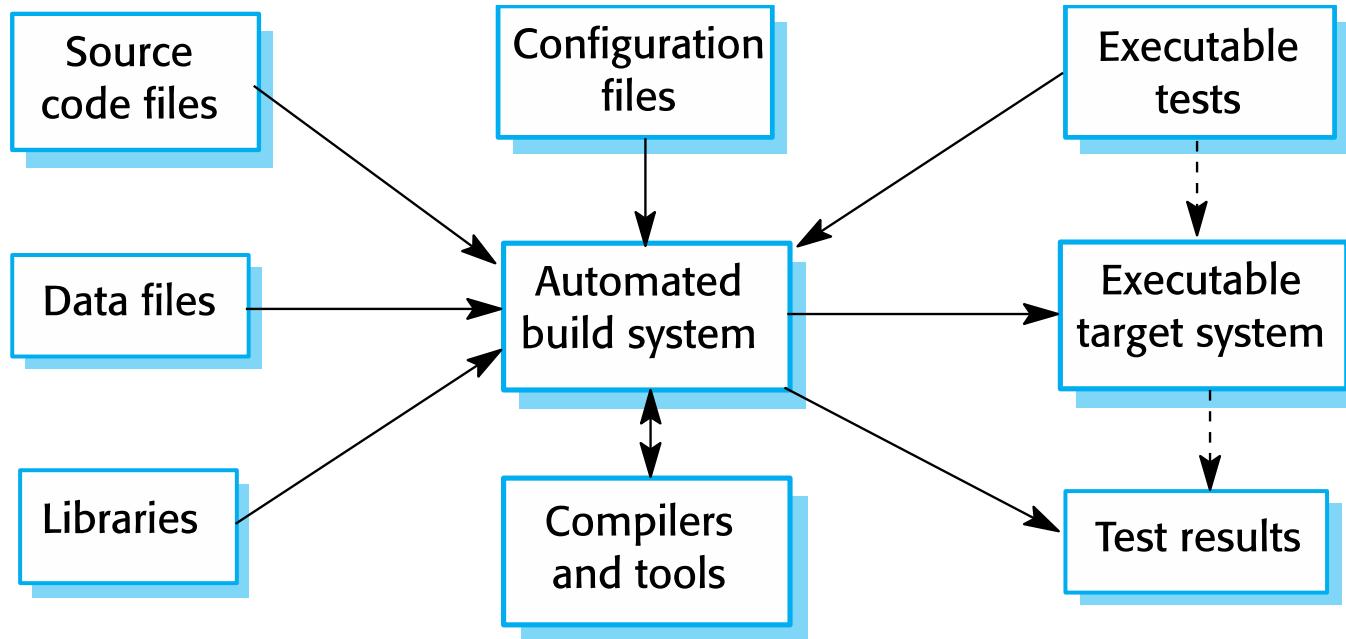
- ✧ System building is the process of creating a complete, executable system by compiling and linking the system components, external libraries, configuration files, etc.
- ✧ System building tools and version management tools must communicate as the build process involves checking out component versions from the repository managed by the version management system.
- ✧ The configuration description used to identify a baseline is also used by the system building tool.

# Build platforms

---

- ✧ The development system, which includes development tools such as compilers, source code editors, etc.
  - Developers check out code from the version management system into a private workspace before making changes to the system.
- ✧ The build server, which is used to build definitive, executable versions of the system.
  - Developers check-in code to the version management system before it is built. The system build may rely on external libraries that are not included in the version management system.
- ✧ The target environment, which is the platform on which the system executes.

# System building



# Build system functionality

---

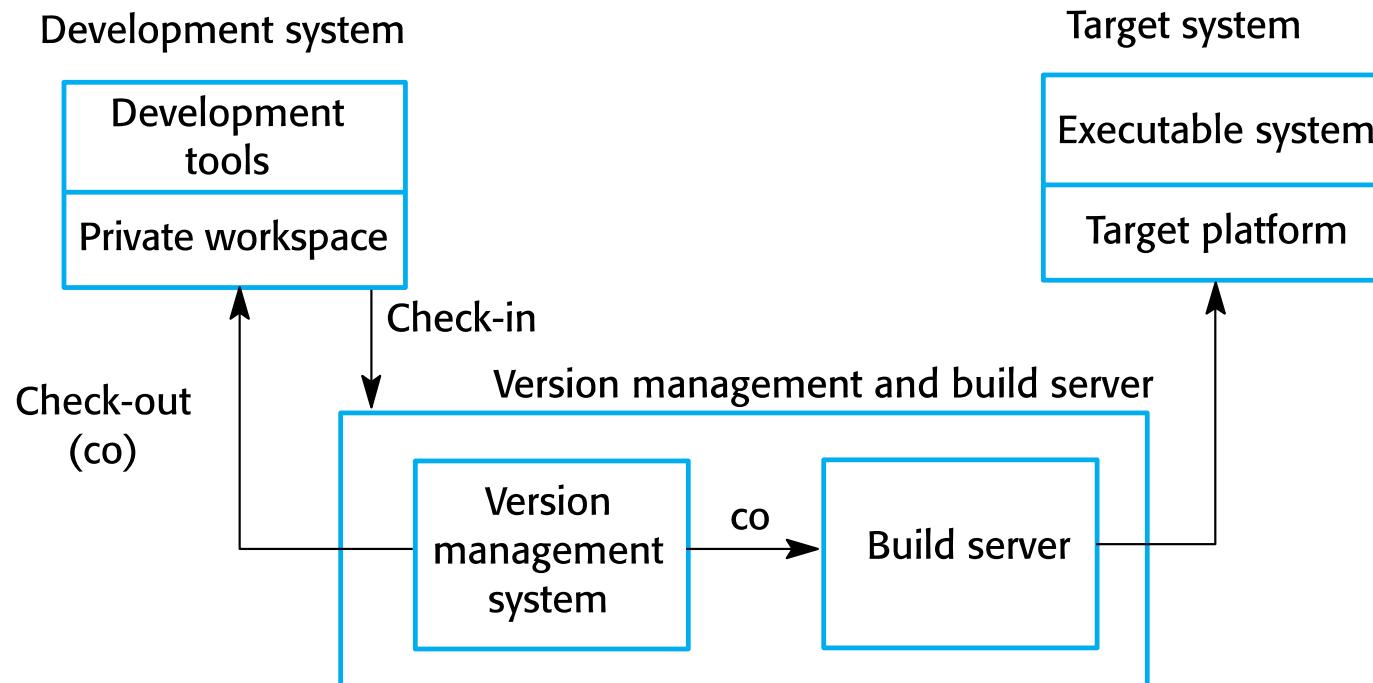
- ✧ Build script generation
- ✧ Version management system integration
- ✧ Minimal re-compilation
- ✧ Executable system creation
- ✧ Test automation
- ✧ Reporting
- ✧ Documentation generation

# System platforms

---

- ✧ The development system, which includes development tools such as compilers, source code editors, etc.
- ✧ The build server, which is used to build definitive, executable versions of the system. This server maintains the definitive versions of a system.
- ✧ The target environment, which is the platform on which the system executes.
  - For real-time and embedded systems, the target environment is often smaller and simpler than the development environment (e.g. a cell phone)

# Development, build, and target platforms



# Agile building

---

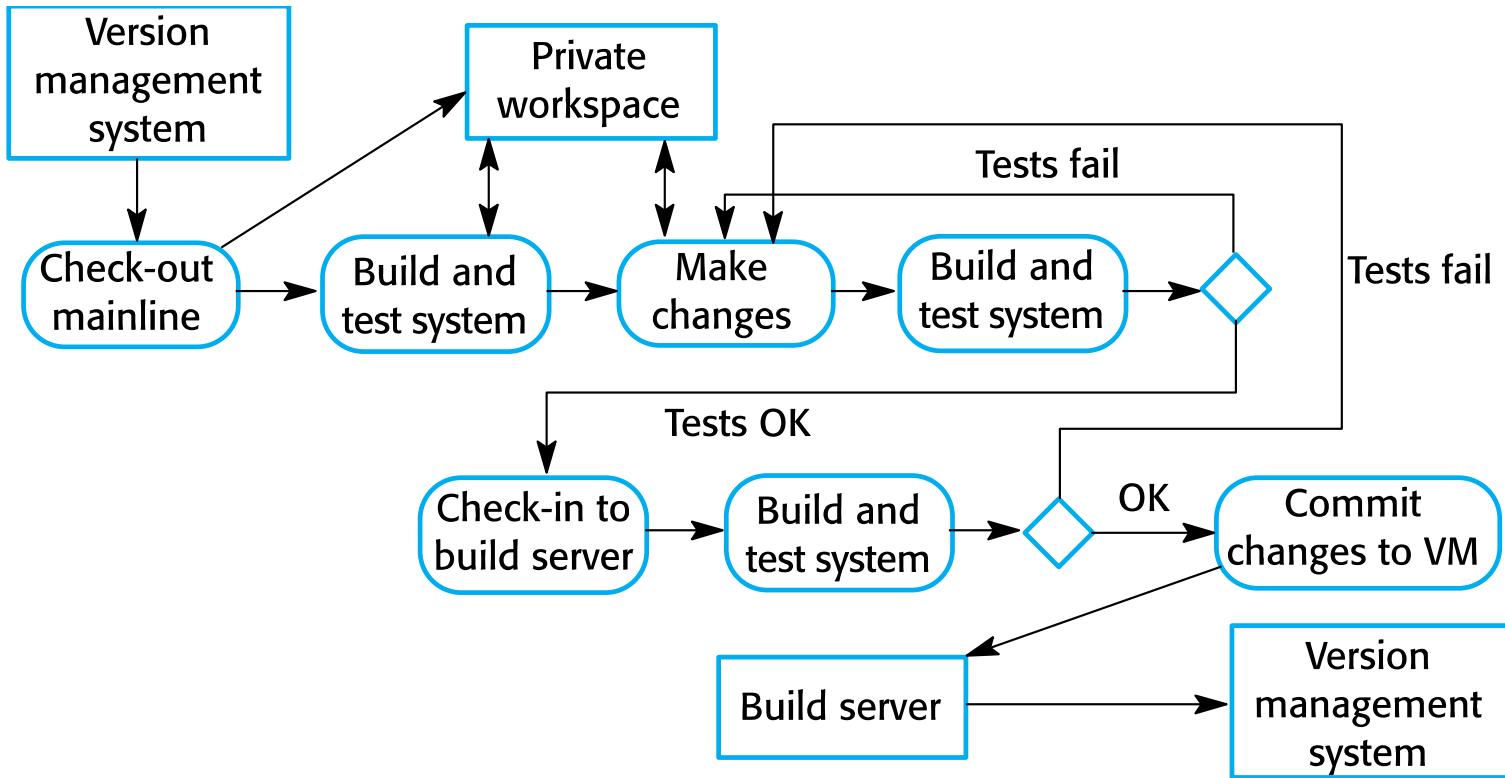
- ✧ Check out the mainline system from the version management system into the developer's private workspace.
- ✧ Build the system and run automated tests to ensure that the built system passes all tests. If not, the build is broken and you should inform whoever checked in the last baseline system. They are responsible for repairing the problem.
- ✧ Make the changes to the system components.
- ✧ Build the system in the private workspace and rerun system tests. If the tests fail, continue editing.

# Agile building

---

- ✧ Once the system has passed its tests, check it into the build system but do not commit it as a new system baseline.
- ✧ Build the system on the build server and run the tests. You need to do this in case others have modified components since you checked out the system. If this is the case, check out the components that have failed and edit these so that tests pass on your private workspace.
- ✧ If the system passes its tests on the build system, then commit the changes you have made as a new baseline in the system mainline.

# Continuous integration



# Pros and cons of continuous integration

---

## ✧ Pros

- The advantage of continuous integration is that it allows problems caused by the interactions between different developers to be discovered and repaired as soon as possible.
- The most recent system in the mainline is the definitive working system.

## ✧ Cons

- If the system is very large, it may take a long time to build and test, especially if integration with other application systems is involved.
- If the development platform is different from the target platform, it may not be possible to run system tests in the developer's private workspace.

# Daily building

---

- ✧ The development organization sets a delivery time (say 2 p.m.) for system components.
  - If developers have new versions of the components that they are writing, they must deliver them by that time.
  - A new version of the system is built from these components by compiling and linking them to form a complete system.
  - This system is then delivered to the testing team, which carries out a set of predefined system tests
  - Faults that are discovered during system testing are documented and returned to the system developers. They repair these faults in a subsequent version of the component.

# Minimizing recompilation

---

- ✧ Tools to support system building are usually designed to minimize the amount of compilation that is required.
- ✧ They do this by checking if a compiled version of a component is available. If so, there is no need to recompile that component.
- ✧ A unique signature identifies each source and object code version and is changed when the source code is edited.
- ✧ By comparing the signatures on the source and object code files, it is possible to decide if the source code was used to generate the object code component.

# File identification

---

## ✧ Modification timestamps

- The signature on the source code file is the time and date when that file was modified. If the source code file of a component has been modified after the related object code file, then the system assumes that recompilation to create a new object code file is necessary.

## ✧ Source code checksums

- The signature on the source code file is a checksum calculated from data in the file. A checksum function calculates a unique number using the source text as input. If you change the source code (even by 1 character), this will generate a different checksum. You can therefore be confident that source code files with different checksums are actually different.

# Timestamps vs checksums

---

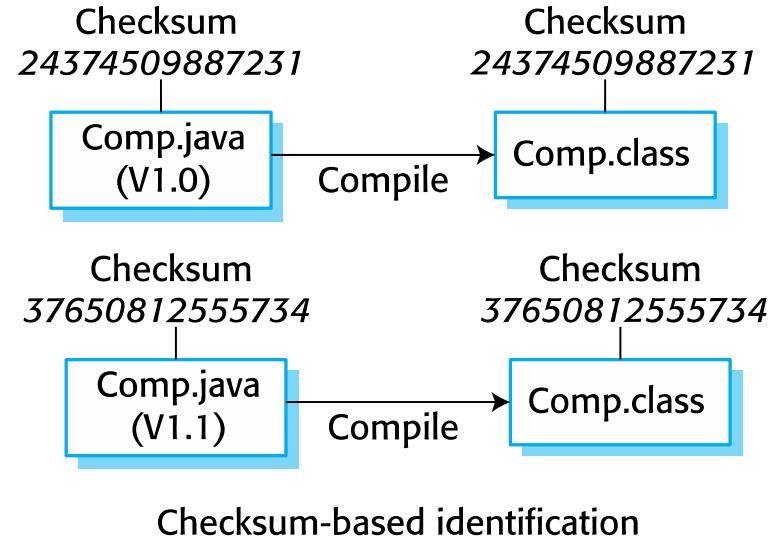
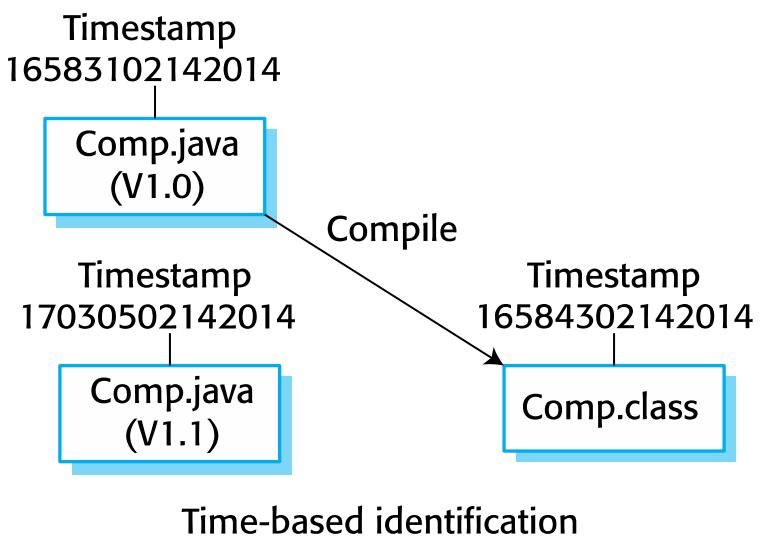
## ✧ Timestamps

- Because source and object files are linked by name rather than an explicit source file signature, it is not usually possible to build different versions of a source code component into the same directory at the same time, as these would generate object files with the same name.

## ✧ Checksums

- When you recompile a component, it does not overwrite the object code, as would normally be the case when the timestamp is used. Rather, it generates a new object code file and tags it with the source code signature. Parallel compilation is possible and different versions of a component may be compiled at the same time.

# Linking source and object code



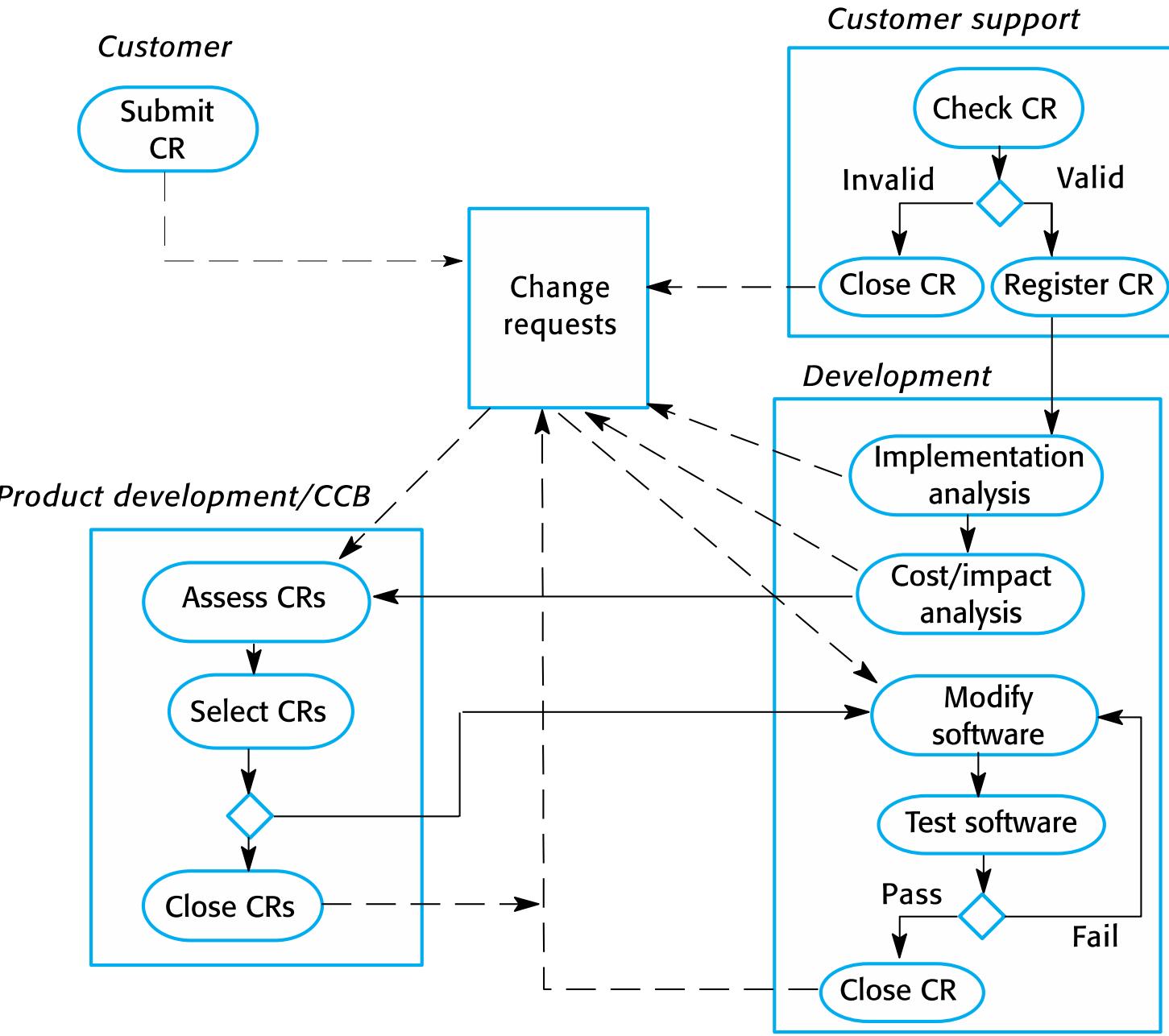
---

# **Change management**

# Change management

---

- ✧ Organizational needs and requirements change during the lifetime of a system, bugs have to be repaired and systems have to adapt to changes in their environment.
- ✧ Change management is intended to ensure that system evolution is a managed process and that priority is given to the most urgent and cost-effective changes.
- ✧ The change management process is concerned with analyzing the costs and benefits of proposed changes, approving those changes that are worthwhile and tracking which components in the system have been changed.



## The change management process

# A partially completed change request form (a)

---

## Change Request Form

**Project:** SICSA/AppProcessing

**Number:** 23/02

**Change requester:** I. Sommerville

**Date:** 20/07/12

**Requested change:** The status of applicants (rejected, accepted, etc.) should be shown visually in the displayed list of applicants.

**Change analyzer:** R. Looek

**Analysis date:** 25/07/12

**Components affected:** ApplicantListDisplay, StatusUpdater

**Associated components:** StudentDatabase

# A partially completed change request form (b)

---

## Change Request Form

**Change assessment:** Relatively simple to implement by changing the display color according to status. A table must be added to relate status to colors. No changes to associated components are required.

**Change priority:** Medium

**Change implementation:**

**Estimated effort:** 2 hours

**Date to SGA app. team:** 28/07/12

**CCB decision date:** 30/07/12

**Decision:** Accept change. Change to be implemented in Release 1.2

**Change implementor:**      **Date of change:**

**Date submitted to QA:**      **QA decision:**

**Date submitted to CM:**

**Comments:**

# Factors in change analysis

---

- ✧ The consequences of not making the change
- ✧ The benefits of the change
- ✧ The number of users affected by the change
- ✧ The costs of making the change
- ✧ The product release cycle

# Derivation history

---

```
// SICSA project (XEP 6087)
```

```
//
```

```
// APP-SYSTEM/AUTH/RBAC/USER_ROLE
```

```
//
```

```
// Object: currentRole
```

```
// Author: R. Looek
```

```
// Creation date: 13/11/2012
```

```
//
```

```
// © St Andrews University 2012
```

```
//
```

```
// Modification history
```

Version	Modifier	Date	Change	Reason
---------	----------	------	--------	--------

1.0	J. Jones	11/11/2009	Add header	Submitted to CM
-----	----------	------------	------------	-----------------

1.1	R. Looek	13/11/2012	New field	Change req. R07/02
-----	----------	------------	-----------	--------------------

# Change management and agile methods

---

- ✧ In some agile methods, customers are directly involved in change management.
- ✧ They propose a change to the requirements and work with the team to assess its impact and decide whether the change should take priority over the features planned for the next increment of the system.
- ✧ Changes to improve the software improvement are decided by the programmers working on the system.
- ✧ Refactoring, where the software is continually improved, is not seen as an overhead but as a necessary part of the development process.

---

# **Release management**

# Release management

---

- ✧ A system release is a version of a software system that is distributed to customers.
- ✧ For mass market software, it is usually possible to identify two types of release: major releases which deliver significant new functionality, and minor releases, which repair bugs and fix customer problems that have been reported.
- ✧ For custom software or software product lines, releases of the system may have to be produced for each customer and individual customers may be running several different releases of the system at the same time.

# Release components

---

- ✧ As well as the executable code of the system, a release may also include:
  - configuration files defining how the release should be configured for particular installations;
  - data files, such as files of error messages, that are needed for successful system operation;
  - an installation program that is used to help install the system on target hardware;
  - electronic and paper documentation describing the system;
  - packaging and associated publicity that have been designed for that release.

# Factors influencing system release planning

---

Factor	Description
Competition	For mass-market software, a new system release may be necessary because a competing product has introduced new features and market share may be lost if these are not provided to existing customers.
Marketing requirements	The marketing department of an organization may have made a commitment for releases to be available at a particular date.
Platform changes	You may have to create a new release of a software application when a new version of the operating system platform is released.
Technical quality of the system	If serious system faults are reported which affect the way in which many customers use the system, it may be necessary to issue a fault repair release. Minor system faults may be repaired by issuing patches (usually distributed over the Internet) that can be applied to the current release of the system.

# Release creation

---

- ✧ The executable code of the programs and all associated data files must be identified in the version control system.
- ✧ Configuration descriptions may have to be written for different hardware and operating systems.
- ✧ Update instructions may have to be written for customers who need to configure their own systems.
- ✧ Scripts for the installation program may have to be written.
- ✧ Web pages have to be created describing the release, with links to system documentation.
- ✧ When all information is available, an executable master image of the software must be prepared and handed over for distribution to customers or sales outlets.

# Release tracking

---

- ✧ In the event of a problem, it may be necessary to reproduce exactly the software that has been delivered to a particular customer.
- ✧ When a system release is produced, it must be documented to ensure that it can be re-created exactly in the future.
- ✧ This is particularly important for customized, long-lifetime embedded systems, such as those that control complex machines.
  - Customers may use a single release of these systems for many years and may require specific changes to a particular software system long after its original release date.

# Release reproduction

---

- ✧ To document a release, you have to record the specific versions of the source code components that were used to create the executable code.
- ✧ You must keep copies of the source code files, corresponding executables and all data and configuration files.
- ✧ You should also record the versions of the operating system, libraries, compilers and other tools used to build the software.

# Release planning

---

- ✧ As well as the technical work involved in creating a release distribution, advertising and publicity material have to be prepared and marketing strategies put in place to convince customers to buy the new release of the system.
- ✧ Release timing
  - If releases are too frequent or require hardware upgrades, customers may not move to the new release, especially if they have to pay for it.
  - If system releases are too infrequent, market share may be lost as customers move to alternative systems.

# Software as a service

---

- ✧ Delivering software as a service (SaaS) reduces the problems of release management.
- ✧ It simplifies both release management and system installation for customers.
- ✧ The software developer is responsible for replacing the existing release of a system with a new release and this is made available to all customers at the same time.

# Key points

---

- ✧ Configuration management is the management of an evolving software system. When maintaining a system, a CM team is put in place to ensure that changes are incorporated into the system in a controlled way and that records are maintained with details of the changes that have been implemented.
- ✧ The main configuration management processes are concerned with version management, system building, change management, and release management.
- ✧ Version management involves keeping track of the different versions of software components as changes are made to them.

# Key points

---

- ✧ System building is the process of assembling system components into an executable program to run on a target computer system.
- ✧ Software should be frequently rebuilt and tested immediately after a new version has been built. This makes it easier to detect bugs and problems that have been introduced since the last build.
- ✧ Change management involves assessing proposals for changes from system customers and other stakeholders and deciding if it is cost-effective to implement these in a new version of a system.
- ✧ System releases include executable code, data files, configuration files and documentation. Release management involves making decisions on system release dates, preparing all information for distribution and documenting each system release.