



# Why Software Engineering ?

---

- The problem is *complexity*
- Many sources, but *size* is key:
  - UNIX contains 4 million lines of code
  - Windows 2000 contains  $10^8$  lines of code

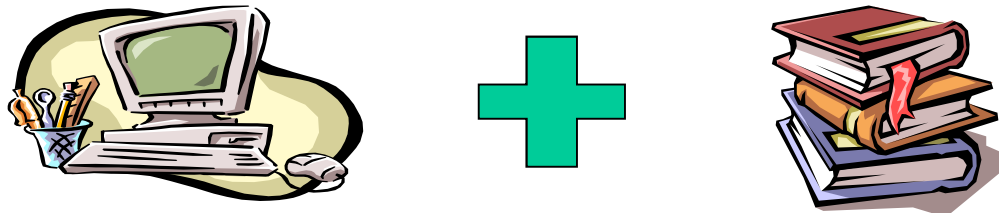
Software engineering is about managing  
this complexity.

# What is Software Engineering?

- ▶ The process of solving customers' problems by the systematic development and evolution of large, high-quality software systems within cost, time and other constraints
- ▶ Note:
  - Process, systematic (not ad hoc), evolutionary...
  - Constraints: high quality, cost, time, meets user requirements

# 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
- **Software products** may be
  - **Generic** - developed to be sold to a **range** of different customers
  - **Bespoke** (custom) - developed for a **single** customer according to their specification

# What is software engineering?

**Software engineering** is an engineering discipline which is concerned with **all** aspects of software **production**

**Software engineers** should

- **adopt a systematic and organised approach to their work**
- **use appropriate tools and techniques** depending on
  - the **problem** to be solved,
  - the development **constraints** and
  - the **resources** available



# What is the difference between software engineering and computer science?

## Computer Science

- theory
- fundamentals

is concerned with

## Software Engineering

- the practicalities of developing
- delivering useful software

*Computer science theories* are currently insufficient to act as a complete underpinning for software engineering



ALC1

## Slide 5

---

**ALC1**

دعامة  
AL Laith Co, 10/7/2018



# 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 process
- **System engineers** are involved in system specification, architectural design, integration and deployment



# What is a software process?

---

- A **set of activities** whose goal is the development or evolution of software
- Generic activities in all software processes are:
  - **Specification** - what the system should do and its development constraints
  - **Development** - production of the software system(design +program).
  - **Validation** - checking that the software is what the customer wants
  - **Evolution** - changing the software in response to changing demands





# What is a software process model?

A simplified representation of a software process,  
presented from a specific perspective

- **Examples of process perspectives:**

**Workflow perspective** represents **inputs, outputs** and dependencies

**Data-flow perspective** represents **data transformation** activities

**Role/action perspective** represents the roles/activities of the  
people involved in software process

- **Generic process models**

- **Waterfall**

- **Evolutionary development**

- **Formal transformation**

- **Integration from reusable components**



# What are the **costs of software engineering**?

---

- **Roughly 60% of costs are development costs, 40% are testing costs.** For custom software, evolution costs often exceed development costs
- **Costs vary depending on the 1- type of system** being developed **and 2-the requirements** of system attributes such as performance and system **reliability**
- **Distribution of costs depends on the development model that is used**



# What are software engineering methods?

---

**Structured approaches to software development**  
which include **system models, notations, rules, design advice** and **process guidance**

---

- **Model descriptions** (*Descriptions of graphical models which should be produced*)
- **Rules** (*Constraints applied to system models*)
- **Recommendations** (*Advice on good design practice*)
- **Process guidance** (*What activities to follow*)

# What is CASE ?

(Computer-Aided Software Engineering)

Software systems which are intended to provide **automated support** for software process **activities**, such as requirements analysis, system modelling, debugging and testing

- **Upper-CASE**

- Tools to support the early process activities of requirements and design

- **Lower-CASE**

- Tools to support later activities such as programming, debugging and testing





# What are the attributes of good software?

---

The software should deliver the required functionality and performance to the user and should be **maintainable, dependable and usable**

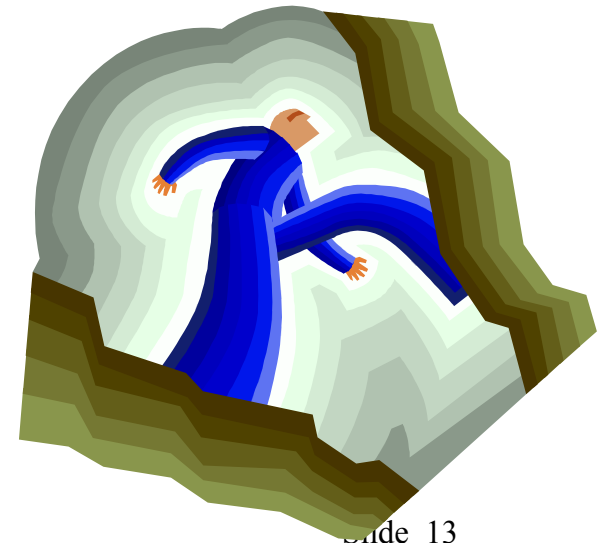
---

- **Maintainability**
  - Software must evolve to meet changing needs
- **Dependability**
  - Software must be trustworthy
- **Efficiency**
  - Software should not make wasteful use of system resources
- **Usability**
  - Software must be usable by the users for which it was designed

# What are the key challenges facing software engineering?

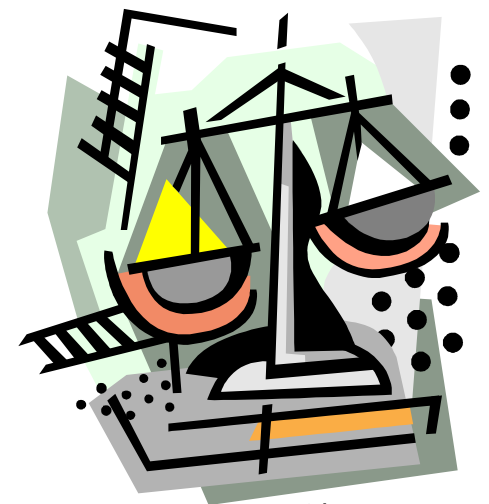
Software engineering in the 21<sup>st</sup> century faces three key challenges:

- **Legacy systems**
  - Old, valuable systems must be **maintained** and **updated**
- **Heterogeneity**
  - Systems are **distributed** and include a **mix** of hardware and software
- **Delivery**
  - There is increasing **pressure** for faster delivery of software



# Professional and ethical responsibility

- **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**



# Managing the Software Process



# 11.1 What is Project Management?

**Project management encompasses all the activities needed to plan and execute a project:**

- Deciding what needs to be done
- Estimating costs
- Ensuring there are suitable people to undertake the project
- Defining responsibilities
- Scheduling
- Making arrangements for the work
- *continued ...*

# What is Project Management?

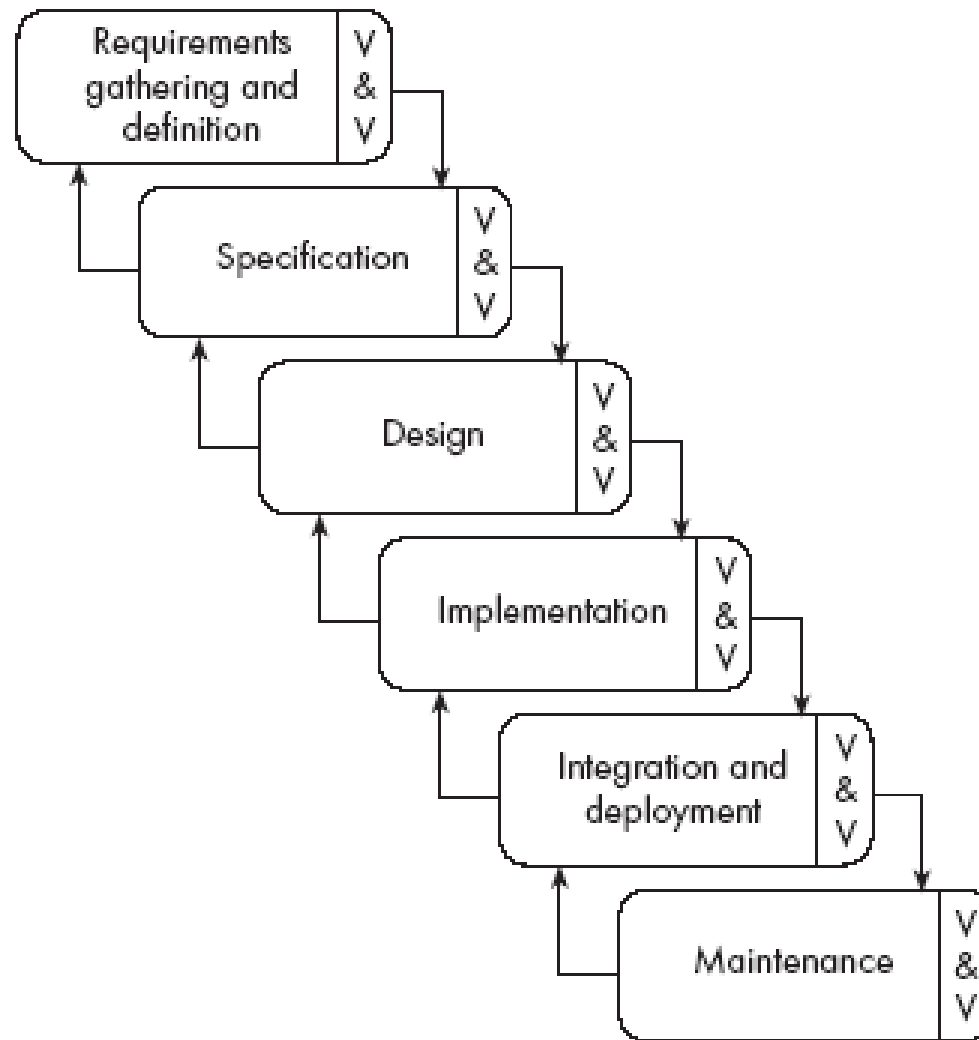
- Directing
- Being a technical leader
- Reviewing and approving decisions made by others
- Building morale and supporting staff
- Monitoring and controlling
- Co-ordinating the work with managers of other projects
- Reporting
- Continually striving to improve the process

# 11.2 Software Process Models

**Software process models are general approaches for organizing a project into activities.**

- Help the project manager and his or her team to decide:
  - What work should be done;
  - In what sequence to perform the work.
- The models should be seen as *aids to thinking*, not rigid prescriptions of the way to do things.
- Each project ends up with its own unique plan.

# The waterfall model



# The waterfall model

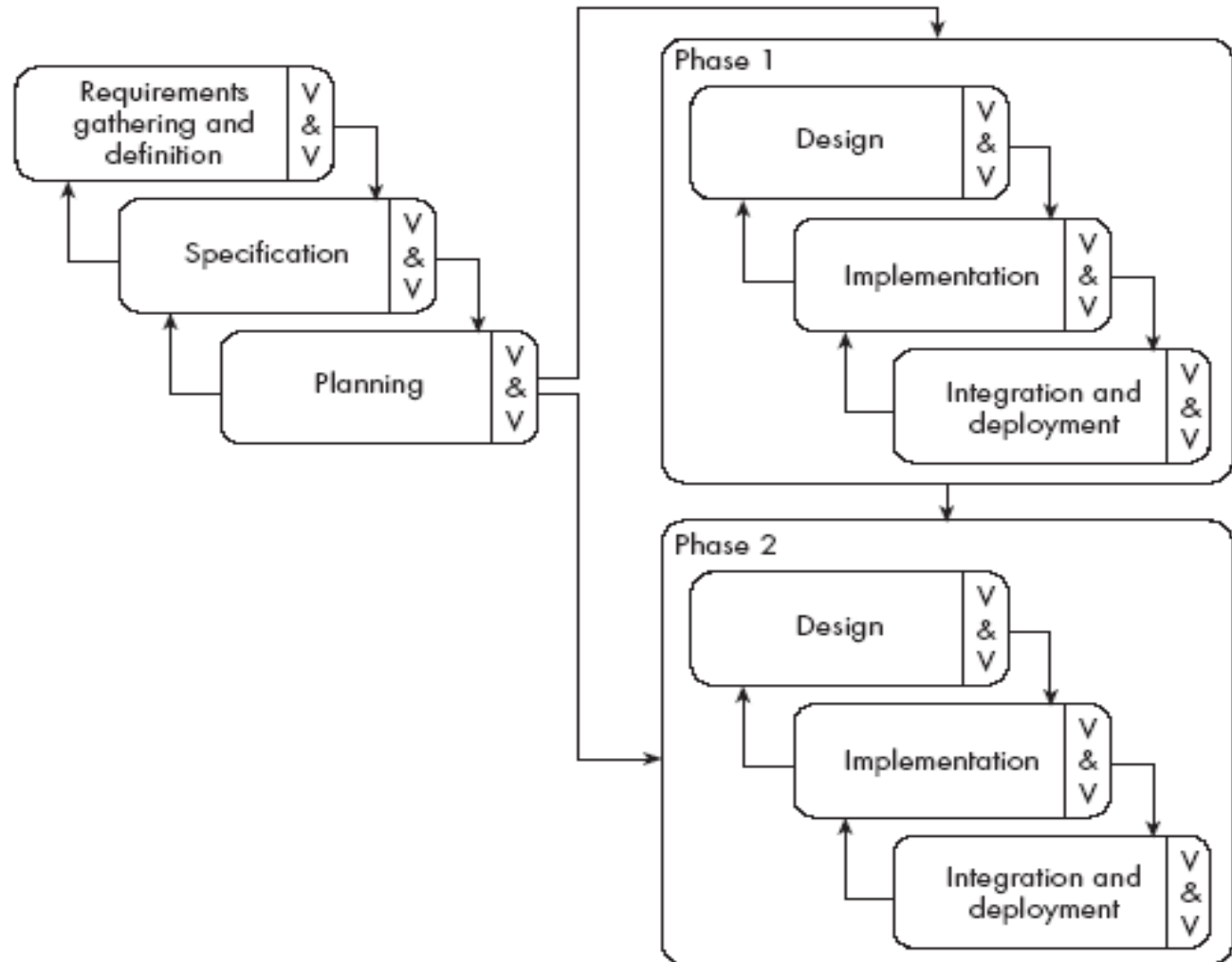
**The classic way of looking at S.E. that accounts for the importance of requirements, design and quality assurance.**

- The model suggests that software engineers should work in a series of stages.
- Before completing each stage, they should perform quality assurance (verification and validation).
- The waterfall model also recognizes, to a limited extent, that you sometimes have to step back to earlier stages.

# Limitations of the waterfall model

- The model implies that you should attempt to complete a given stage before moving on to the next stage
  - Does not account for the fact that requirements constantly change.
  - It also means that customers can not use anything until the entire system is complete.
- The model makes no allowances for prototyping.
- It implies that you can get the requirements right by simply writing them down and reviewing them.
- The model implies that once the product is finished, everything else is maintenance.

# The phased-release model



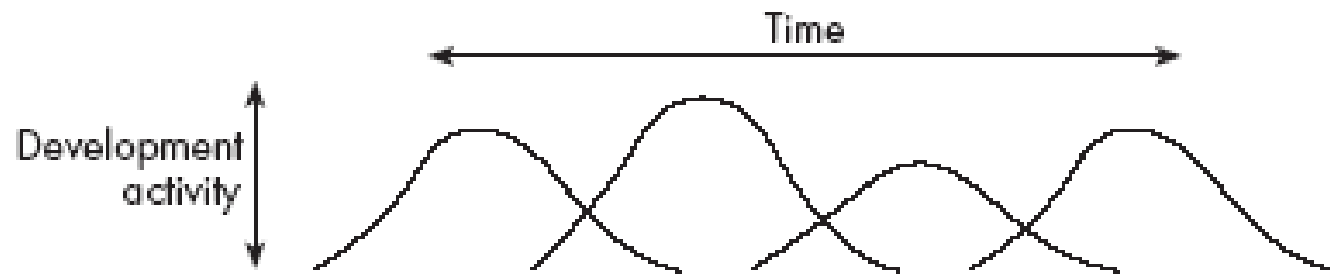
# The phased-release model

**It introduces the notion of *incremental* development.**

- After requirements gathering and planning, the project should be broken into separate subprojects, or *phases*.
- Each phase can be released to customers when ready.
- Parts of the system will be available earlier than when using a strict waterfall approach.
- However, it continues to suggest that all requirements be finalized at the start of development.



# The evolutionary model

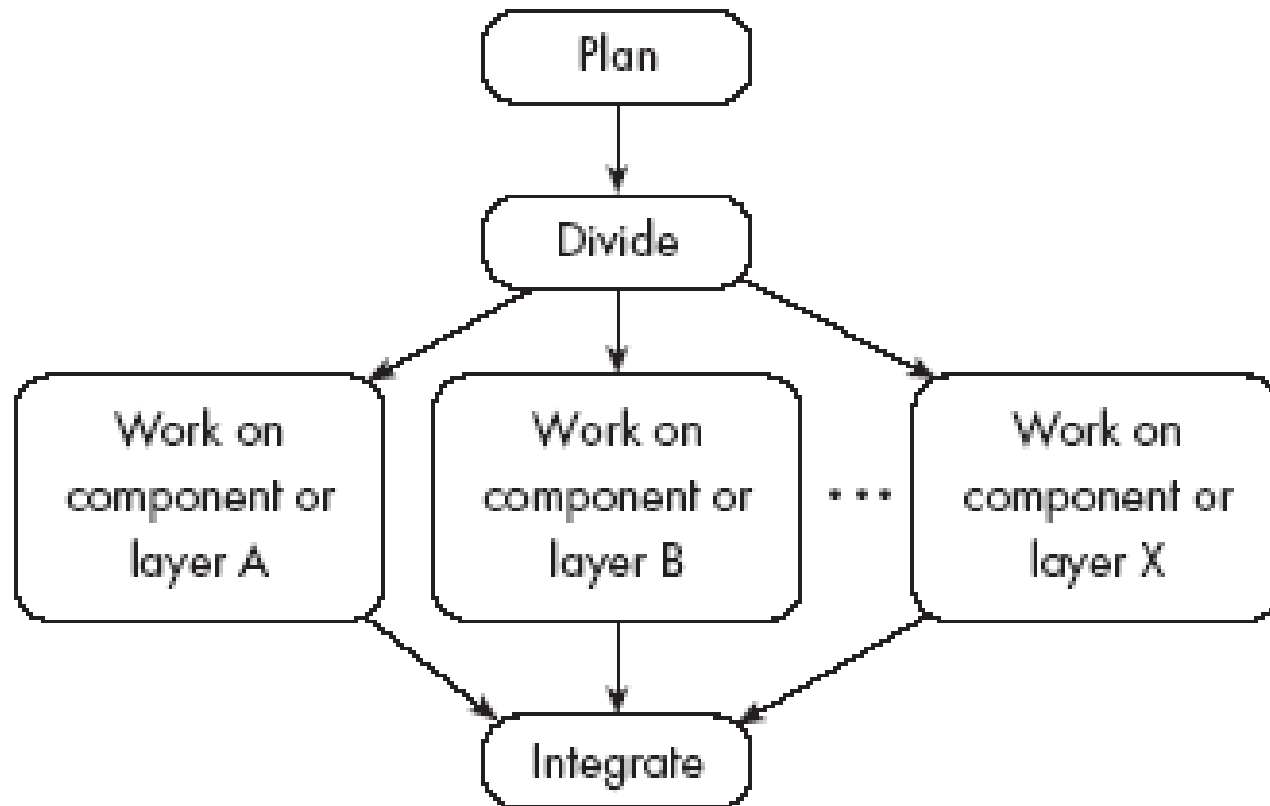


# The evolutionary model

**It shows software development as a series of hills, each representing a separate loop of the spiral.**

- Shows that loops, or releases, tend to overlap each other.
- Makes it clear that development work tends to reach a peak, at around the time of the deadline for completion.
- Shows that each prototype or release can take
  - different amounts of time to deliver;
  - differing amounts of effort.

# The concurrent engineering model



# The concurrent engineering model

**It explicitly accounts for the divide and conquer principle.**

- Each team works on its own component, typically following a spiral or evolutionary approach.
- There has to be some initial planning, and periodic integration.

# Choosing a process model

- From the waterfall model:
  - Incorporate the notion of stages.
- From the phased-release model:
  - Incorporate the notion of doing some initial high-level analysis, and then dividing the project into releases.
- From the spiral model:
  - Incorporate prototyping and risk analysis.
- From the evolutionary model:
  - Incorporate the notion of varying amounts of time and work, with overlapping releases.
- From concurrent engineering:
  - Incorporate the notion of breaking the system down into components and developing them in parallel.

# 11.3 Cost estimation

**To estimate how much software-engineering time will be required to do some work.**

- *Elapsed time*
  - The difference in time from the start date to the end date of a task or project.
- *Development effort*
  - The amount of labour used in *person-months* or *person-days*.
  - To convert an estimate of development effort to an amount of money:

You multiply it by the *weighted average cost (burdened cost)* of employing a software engineer for a month (or a day).

# Principles of effective cost estimation

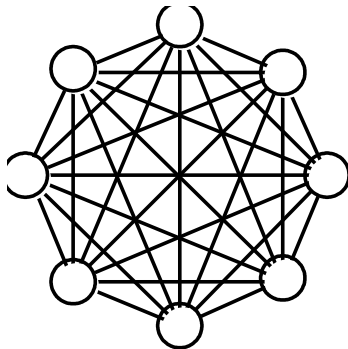
## **Principle: Include all activities when making estimates.**

- The time required for *all* development activities must be taken into account.
- Including:
  - Prototyping
  - Design
  - Inspecting
  - Testing
  - Debugging
  - Writing user documentation
  - Deployment.

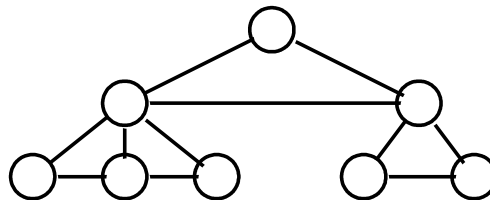
# 11.4 Building Software Engineering Teams

**Software engineering is a human process.**

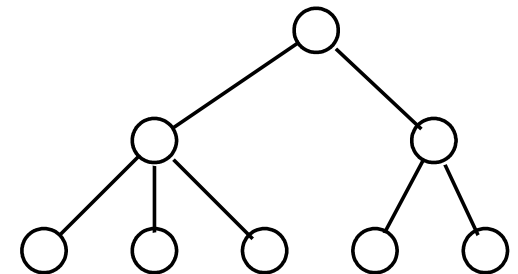
- Choosing appropriate people for a team, and assigning roles and responsibilities to the team members, is therefore an important project management skill
- Software engineering teams can be organized in many different ways



a) Egoless



b) Chief programmer



c) Strict hierarchy



# Software engineering teams

## **Egoless team:**

- In such a team everybody is equal, and the team works together to achieve a common goal.
- Decisions are made by consensus.
- Most suited to difficult projects with many technical challenges.

# Software engineering teams

## **Hierarchical manager-subordinate structure:**

- Each individual reports to a manager and is responsible for performing the tasks delegated by that manager.
- Suitable for large projects with a strict schedule where everybody is well-trained and has a well-defined role.
- However, since everybody is only responsible for their own work, problems may go unnoticed.

# Software engineering teams

## **Chief programmer team:**

- Midway between egoless and hierarchical.
- The chief programmer leads and guides the project.
- He or she consults with, and relies on, individual specialists.

# Choosing an effective size for a team

- For a given estimated development effort, in person months, there is an optimal team size.
  - Doubling the size of a team will not halve the development time.
- Subsystems and teams should be sized such that the total amount of required knowledge and exchange of information is reduced.
- For a given project or project iteration, the number of people on a team will not be constant.
- You can not generally add people if you get behind schedule, in the hope of catching up.

# Skills needed on a team

- Architect
- Project manager
- Configuration management and build specialist
- User interface specialist
- Technology specialist
- Hardware and third-party software specialist
- User documentation specialist
- Tester

# 11.5 Project Scheduling and Tracking

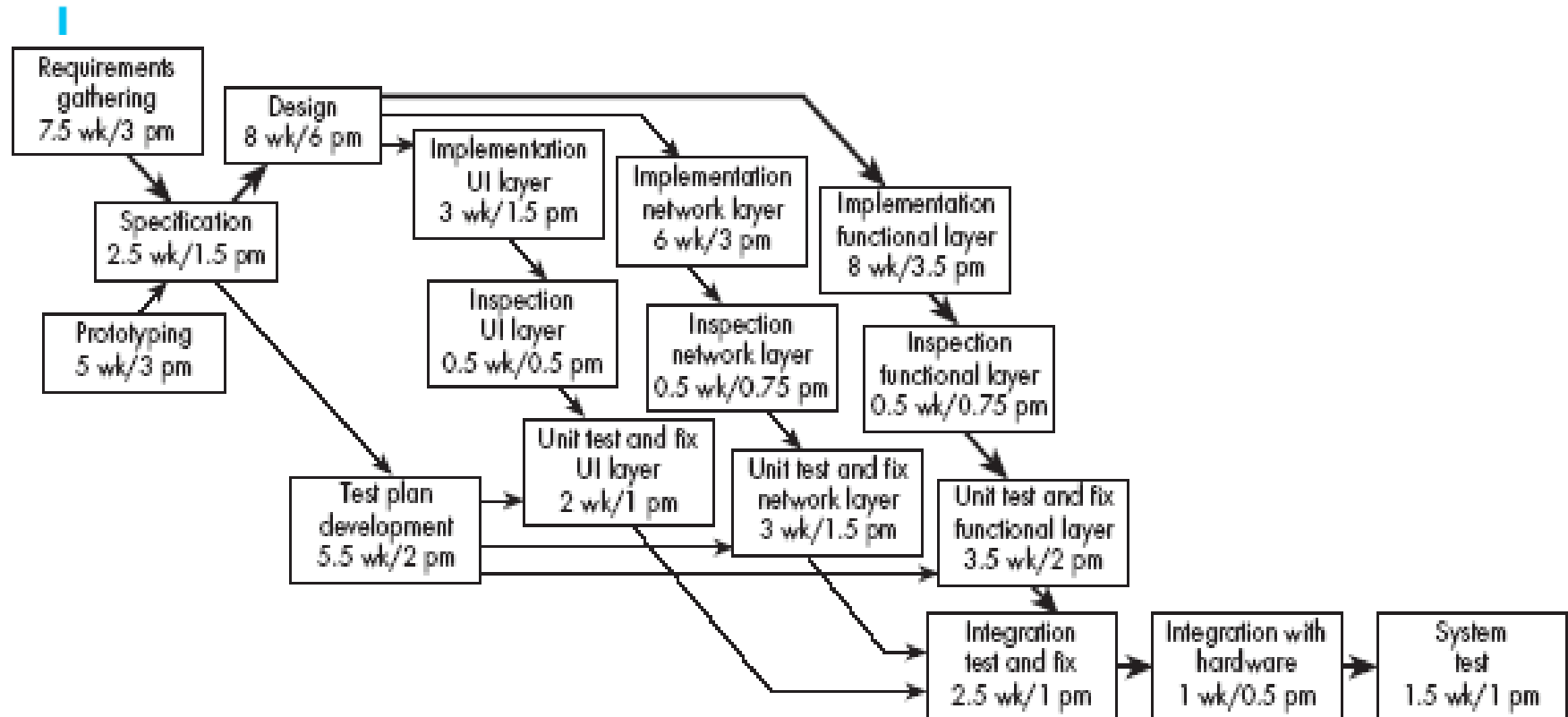
- *Scheduling* is the process of deciding:
  - In what sequence a set of activities will be performed.
  - When they should start and be completed.
- *Tracking* is the process of determining how well you are sticking to the cost estimate and schedule.

# PERT charts

**A PERT chart shows the sequence in which tasks must be completed.**

- In each node of a PERT chart, you typically show the elapsed time and effort estimates.
- The *critical path* indicates the minimum time in which it is possible to complete the project.

# Example of a PERT chart



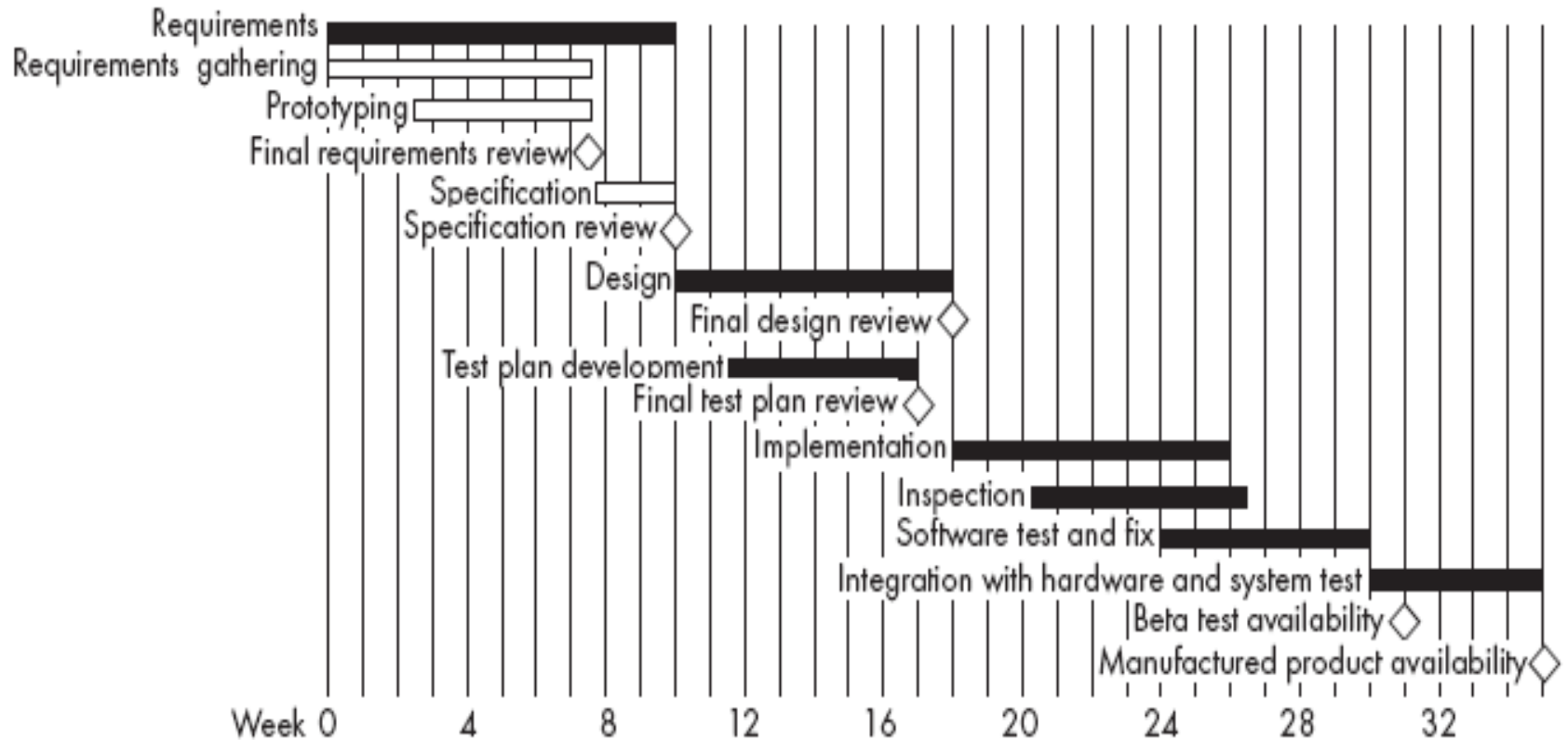


# Gantt charts

**A Gantt chart is used to graphically present the start and end dates of each software engineering task**

- One axis shows time.
- The other axis shows the activities that will be performed.
- The black bars are the top-level tasks.
- The white bars are subtasks
- The diamonds are *milestones*:
  - Important deadline dates, at which specific events may occur

# Example of a Gantt chart



# 11.6 Contents of a Project Plan

- A. Purpose**
- B. Background information**
- C. Processes to be used**
- D. Subsystems and planned releases**
- E. Risks and challenges**
- F. Tasks**
- G. Cost estimates**
- H. Team**
- I. Schedule and milestones**

# 11.7 Difficulties and Risks in Project Management

- **Accurately estimating costs is a constant challenge**
  - Follow the cost estimation guidelines.*
- **It is very difficult to measure progress and meet deadlines**
  - Improve your cost estimation skills so as to account for the kinds of problems that may occur.*
  - Develop a closer relationship with other members of the team.*
  - Be realistic in initial requirements gathering, and follow an iterative approach.*
  - Use earned value charts to monitor progress.*

# Difficulties and Risks in Project Management

- **It is difficult to deal with lack of human resources or technology needed to successfully run a project**
  - *When determining the requirements and the project plan, take into consideration the resources available.*
  - *If you cannot find skilled people or suitable technology then you must limit the scope of your project.*

# Difficulties and Risks in Project Management

- **Communicating effectively in a large project is hard**
  - Take courses in communication, both written and oral.*
  - Learn how to run effective meetings.*
  - Review what information everybody should have, and make sure they have it.*
  - Make sure that project information is readily available.*
  - Use 'groupware' technology to help people exchange the information they need to know*

# Difficulties and Risks in Project Management

- **It is hard to obtain agreement and commitment from others**
  - *Take courses in negotiating skills and leadership.*
  - *Ensure that everybody understands*
    - *The position of everybody else.*
    - *The costs and benefits of each alternative.*
    - *The rationale behind any compromises.*
  - *Ensure that everybody's proposed responsibility is clearly expressed.*
  - *Listen to everybody's opinion, but take assertive action, when needed, to ensure progress occurs.*

---

# Project Planning



# Project planning

---

- Probably the most time-consuming project management activity
- Continuous activity from initial concept through to system delivery. Plans must be regularly revised as new information becomes available
- Various different types of plan may be developed to support the main software project plan that is concerned with schedule and budget

# Types of project plan

---

Plan	Description
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.
Configuration management plan	Describes the configuration management procedures and structures to be used.
Maintenance plan	Predicts the maintenance requirements of the system, maintenance costs and effort required.
Staff development plan.	Describes how the skills and experience of the project team members will be developed.

# Project plan structure

---

- Introduction
- Project organisation
- Risk analysis
- Hardware and software resource requirements
- Work breakdown
- Project schedule
- Monitoring and reporting mechanisms

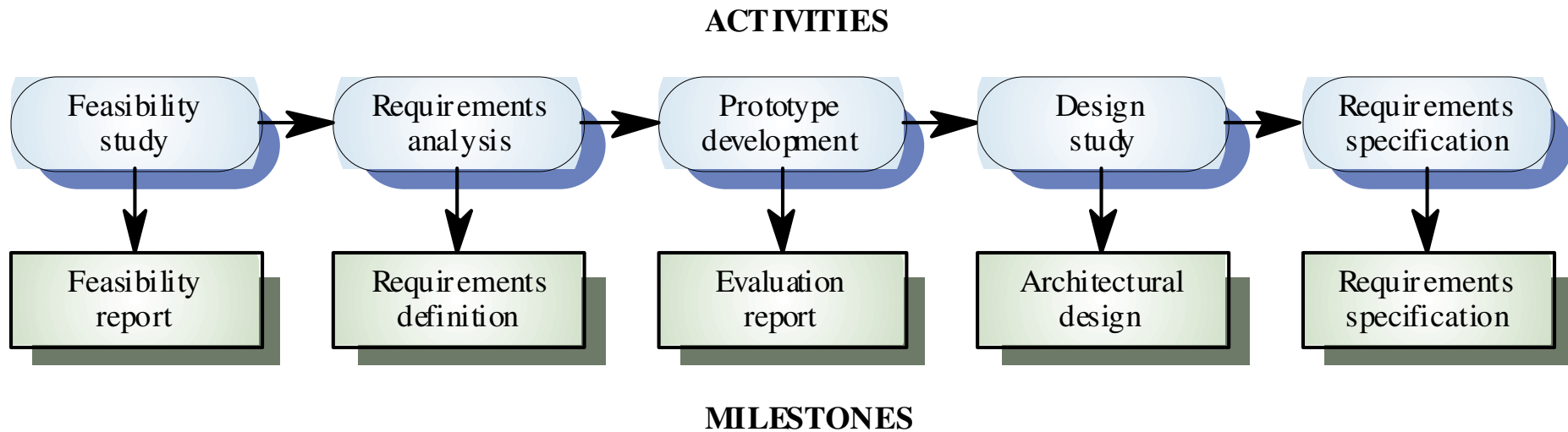
# Activity organization

---

- Activities in a project should be organised to produce tangible outputs for management to judge progress
- *Milestones* are the end-point of a process activity
- *Deliverables* are project results delivered to customers
- The waterfall process allows for the straightforward definition of progress milestones

# Milestones in the RE process

---



# Project scheduling

---

- 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

# 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

# Bar charts and activity networks

---

- Graphical notations used to illustrate the project schedule
- Show project breakdown into tasks. Tasks should not be too small. They should take about a week or two
- Activity charts show task dependencies and the critical path
- Bar charts show schedule against calendar time

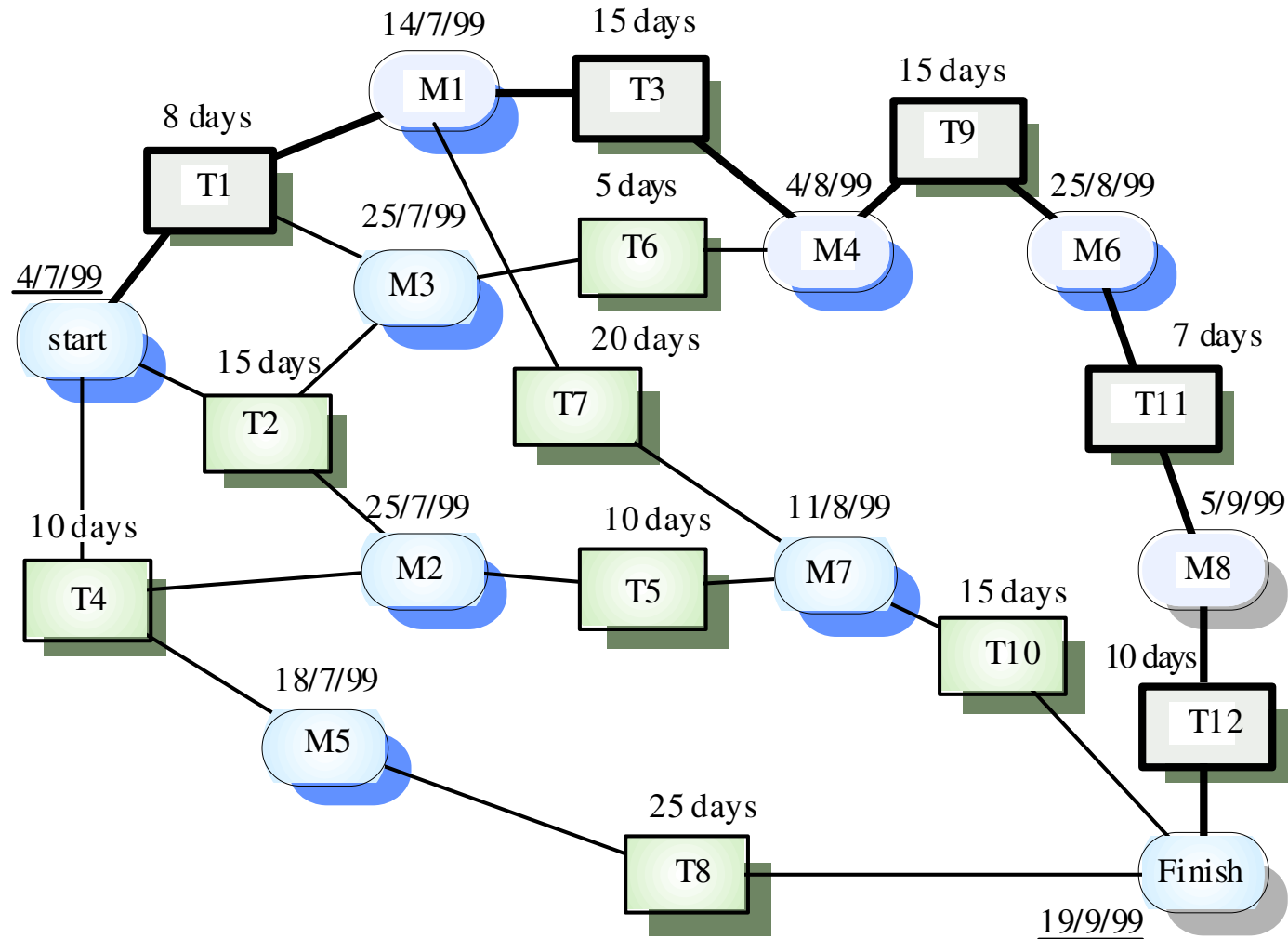


# Task durations and dependencies

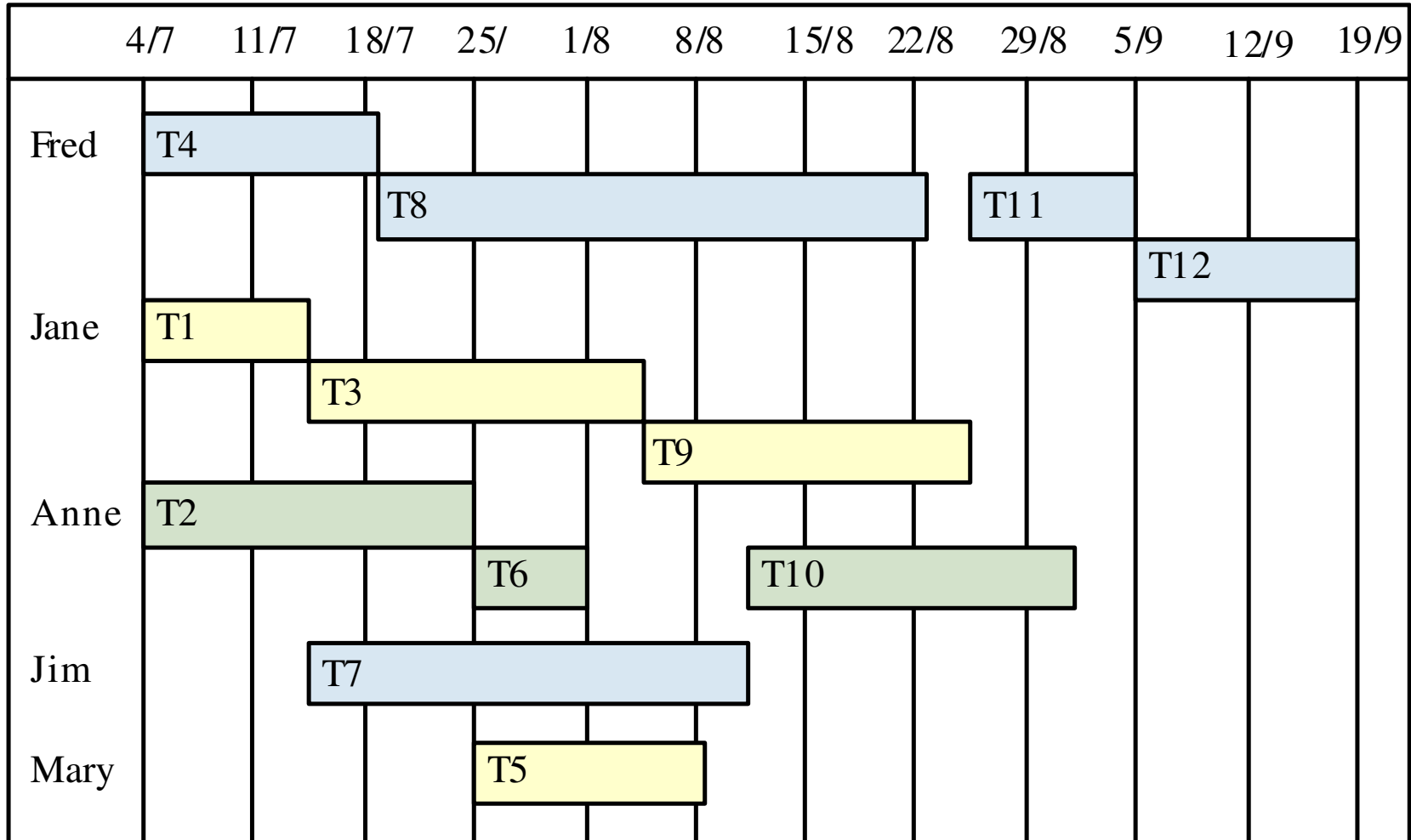
---

Task	Duration (days)	Dependencies
T1	8	
T2	15	
T3	15	T1 (M1)
T4	10	
T5	10	T2, T4 (M2)
T6	5	T1, T2 (M3)
T7	20	T1 (M1)
T8	25	T4 (M5)
T9	15	T3, T6 (M4)
T10	15	T5, T7 (M7)
T11	7	T9 (M6)
T12	10	T11 (M8)

# Activity network



# Staff allocation



# Risk identification

---

- Technology risks
- People risks
- Organisational risks
- Requirements risks
- Estimation risks

# Risks and risk types

---

<b>Risk type</b>	<b>Possible risks</b>
Technology	The database used in the system cannot process as many transactions per second as expected. Software components which should be reused contain defects which limit their functionality.
People	It is impossible to recruit staff with the skills required. Key staff are ill and unavailable at critical times. Required training for staff is not available.
Organisational	The organisation is restructured so that different management are responsible for the project. Organisational financial problems force reductions in the project budget.
Tools	The code generated by CASE tools is inefficient. CASE tools cannot be integrated.
Requirements	Changes to requirements which require major design rework are proposed. Customers fail to understand the impact of requirements changes.
Estimation	The time required to develop the software is underestimated. The rate of defect repair is underestimated. The size of the software is underestimated.

# Key points

---

- Planning and estimating are iterative processes which continue throughout the course of a project

# Key points

---

- A project milestone is a predictable state where some formal report of progress is presented to management.
- Risks may be project risks, product risks or business risks
- Risk management is concerned with identifying risks which may affect the project and planning to ensure that these risks do not develop into major threats



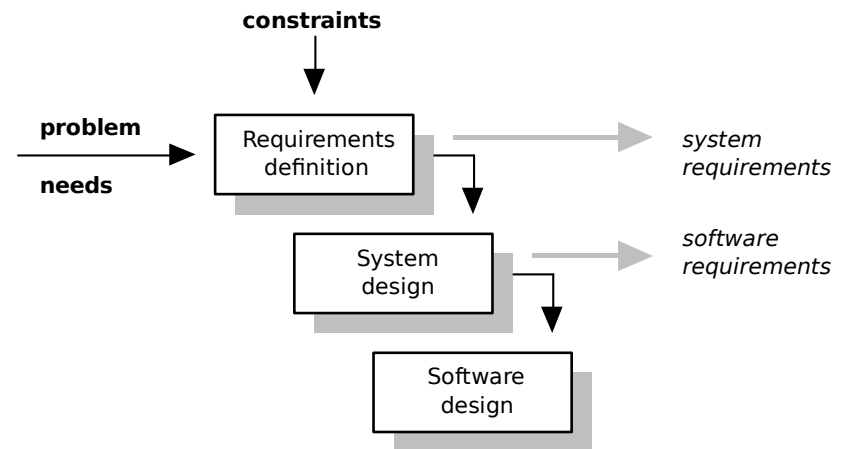
# Requirements and Architecture

---



# Requirements in Context

- Requirements may vary
  - in level of abstraction, contents
  - from one context to another
- System requirements
  - result from an analysis or discovery process
- Software requirements
  - result from a design process involving **requirements allocation**
- Sometimes there is no distinction between them





# Terminology

---

- A **requirement** is a technical objective which is imposed upon the software, i.e., anything that might affect the kind of software that is produced
- A requirement may be imposed by
  - the customer
  - the developer
  - the operating environment
- The source, rationale, and nature of the requirement must be documented
- Requirements fall into two broad categories
  - functional
  - non-functional



# Functional Requirements

---

- Functional requirements are concerned with **what** the software must do
  - capabilities, services, or operations
- Functional requirements are **not concerned with how** the software does things, i.e., they must be free of design considerations
- Functional requirements are incomplete unless they capture all **relevant** aspects of the software's environment
  - they define the interactions between the software and the environment
  - the environment may consist of users, other systems, support hardware, operating system, etc.
  - the system/environment boundary must be defined



# Non-Functional Requirements

---

- Non-functional requirements place restrictions on the range of acceptable solutions
- Non-functional requirements cover a broad range of issues
  - interface constraints
  - performance constraints
  - operating constraints
  - life-cycle constraints
  - economic constraints
  - political constraints
  - manufacturing



# Software Requirements Specification (SRS)

---

- Point of origin
  - elicitation and/or allocation activity
- Purpose
  - provide a baseline for all software development activities
- Focus
  - software/environment interactions
  - technical reformulation of constraints
- Nature
  - highly technical
- Usage
  - design
  - testing
  - technical studies



# Sample SRS Table of Contents

---

1. Introduction (ANSI/IEEE STD-830-1984)
2. General description
3. Specific requirements
  - 3.1 Functional requirements
    - input/processing/output
  - 3.2 External interface requirements
    - interface specification
4. Performance requirements (non-functional)
5. Design constraints
6. Attributes
7. Other requirements



# Traceability

---

- Traceability is a property of the software development process
  - refers to the ability to relate elements of a specification to those design components that ensure their satisfiability
  - relations that are precise and narrow in scope simplify analysis
  - relations that are diffused often entail complex analysis
- Specifications may be
  - functional or non-functional
  - part of the system requirements or the byproduct of design
- Traceability is a useful tool, but not a substitute for verification
- Most traceability chains run through the software architecture
  - requirements to design
  - design to code



# Requirements Verification

---

- Requirements verification is an activity directed towards the discovery of specification errors
- The ultimate goal is to ensure that the specification (when considered on its own) is
  - correct
  - consistent
  - complete
- The verification must be carried out against a model (formal or informal)
- Formal and semi-formal specifications can be checked out by tools





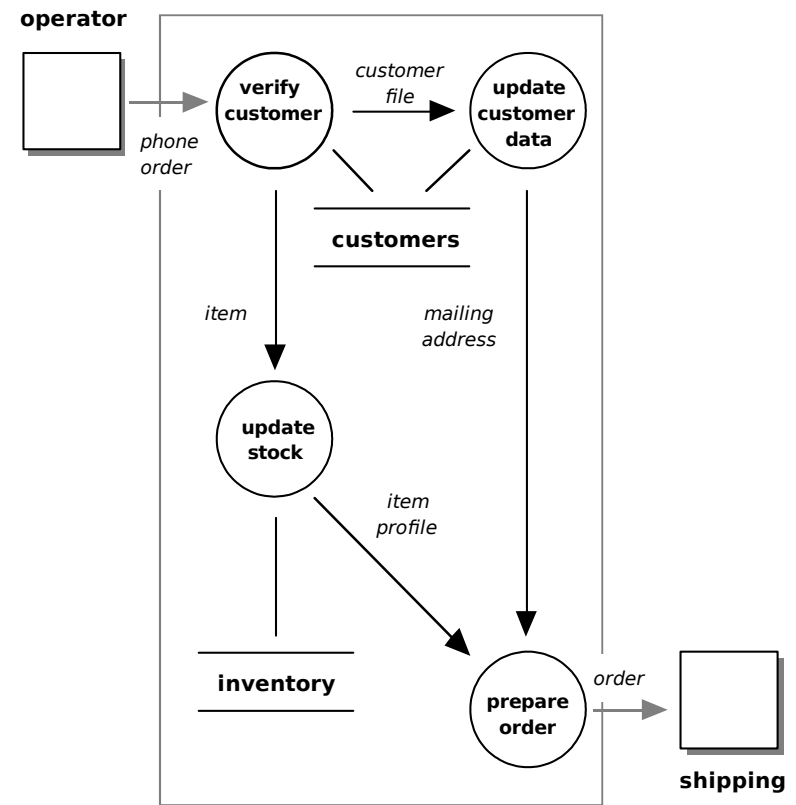
# Requirements Validation

---

- Concerned with establishing that specified requirements represent the needs of the customer and/or user
- Needs are not reflected by any model or document
  - Thus, validation cannot be performed in a mechanical way
- Good communication is the key to a successful validation
  - well-defined terminology
  - well-written and simple specifications
  - formal reviews
  - rapid prototypes
  - simulations

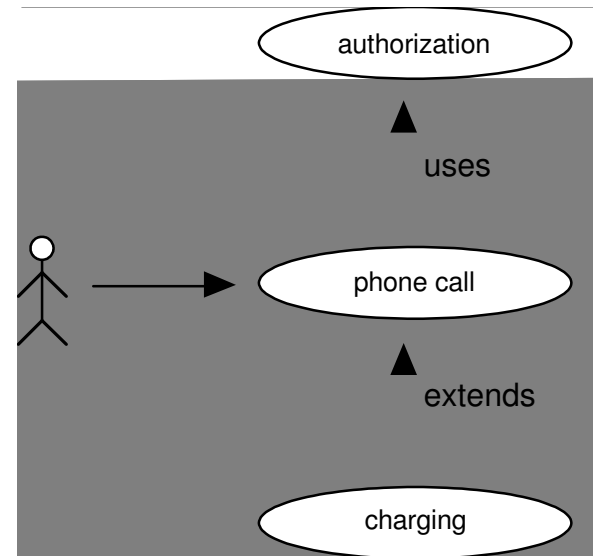
# Data Flow Models

- Dataflow diagram notation
  - functions—deterministic input/output **transformations** triggered by the presence of all needed inputs
  - **flows**—unbounded queues
  - **stores**—global system data
  - terminators—**interface** models
  - minispecs—**semantics** of the lowest level functions



# Use Case Models

- Actors
  - model the environment and the users
  - initiate activity by providing stimuli
  - can be primary or secondary
- Use cases
  - are complete courses of action initiated by actors (basic or alternative)
  - can be extended by (interrupt analogy) or use other use cases (call analogy)





# Architecture Design Phase

---

## Technical goals

- Identify the principal components that make up the system and their (internal and external) interfaces
- Ensure compliance with respect to expectations captured in the requirements specification
- Understand, assess, and control risks
- Ensure predictability of the development process through accurate planning



# Concluding Remarks

---

- Development of software based systems is complex
  - multiple perspectives
  - different notations
  - significant and sophisticated analysis
- Requirements elicitation and specification should focus on communication among people
- Requirements must be traceable through the architecture to the code itself (and vice versa)
- The design process should focus mainly on fundamentals and on the creative activities

# 6. Software Lifecycle Models

A software lifecycle model is a standardised format for

- planning
- organising, and
- running

a new development project.

# Software Engineering Methods

- Methods usually provide a notation and vocabulary, procedures for performing identifiable tasks, and guidelines for checking both the process and the product
- These are categorized as
  - Heuristic methods
    - ✓ dealing with informal approaches.
  - Formal methods
    - ✓ dealing with mathematically based approaches.
  - Prototyping methods
    - ✓ dealing with software engineering approaches based on various forms of prototypings

# Heuristic Methods

- These are categories as
  - Structured methods
    - ✓ The system is built from a functional viewpoint, starting with a high-level view and progressively refining this into a more detailed design.
  - Data-oriented methods
    - ✓ the starting points are the data structures that a program manipulates rather than the function it performs
  - Object-oriented methods
    - ✓ The system is viewed as a collection of objects rather than functions
  - Domain-specific methods
    - ✓ includes specialized methods for developing systems which involve real-time, safety, or security aspects

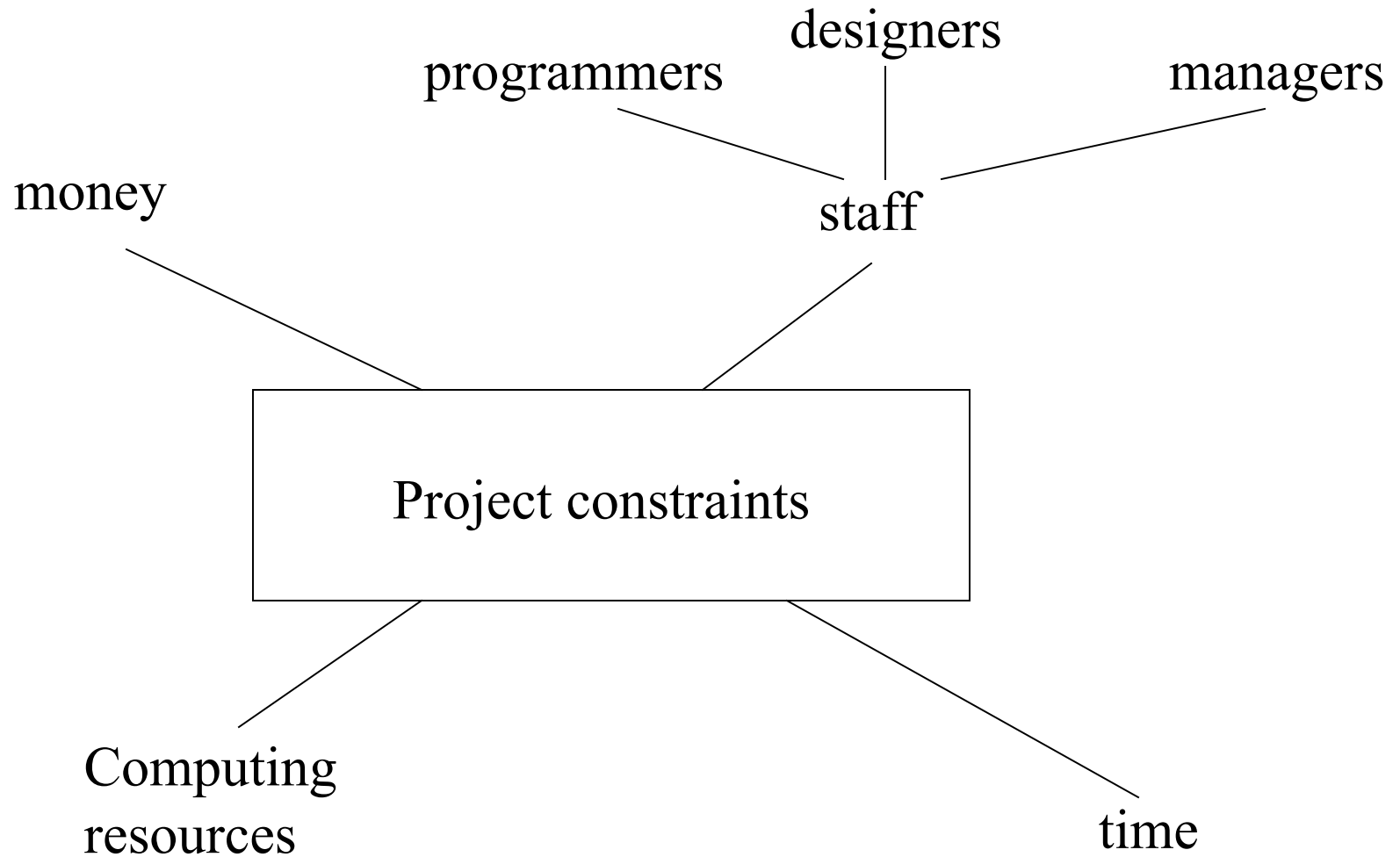


# Formal Methods

- These are categorized as
  - Specification languages and notations
    - ✓ This topic concerns the specification notation or language used. Specification languages can be classified as model-oriented, property-oriented, or behavior-oriented
  - Refinement
    - ✓ This topic deals with how the method refines (or transforms) the specification into a form which is closer to the desired final form of an executable program.
  - Verification/Proving properties:
    - ✓ This topic covers the verification properties that are specific to the formal approach, including both theorem proving and model checking

# Prototyping Methods

- These are categorized as
  - Prototyping styles
    - ✓ The prototyping styles topic identifies the various approaches: throwaway, evolutionary, and executable specification
  - Prototyping targets
    - ✓ Examples of the targets of a prototyping method may be requirements, architectural design, or the user interface
  - Prototyping evaluation techniques
    - ✓ This topic covers the ways in which the results of a prototype exercise are used.



# Examples of Project Constraints

A project plan contains much information,  
but must at least describe:

- resources needed  
*(people, money, equipment, etc)*
- dependency & timing of work  
*(flow graph, work packages)*
- rate of delivery *(reports, code, etc)*

It is impossible to measure rate of progress  
except with reference to a plan.

## 6.3. What is a Lifecycle Model?

### **Definition.**

A (software/system) *lifecycle model* is a description of the sequence of activities carried out in an SE project, and the relative order of these activities.

There are hundreds of different lifecycle models to choose from, e.g:

- *waterfall*,
- *code-and-fix*
- *spiral*
- *rapid prototyping*
- *unified process (UP)*
- *agile methods, extreme programming (XP)*
- *COTS ...*

but many are minor variations on a smaller number of basic models.

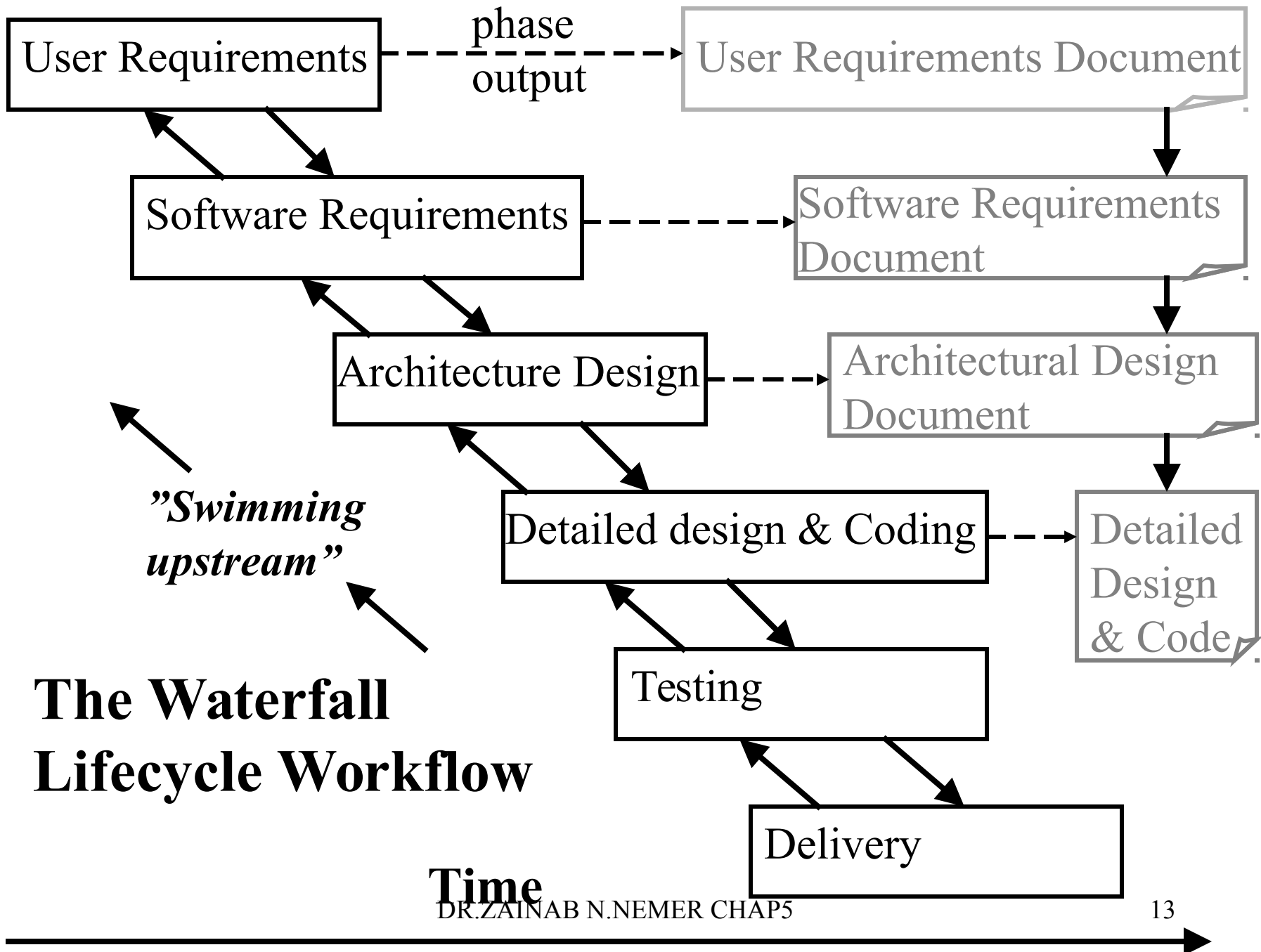


## 6.4. The Waterfall Model

- The waterfall model is the classic lifecycle model – it is widely known, understood and (commonly?) used.
- In some respect, waterfall is the "common sense" approach.
- Introduced by Royce 1970.







# Advantages

1. Easy to understand and implement.
2. Widely used and known (in theory!)
3. Reinforces good habits: define-before- design, design-before-code
4. Identifies deliverables and milestones
5. Document driven, URD, SRD, ... etc. Published documentation standards, e.g. PSS-05.
6. Works well on mature products and weak teams.

# Disadvantages I

1. Idealised, doesn't match reality well.
2. Doesn't reflect iterative nature of exploratory development.
3. Unrealistic to expect accurate requirements so early in project
4. Software is delivered late in project, delays discovery of serious errors.

# Disadvantages II

- 5. Difficult to integrate risk management
- 6. Difficult and expensive to make changes to documents, "swimming upstream".
- 7. Significant administrative overhead, costly for small teams and projects.

# Advantages

1. No administrative overhead
2. Signs of progress (code) early.
3. Low expertise, anyone can use it!
4. Useful for small “*proof of concept*” projects, e.g. as part of risk reduction.

# Disadvantages

1. Dangerous!
  1. No visibility/control
  2. No resource planning
  3. No deadlines
  4. Mistakes hard to detect/correct
2. Impossible for large projects,  
communication breakdown, chaos.

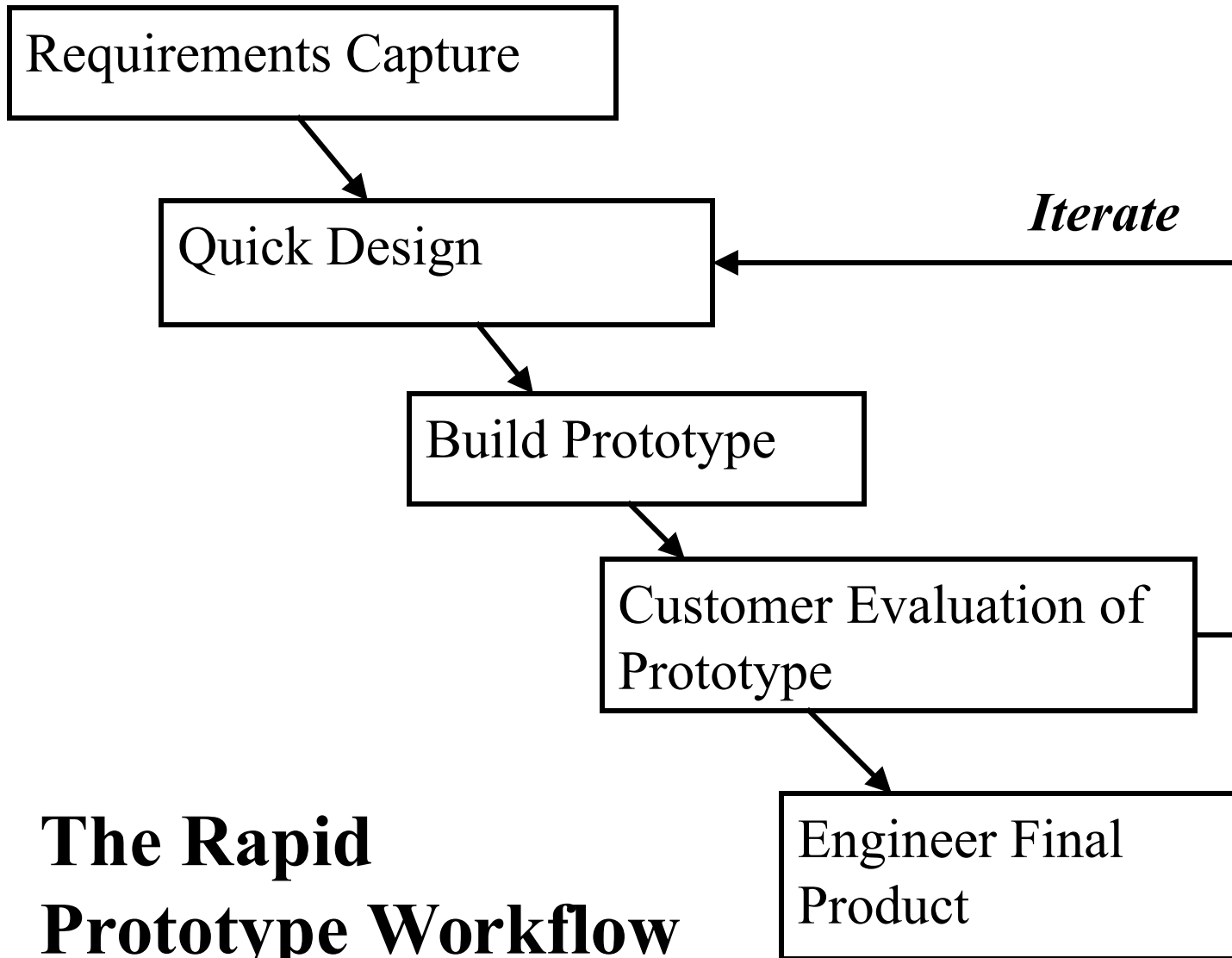
## 6.7. Rapid Prototyping

**Key idea:** Customers are non-technical and usually don't know what they want/can have.

Rapid prototyping emphasises requirements analysis and validation, also called:

- *customer oriented development,*
- *evolutionary prototyping*





## **The Rapid Prototype Workflow**

# Advantages

1. Reduces risk of incorrect user requirements
2. Good where requirements are changing/uncommitted
3. Regular visible progress aids management
4. Supports early product marketing

# Disadvantages I

1. An unstable/badly implemented prototype often becomes the final product.
2. Requires extensive customer collaboration
  - Costs customers money
  - Needs committed customers
  - Difficult to finish if customer withdraws
  - May be too customer specific, no broad market

# Disadvantages II

3. Difficult to know how long project will last
4. Easy to fall back into code-and-fix without proper requirements analysis, design, customer evaluation and feedback.

# Agile (XP) Manifesto

XP = Extreme Programming emphasises:

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

## **6.8.1. Agile Principles (Summary)**

- Continuous delivery of software
- Continuous collaboration with customer
- Continuous update according to changes
- Value participants and their interaction
- Simplicity in code, satisfy the spec

## 6.9. XP Practices (Summary)

- Programming in pairs
- Test driven development
- Continuous planning, change , delivery
- Shared project metaphors, coding standards and ownership of code
- No overtime! (Yeah right!)

# Advantages

- Lightweight methods suit small-medium size projects
- Produces good team cohesion
- Emphasises final product
- Iterative
- Test based approach to requirements and quality assurance



# Disadvantages

- Difficult to scale up to large projects where documentation is essential
- Needs experience and skill if not to degenerate into code-and-fix
- Programming pairs is costly
- Test case construction is a difficult and specialised skill.

## 6.11. COTS

- **COTS =**  
**Commercial Off-The-Shelf software**
- Engineer together a solution from existing commercial software packages using minimal software *”glue”*.
- E.g. using databases, spread sheets, word proccessors, graphics software, web browsers, etc.

## **Advantages**

- Fast, cheap solution
- May give all the basic functionality
- Well defined project, easy to run

## **Disadvantages**

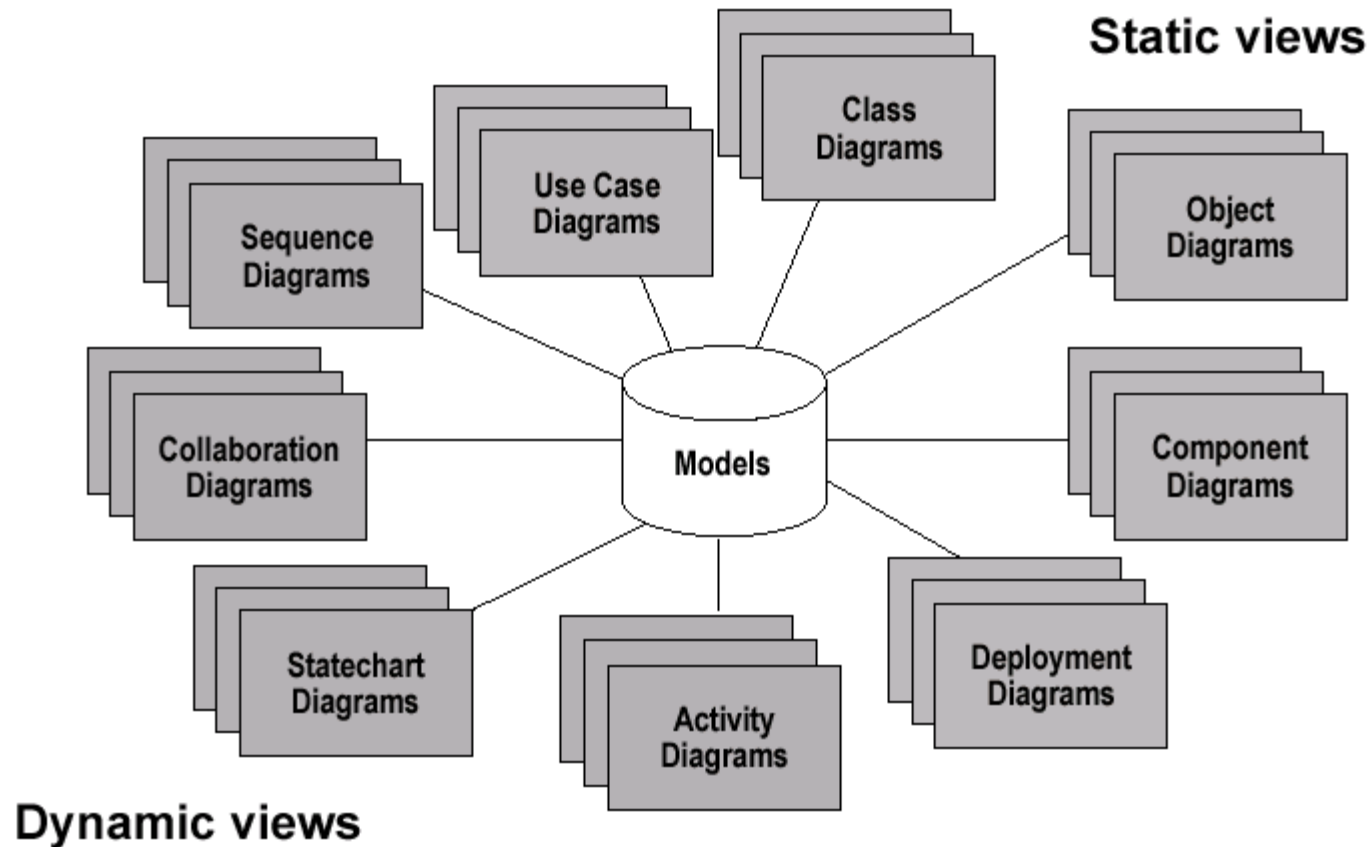
- Limited functionality
- Licensing problems, freeware, shareware, etc.
- License fees, maintainance fees, upgrade compatibility problems

# Introduction to UML

# What is UML?

- Unified Modeling Language
- UML is a modeling language to express and design documents, software
  - Independent of implementation language

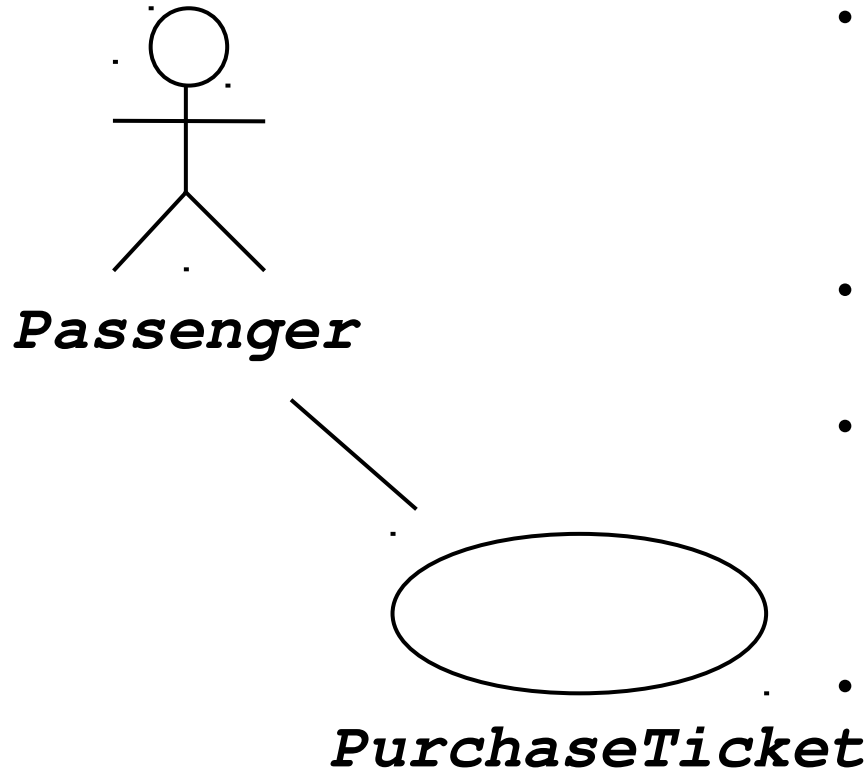
# Models, Views, Diagrams



# UML Baseline

- Use Case Diagrams
- Class Diagrams
- Package Diagrams
- Interaction Diagrams
  - Sequence
  - Collaboration
- Activity Diagrams
- State Transition Diagrams
- Deployment Diagrams

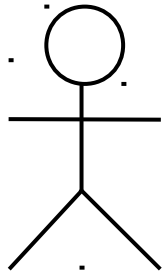
# Use Case Diagrams



- Used during requirements elicitation to represent external behavior
- ***Actors*** represent roles, that is, a type of user of the system
- ***Use cases*** represent a sequence of interaction for a type of functionality; summary of scenarios
- The use case model is the set of all use cases. It is a complete description of the functionality of the system and its environment



# Actors

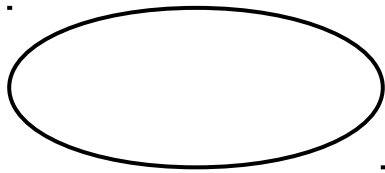


***Passenger***

- An actor models an external entity which communicates with the system:
  - User
  - External system
  - Physical environment
- An actor has a unique name and an optional description.
- Examples:
  - Passenger: A person in the train
  - GPS satellite: Provides the system with GPS coordinates

# Use Case

A use case represents a class of functionality provided by the system as an event flow.



***PurchaseTicket***

A use case consists of:

- Unique name
- Participating actors
- Entry conditions
- Flow of events
- Exit conditions
- Special requirements

# Use Case Diagram: Example

*Name: **Purchase ticket***

*Participating actor: **Passenger***

*Entry condition:*

- **Passenger** standing in front of ticket distributor.
- **Passenger** has sufficient money to purchase ticket.

*Exit condition:*

- **Passenger** has ticket.

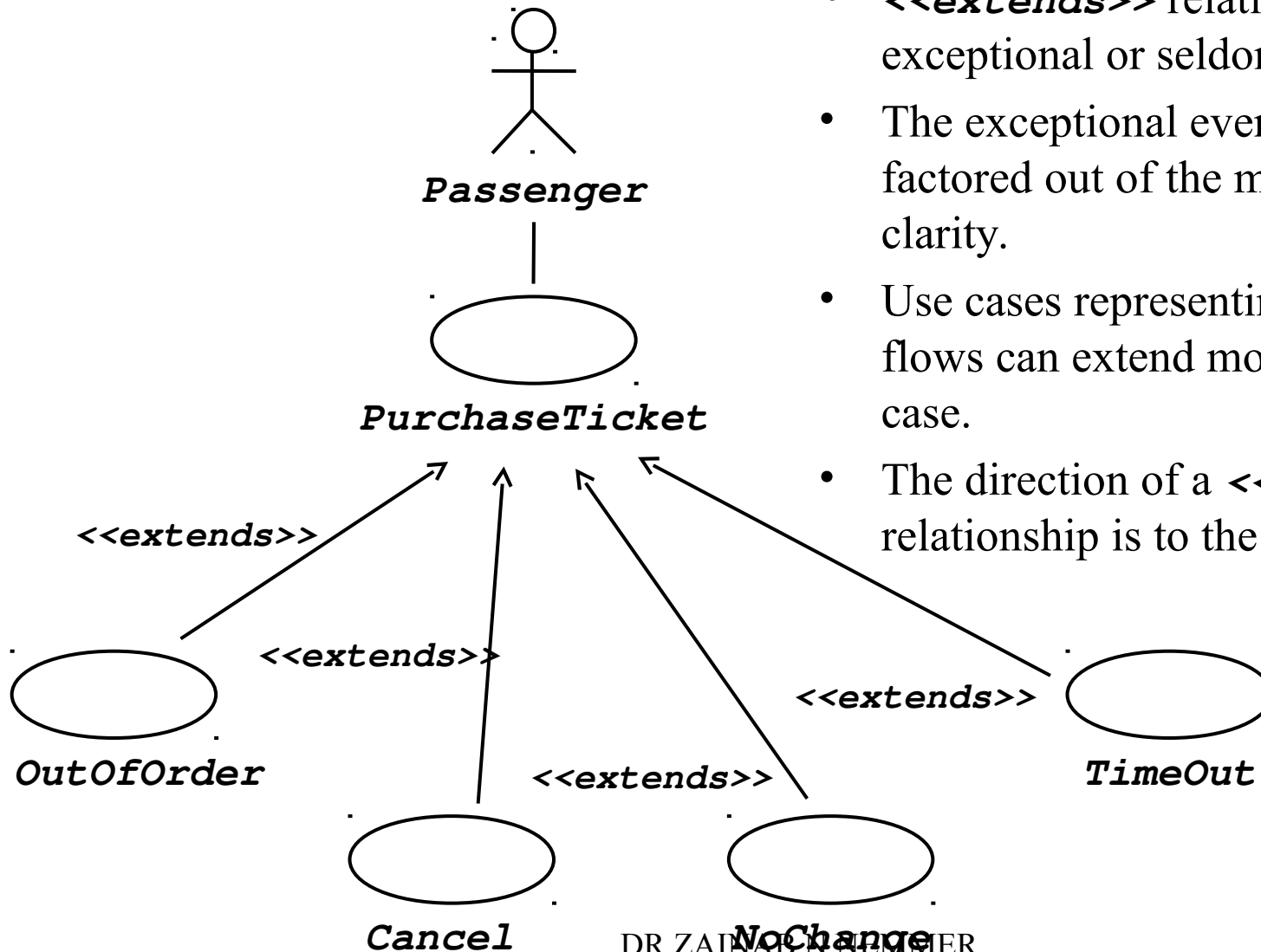
*Event flow:*

1. **Passenger** selects the number of zones to be traveled.
2. Distributor displays the amount due.
3. **Passenger** inserts money, of at least the amount due.
4. Distributor returns change.
5. Distributor issues ticket.

Anything missing?

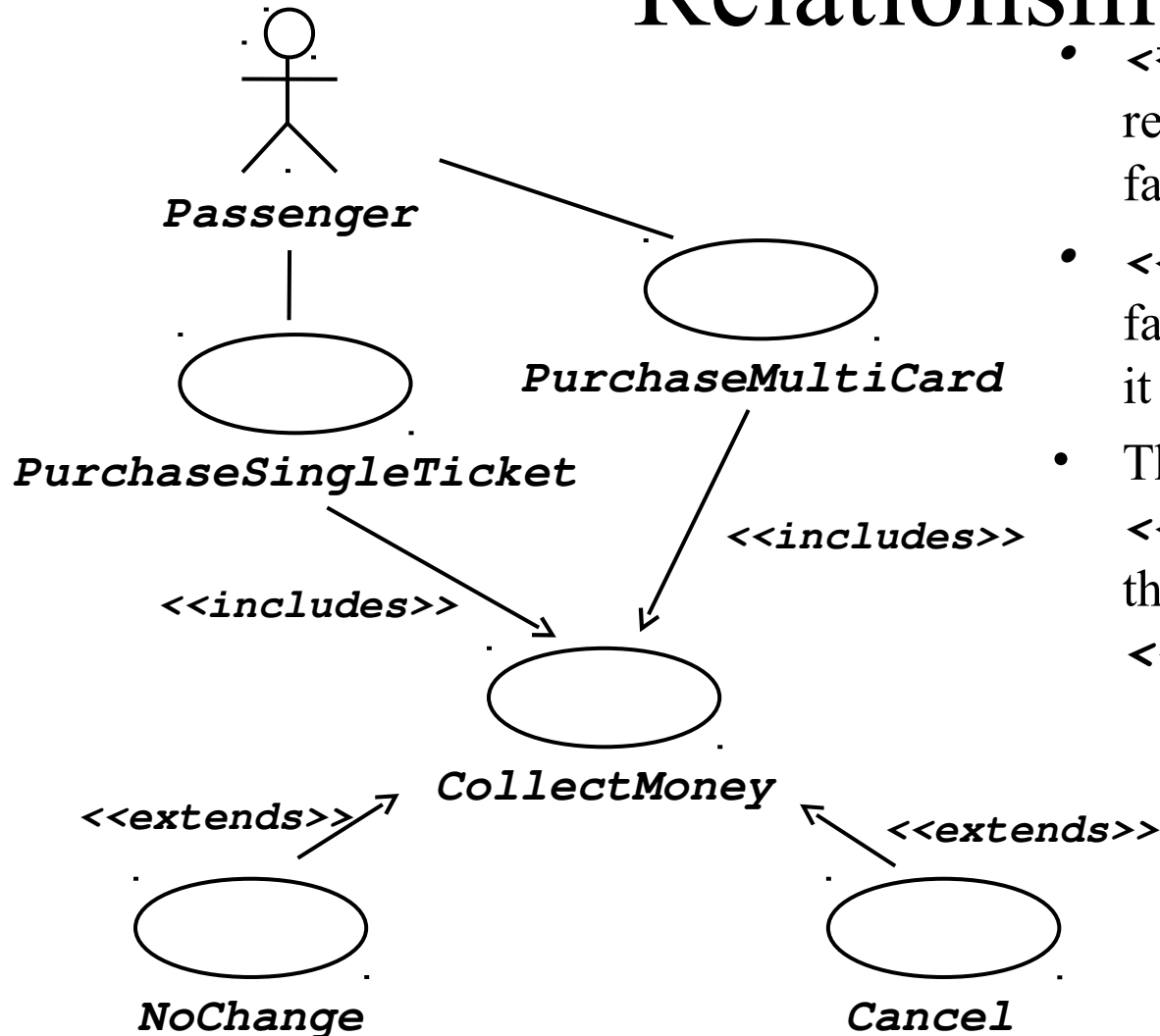
Exceptional cases!

# The <<extends>> Relationship



- <<extends>> relationships represent exceptional or seldom invoked cases.
- The exceptional event flows are factored out of the main event flow for clarity.
- Use cases representing exceptional flows can extend more than one use case.
- The direction of a <<extends>> relationship is to the extended use case

# The **<<includes>>** Relationship



- **<<includes>>** relationship represents behavior that is factored out of the use case.
- **<<includes>>** behavior is factored out for reuse, not because it is an exception.
- The direction of a **<<includes>>** relationship is to the using use case (unlike **<<extends>>** relationships).

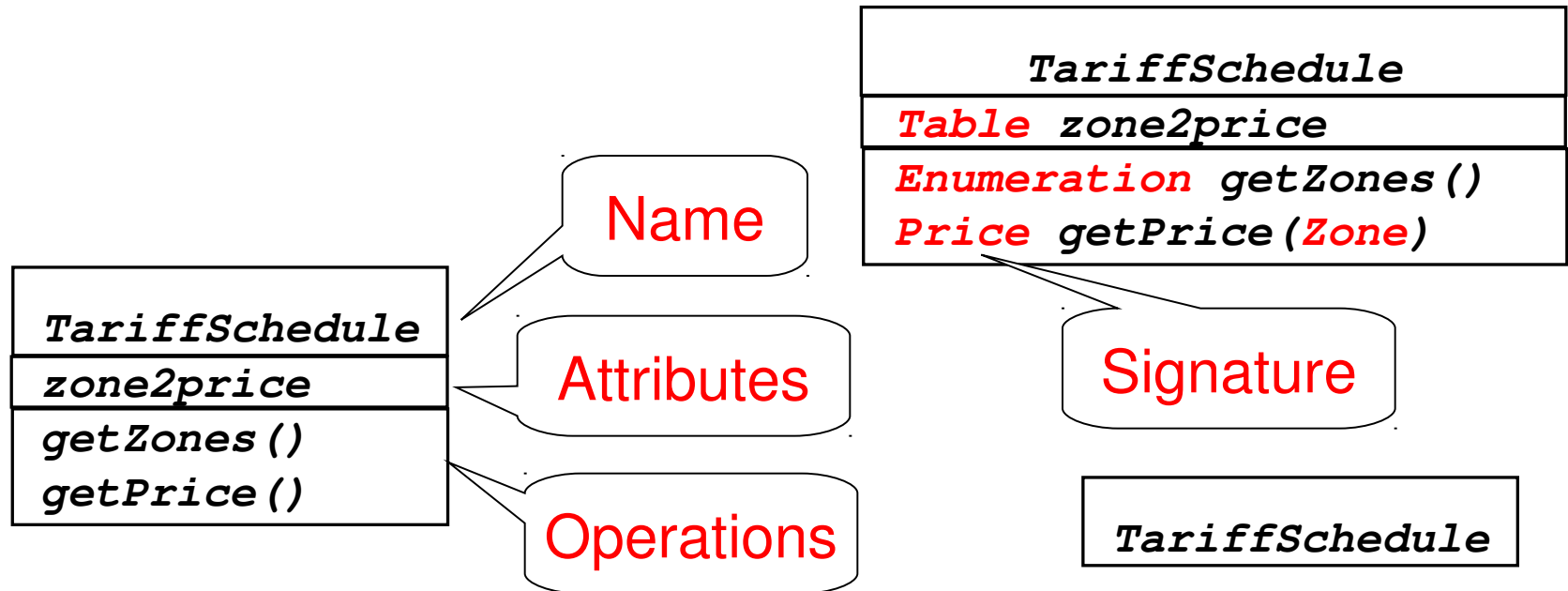
# Use Cases are useful for...

- Determining requirements
  - New use cases often generate new requirements as the system is analyzed and the design takes shape.
- Communicating with clients
  - Their notational simplicity makes use case diagrams a good way for developers to communicate with clients.
  - May require some explanation.
- Generating test cases
  - The collection of scenarios for a use case may suggest a suite of test cases for those scenarios.

# Class Diagrams

- Gives an overview of a system by showing its classes and the relationships among them.
  - Class diagrams are static
  - they display what interacts but not what happens when they do interact
- Also shows attributes and operations of each class
- Good way to describe the overall architecture of system components

# Classes – Not Just for Code



- A *class* represent a concept
- A class encapsulates state (*attributes*) and behavior (*operations*).
- Each attribute has a *type*.
- Each operation has a *signature*.
- The class name is the only mandatory information.



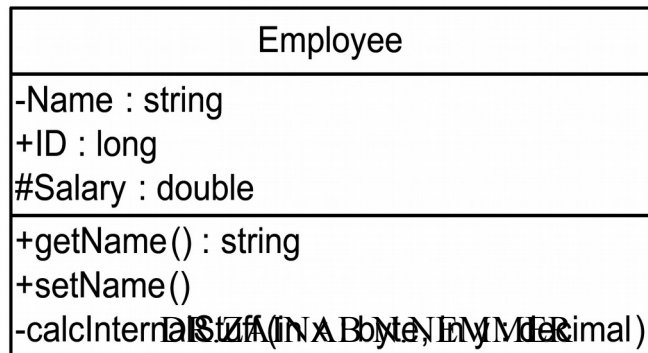
# Instances

<code>tarif_1974:TariffSchedule</code>
<code>zone2price = {   {'1', .20},   {'2', .40},   {'3', .60}}</code>

- An *instance* represents a phenomenon.
- The name of an instance is underlined and can contain the class of the instance.
- The attributes are represented with their *values*.

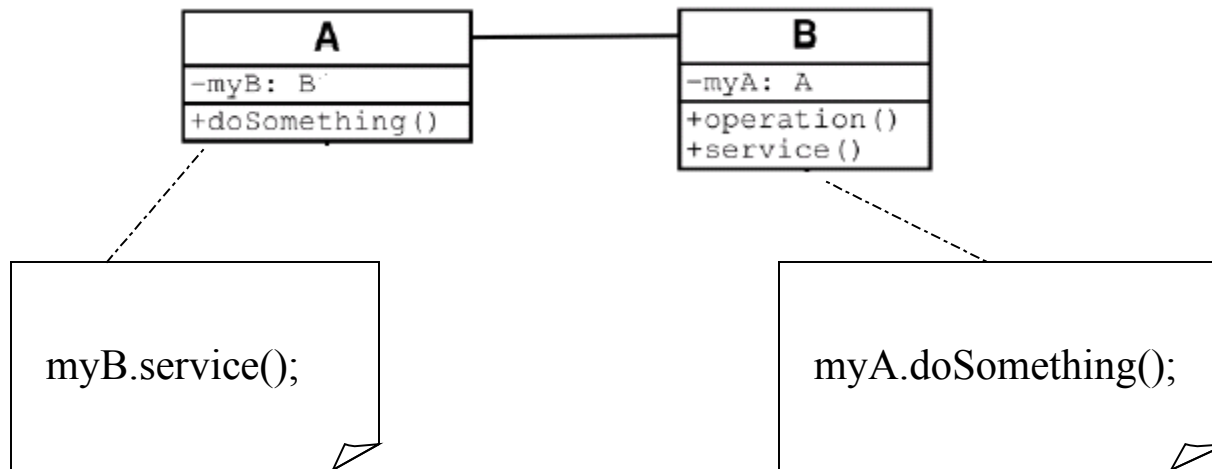
# UML Class Notation

- A class is a rectangle divided into three parts
  - Class name
  - Class attributes (i.e. data members, variables)
  - Class operations (i.e. methods)
- Modifiers
  - Private: -
  - Public: +
  - Protected: #
  - Static: Underlined (i.e. shared among all members of the class)
- Abstract class: Name in italics



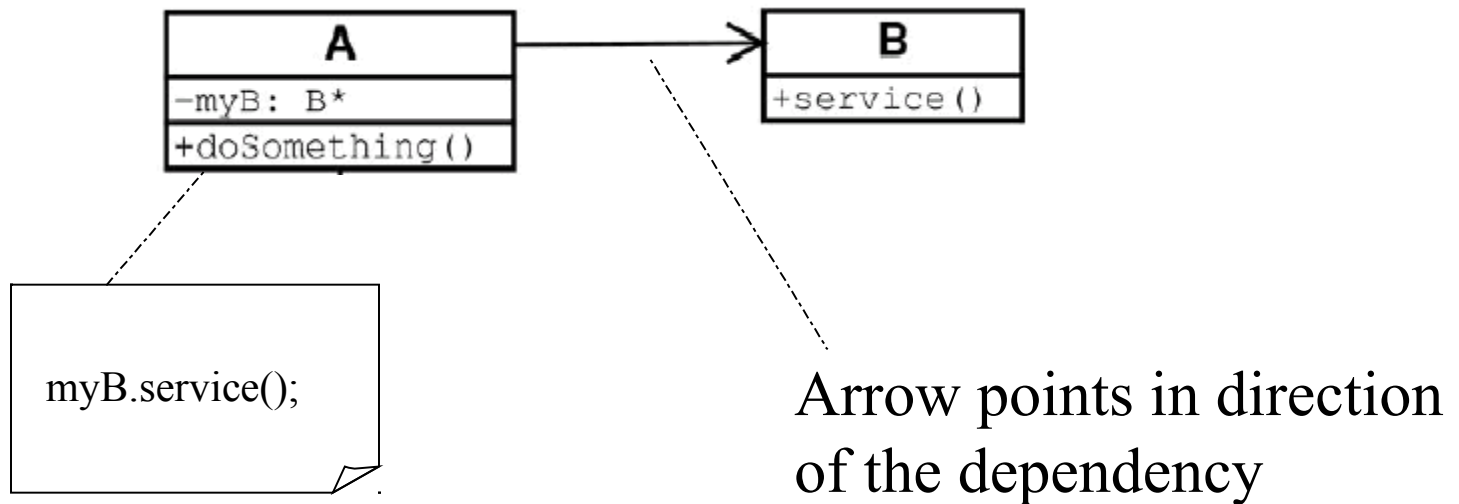
# Binary Association

Binary Association: Both entities “Know About” each other



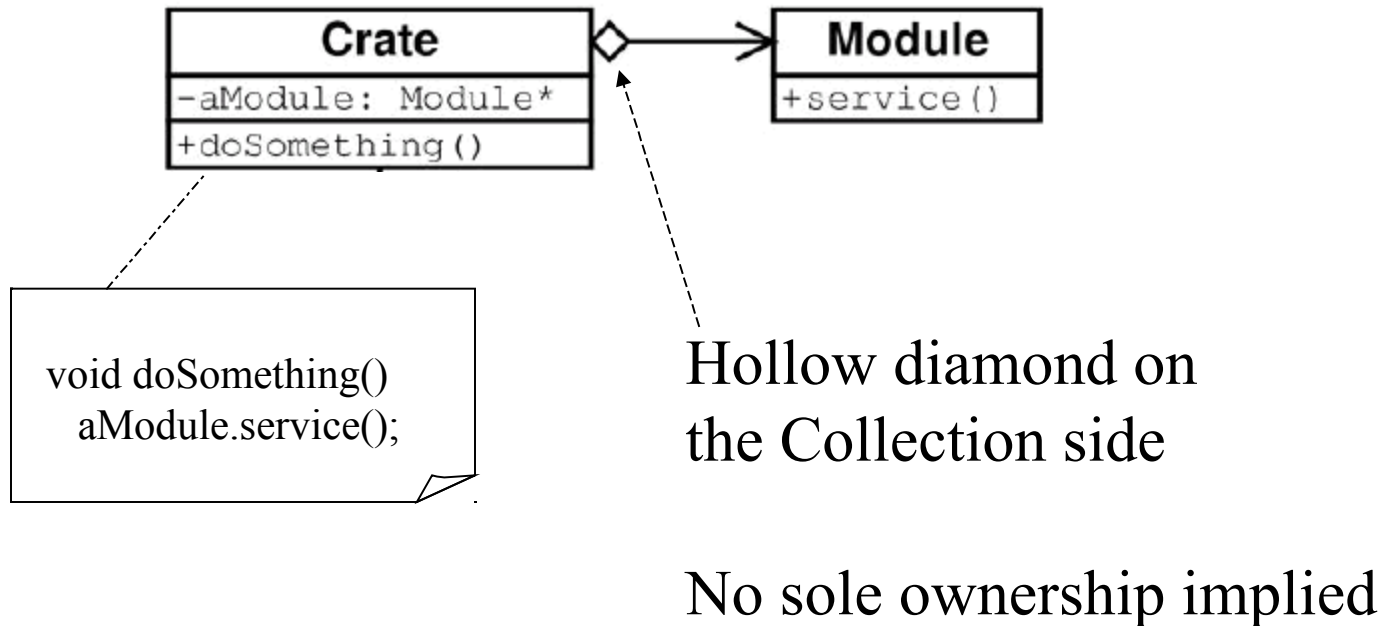
# Unary Association

A knows about B, but B knows nothing about A



# Aggregation

Aggregation is an association with a “collection-member” relationship

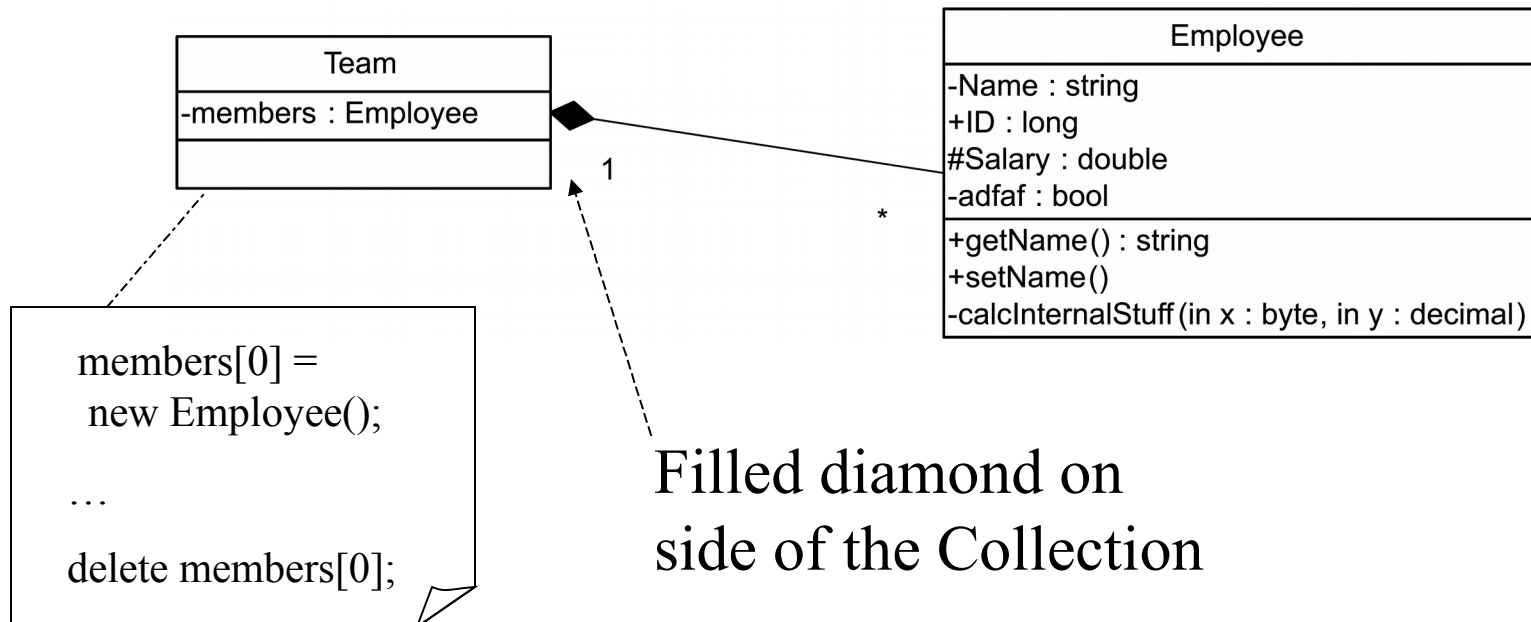


# Composition

Composition is Aggregation with:

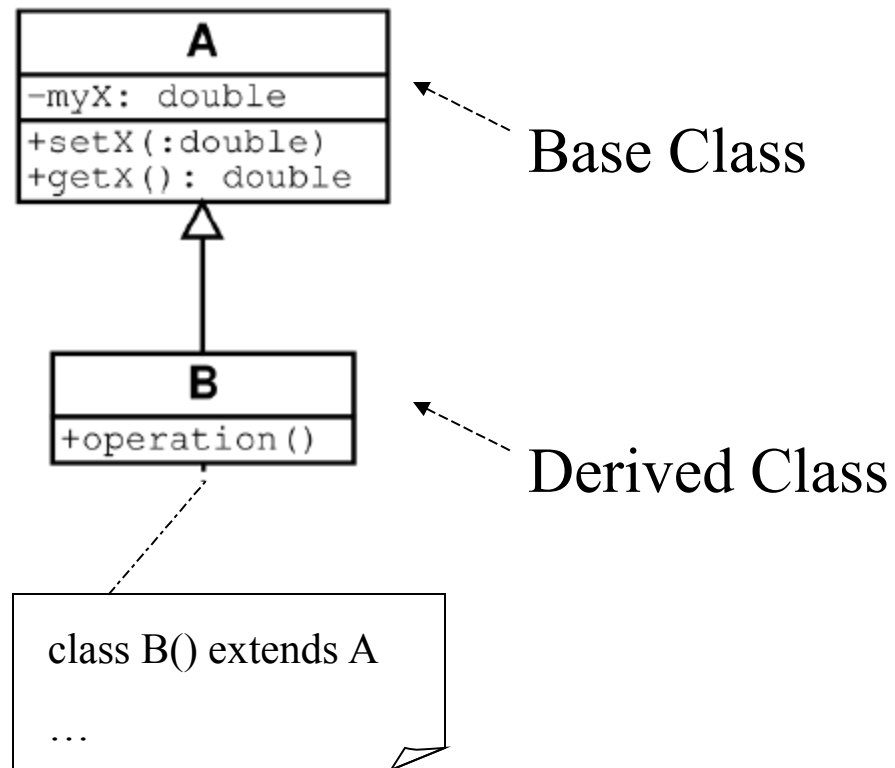
Lifetime Control (owner controls construction, destruction)

Part object may belong to only one whole object



# Inheritance

Standard concept of inheritance



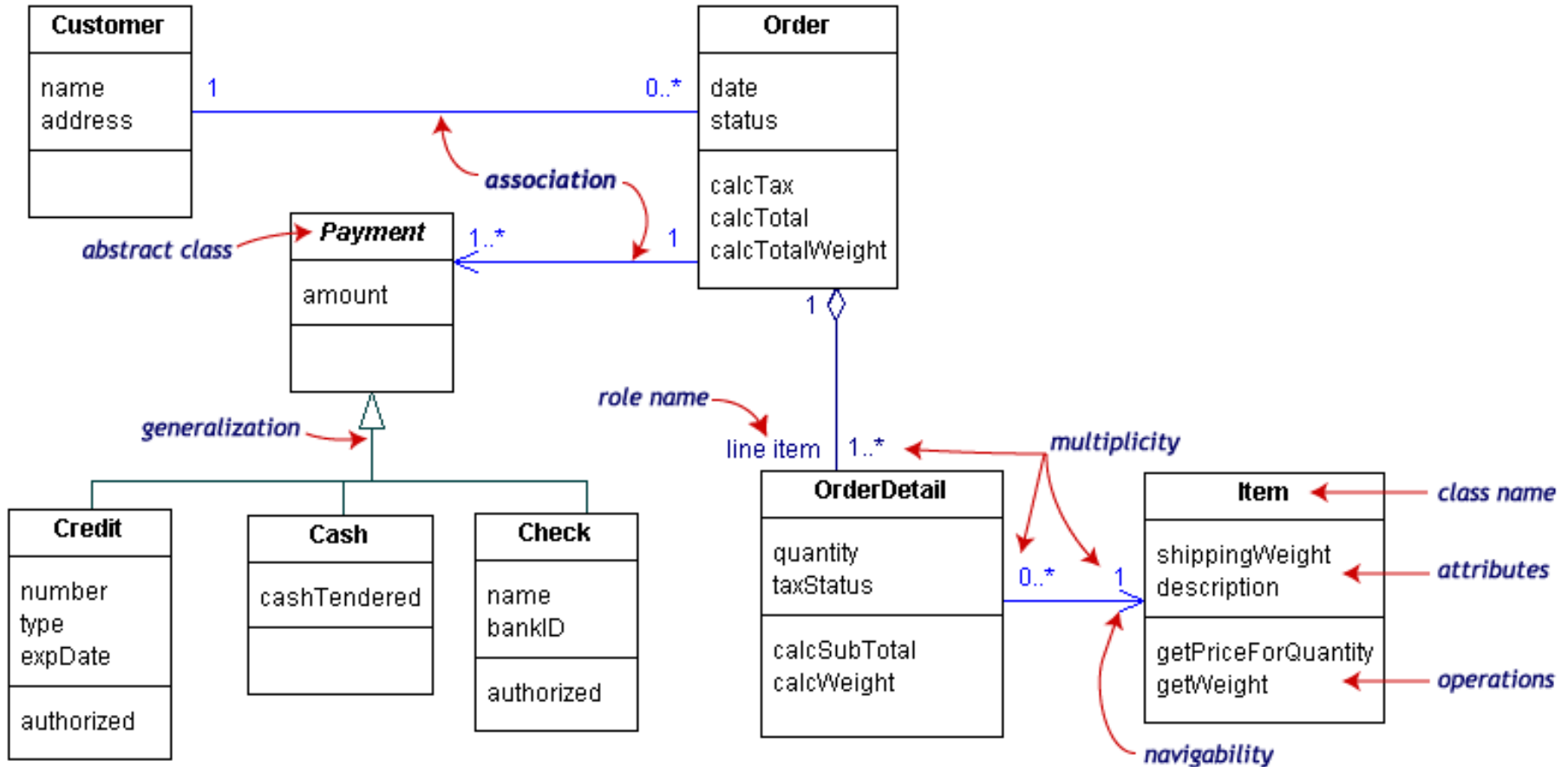
# UML Multiplicities

Links on associations to specify more details about the relationship

Multiplicities	Meaning
<b>0..1</b>	zero or one instance. The notation <b><i>n</i> .. <i>m</i></b> indicates <b><i>n</i></b> to <b><i>m</i></b> instances.
<b>0..*</b> <i>or</i> <b>*</b>	no limit on the number of instances (including none).
<b>1</b>	exactly one instance
<b>1..*</b>	at least one instance

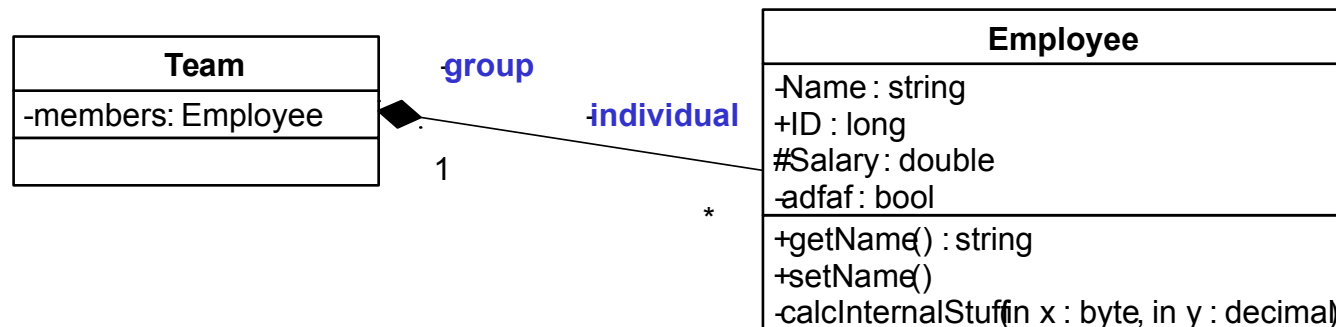


# UML Class Example



# Association Details

- Can assign names to the ends of the association to give further information



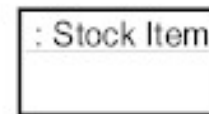
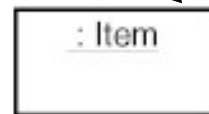
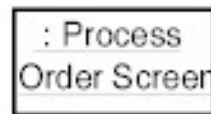
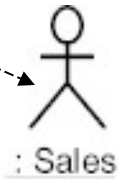
# Static vs. Dynamic Design

- Static design describes code structure and object relations
  - Class relations
  - Objects at design time
  - Doesn't change
- Dynamic design shows communication between objects
  - Similarity to class relations
  - Can follow sequences of events
  - May change depending upon execution scenario
  - Called Object Diagrams

# Sequence Diagram Format

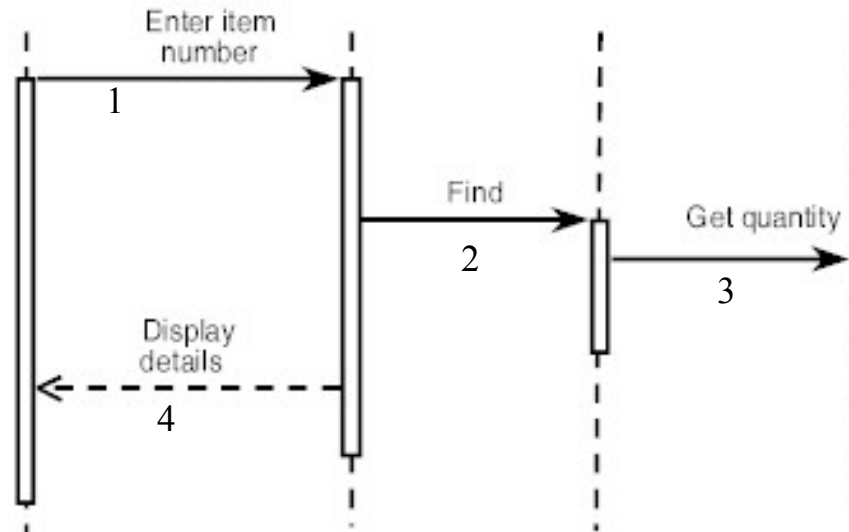
Actor from  
Use Case

Objects



Activation

Lifeline

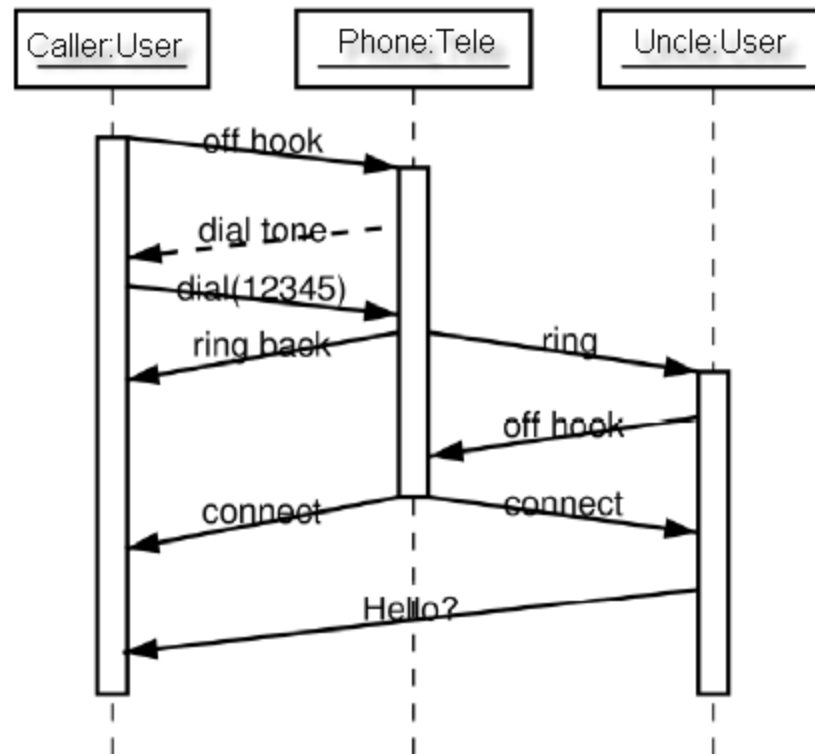


Calls = Solid Lines

Returns = Dashed Lines

# Sequence Diagram : Timing

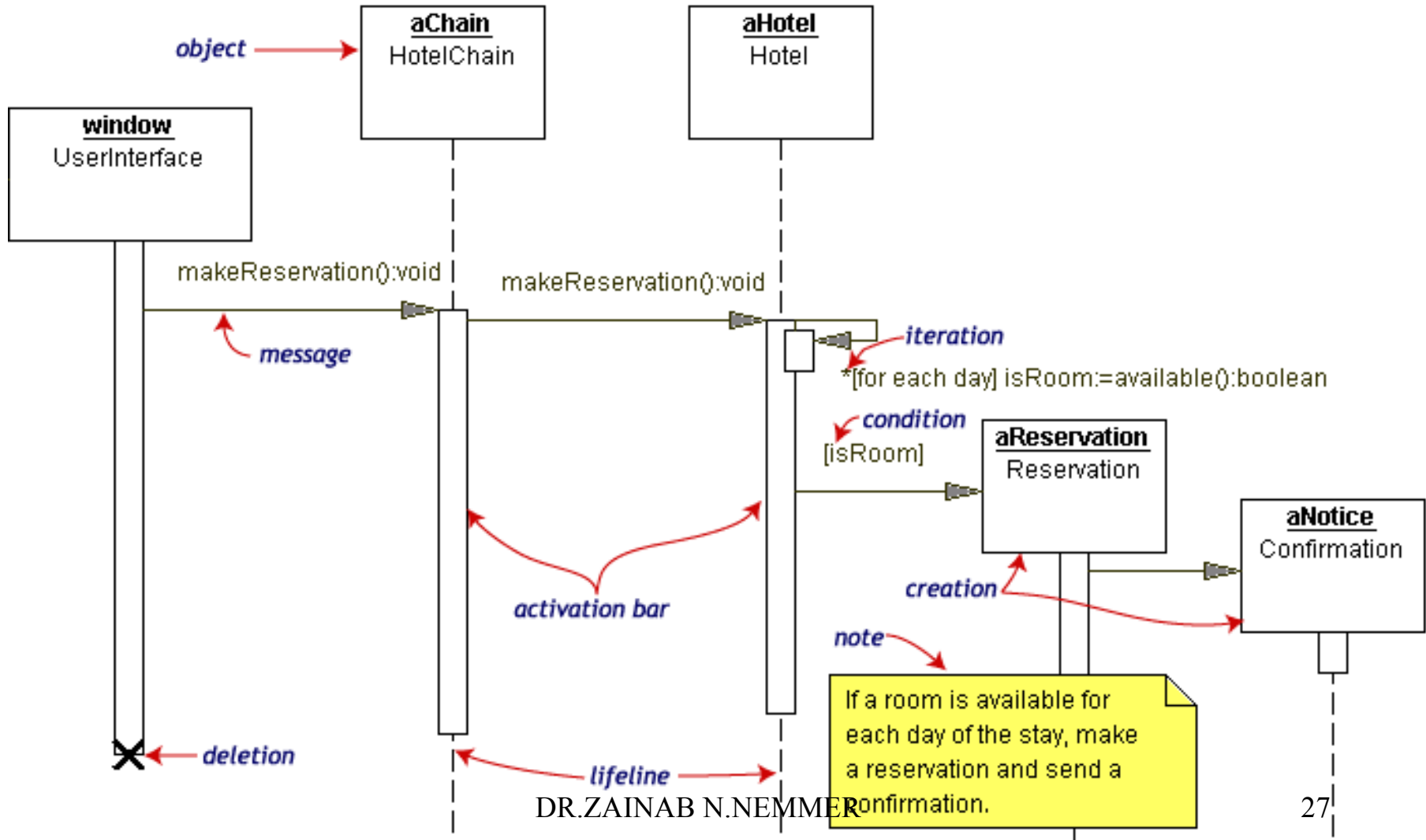
Slanted Lines show propagation delay of messages  
Good for modeling real-time systems





If messages cross this is usually problematic – race conditions

# Sequence Diagram Example

## Hotel Reservation

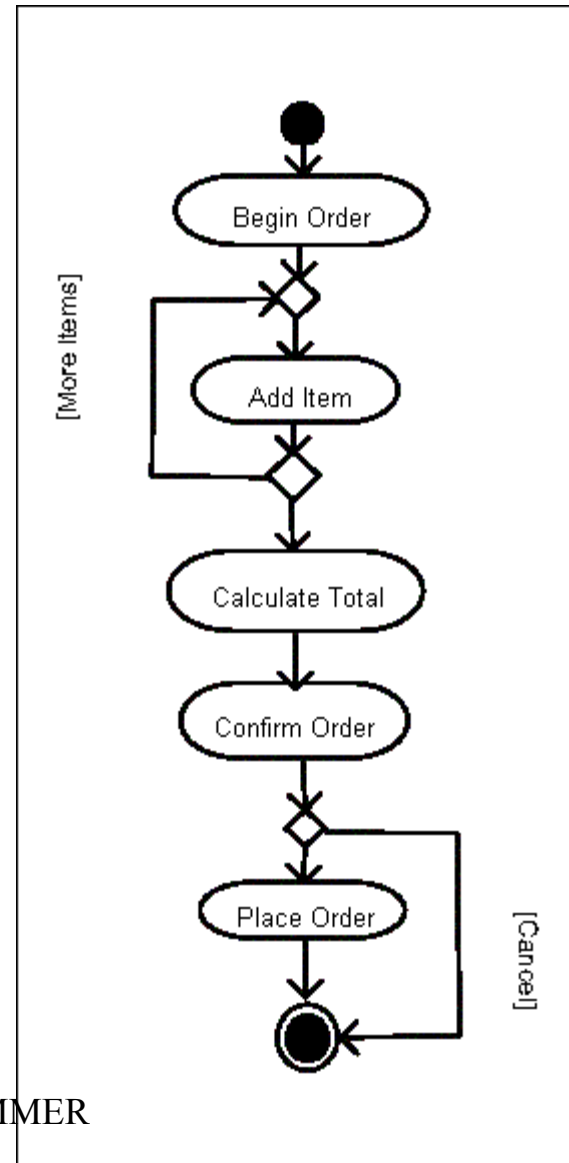


# Activity Diagrams

- Fancy flowchart
  - Displays the flow of activities involved in a single process
  - States
    - Describe what is being processed
    - Indicated by boxes with rounded corners
  - Swim lanes
    - Indicates which object is responsible for what activity
  - Branch
    - Transition that branch
    - Indicated by a diamond
  - Fork
    - Transition forking into parallel activities
    - Indicated by solid bars
  - Start and End
    -  

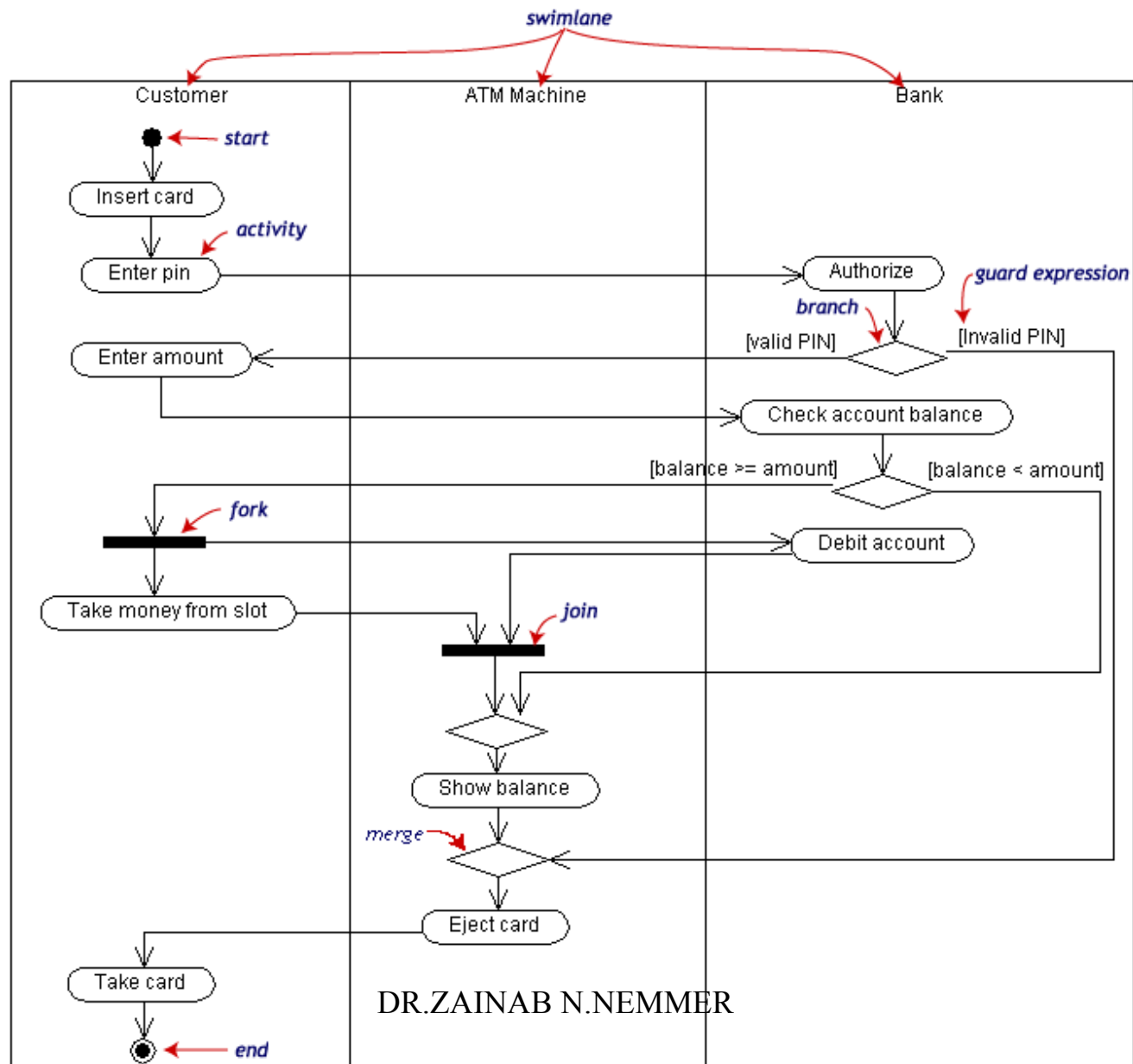
# Sample Activity Diagram

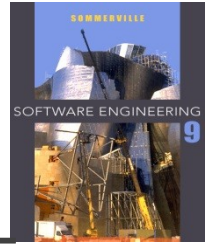
- Ordering System
- May need multiple diagrams from other points of view





# Activity Diagram Example





---

# STRUCTURAL AND BEHAVIOURAL MODELING

### 3. Structural Models

---

- ✂ **Structural models** of software display the **organization** of a system in terms of the **components** 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.
- ✂ Structural models are created when **discussing and designing the system architecture**.

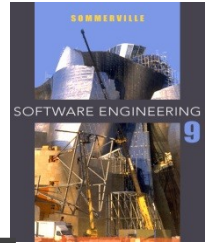
# Class Diagrams

---

- ✂ **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 **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, a device, etc.
- ✂ We have spoken in terms of 'nouns' or 'things.'

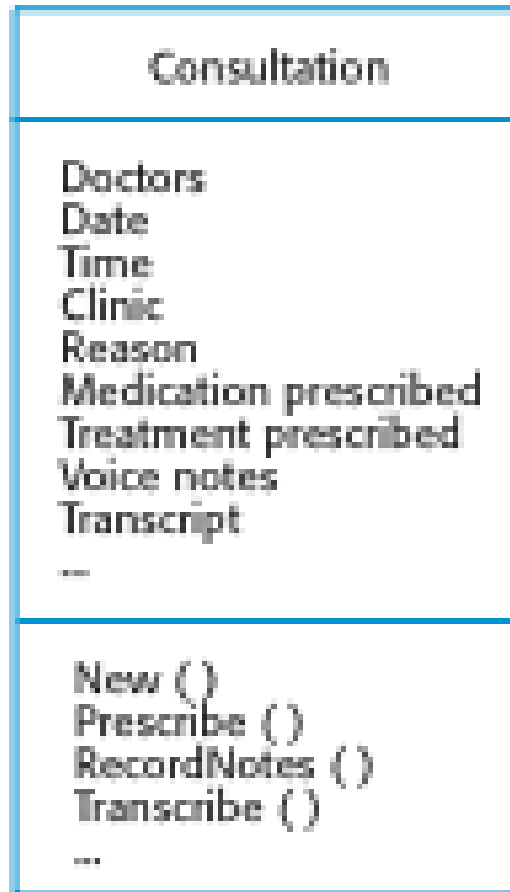
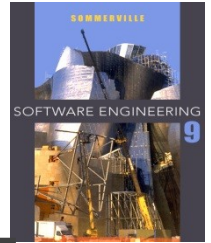
# UML classes and association

---



Here we are also showing multiplicity: one object of type Patient is related to one object of type Patient Record

# The Consultation Class (There are all kinds of classes: domain / entity classes; software classes, and many 'levels' of these!)



This is a high level class – perhaps a first or second cut at class definition.

Notice much is **missing**, such as parameters, returns, etc. from the methods.

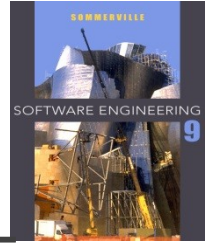
These are 'software classes' because they contain methods (that is, how the attributes will be used / manipulated.)

Finer levels of granularity are needed to address the shortcomings of this class.

But this is a very good way to start your analysis and class definition, especially when trying to develop classes from use cases, where nouns (Consultation) and verbs (new, prescribe....) will be found in the use case itself.

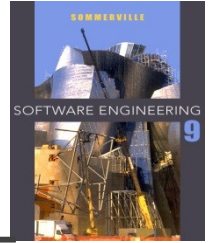
# Generalization

---



- ✂ **Generalization** is an everyday technique that we use to
- ✂ Rather than learn the detailed characteristics of every entity that we experience, we place these entities in more general classes (animals, cars, houses, etc.) and learn the characteristics of these classes.
- ✂ This allows us to infer that different members of these classes have some common characteristics e.g. squirrels and rats are rodents.
- ✂ In **programming**, we refer to this as 'inheritance.'
- ✂ Base class and derived classes, or super class and derived class.... parent and child.... many terms for capturing these important relationships.

# Generalization

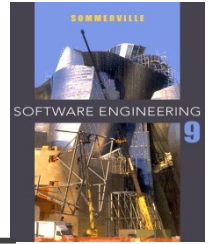


- ✂ In modeling systems, it is often useful to examine the classes in a system to see if there is scope for generalization.
- ✂ In object-oriented languages, such as Java, generalization is implemented using the class inheritance mechanisms built into the language.
- ✂ In a generalization, the attributes and operations associated with higher-level classes are also associated with (**inherited by**) 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**.



# Relationships: **Generalization**

---



A relationship among classes where one class shares the structure and/or behavior of one or more classes

Defines a hierarchy of abstractions in which a subclass inherits from one or more superclasses

- Single inheritance

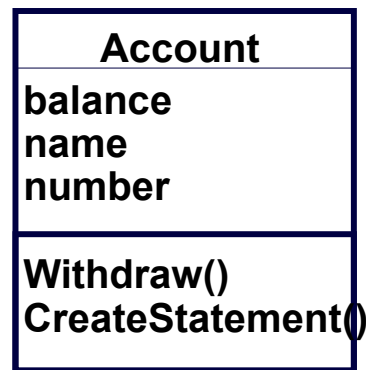
- Multiple inheritance

Generalization is an “is-a-kind of” relationship, or simply, “is\_a” relationship.

# Example: Single Inheritance

## One class inherits from another

Superclass  
(parent)



Generalization Relationship

Subclasses inherit both attributes and methods from base (parent) class.

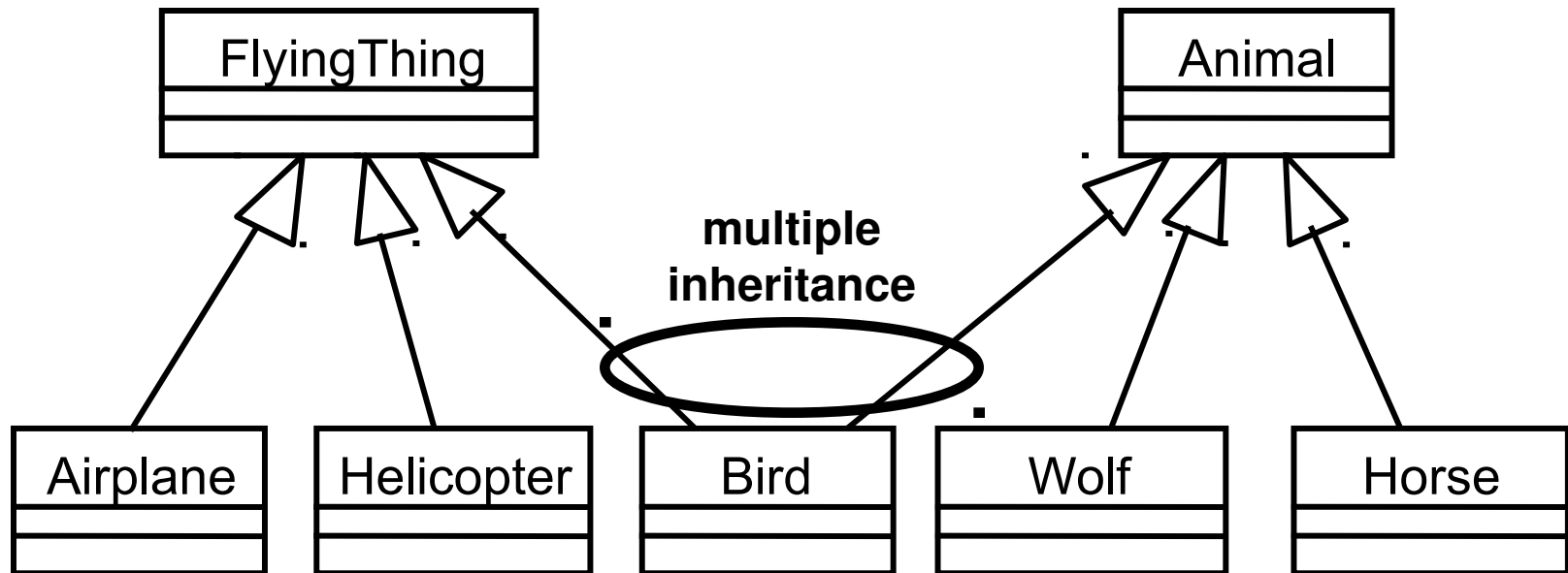
Subclasses



Descendents

# Example: Multiple Inheritance

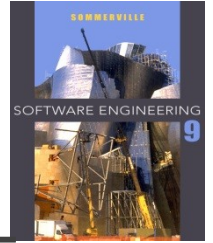
A class can inherit from several other classes



***Use multiple inheritance only when needed, and  
always with caution !***

# What Gets Inherited?

---



**A subclass inherits its parent's attributes, operations, and relationships**

**A subclass may:**

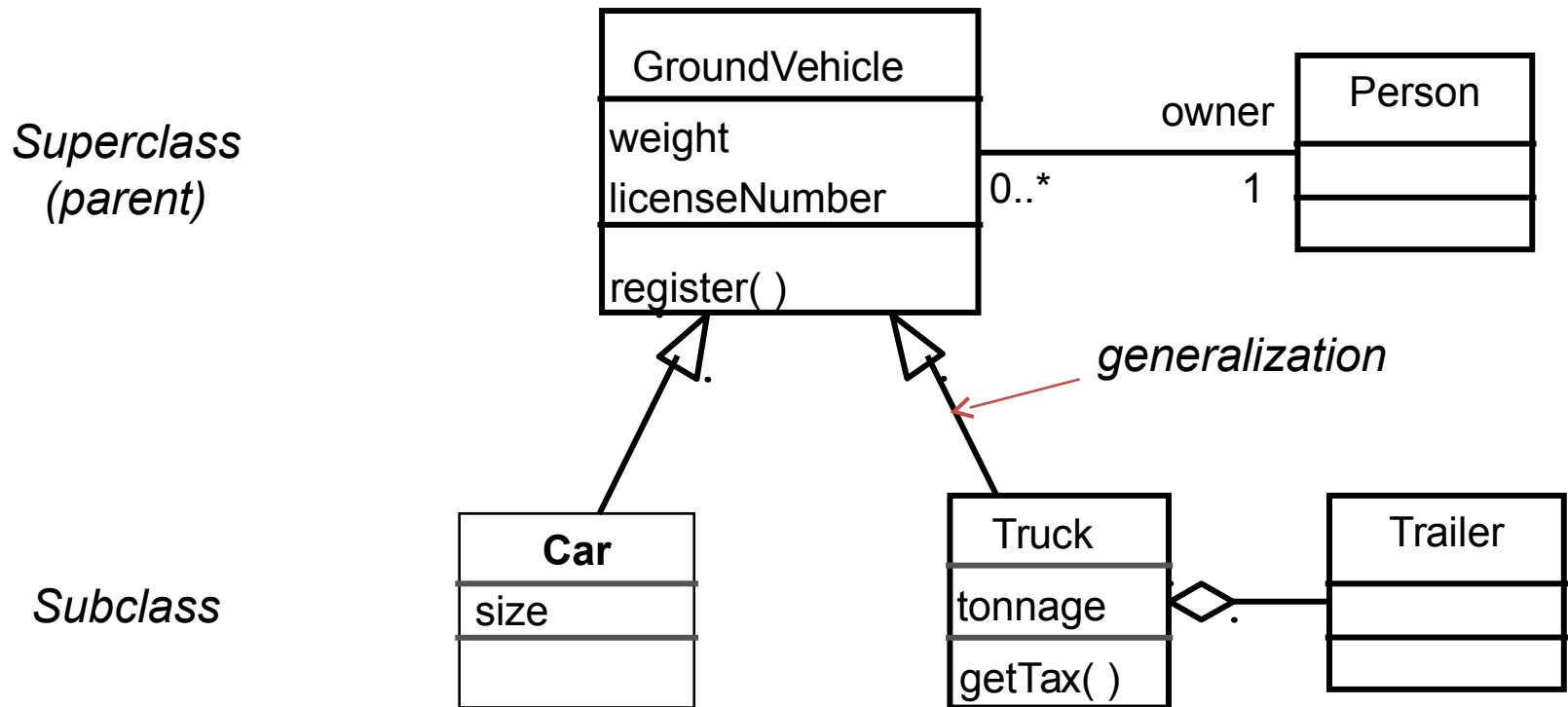
**Add additional attributes, operations, relationships**

**Redefine inherited operations (use caution!)**

**Common attributes, operations, and/or relationships are shown at the highest applicable level in the hierarchy**

*Inheritance leverages the similarities among classes*

# Example: What Gets Inherited

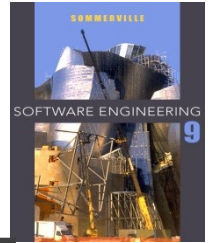


# Object class aggregation models

---

- ✂ 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.

# Relationships



Association

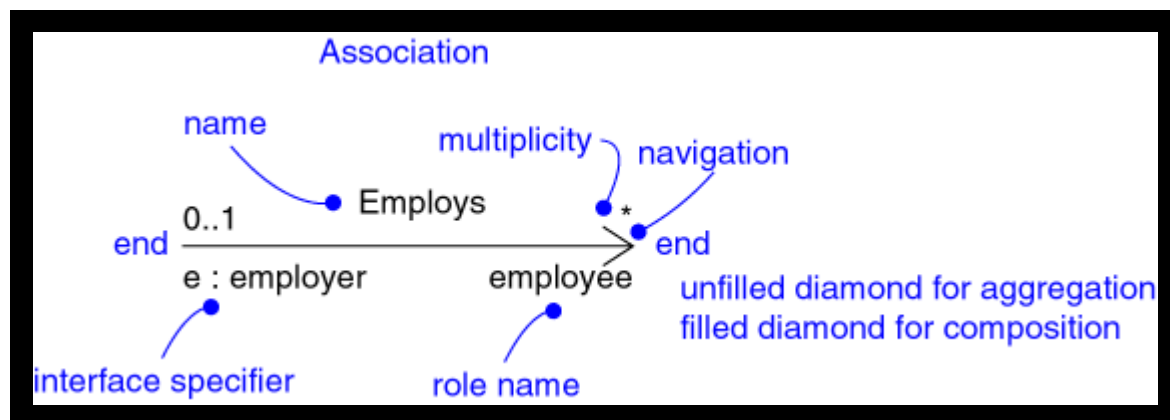
Aggregation

Composition

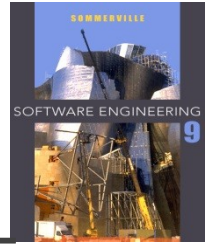
Dependency

Generalization

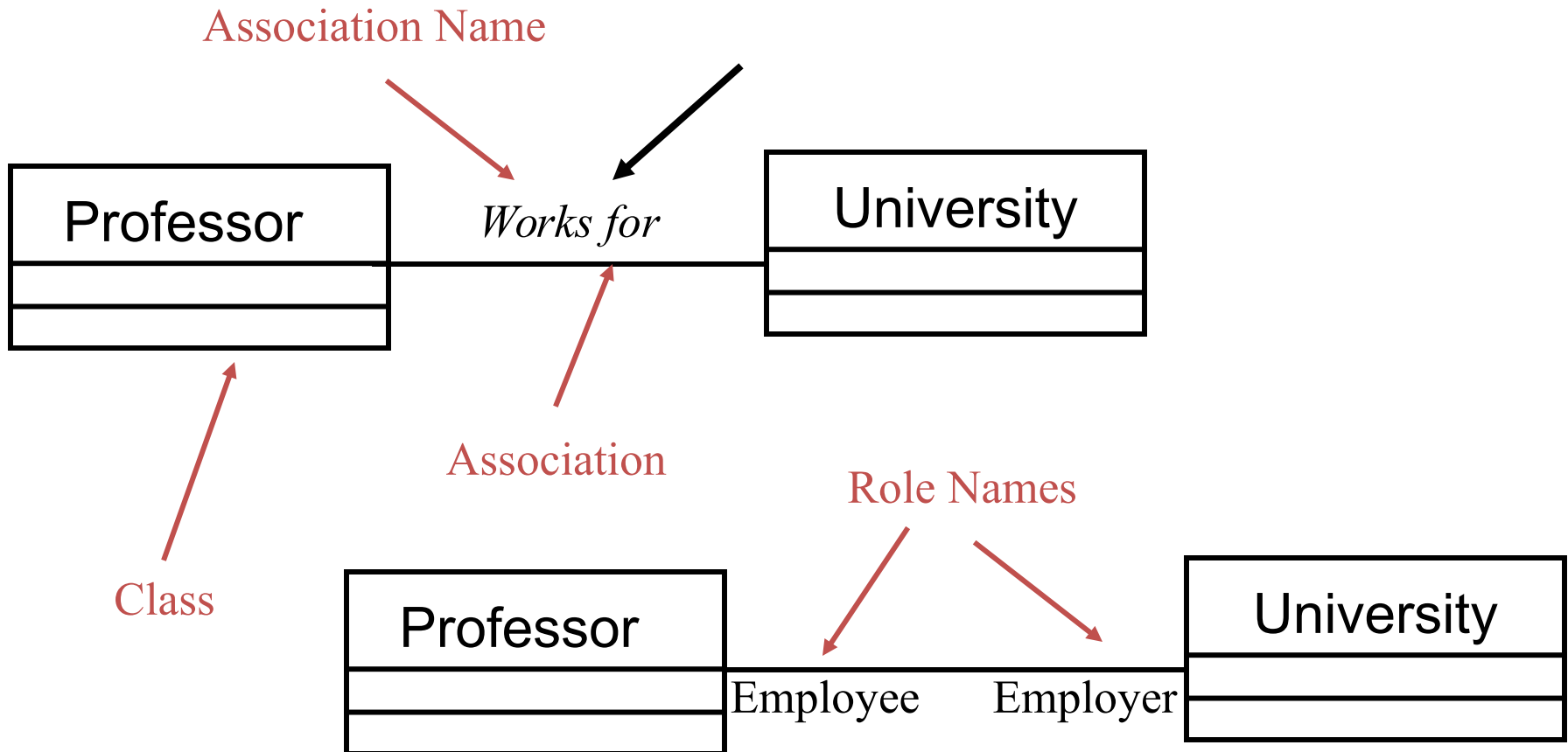
Realization



# Relationships: **Association**



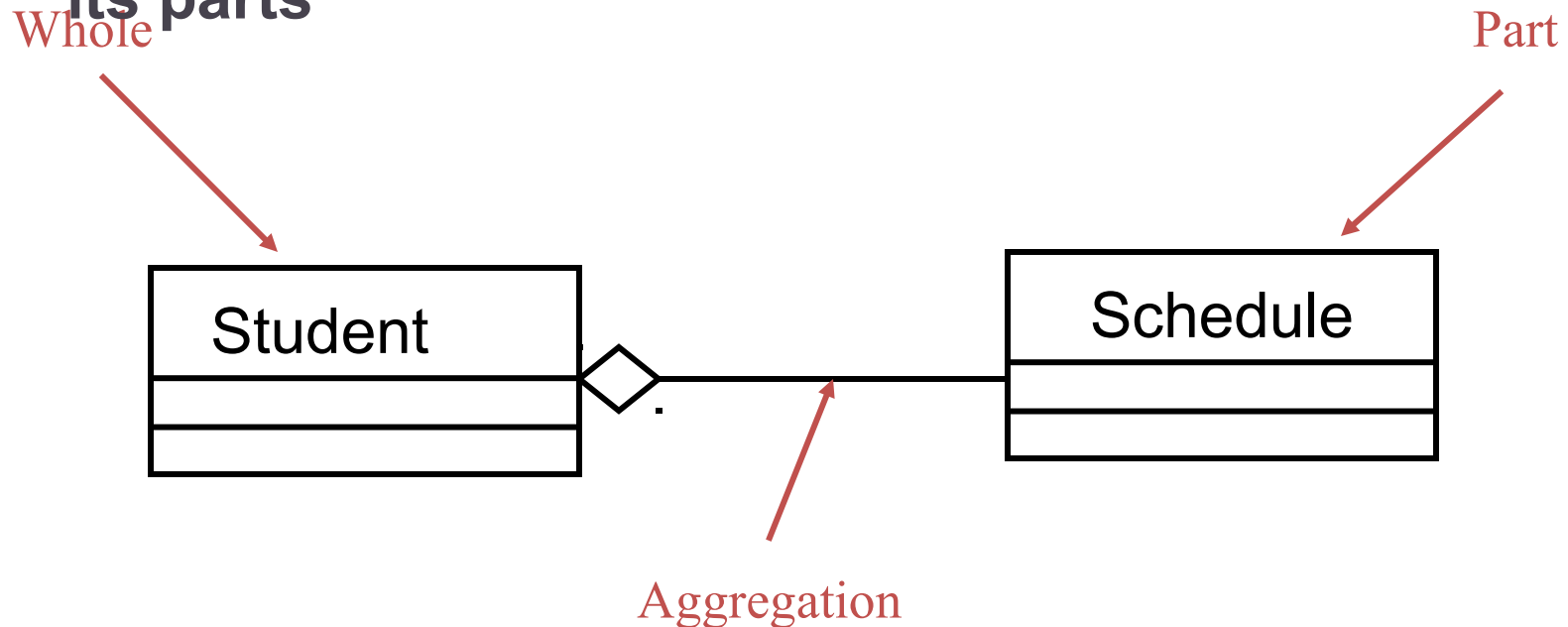
Models a semantic connection among classes





# Relationships: Aggregation

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

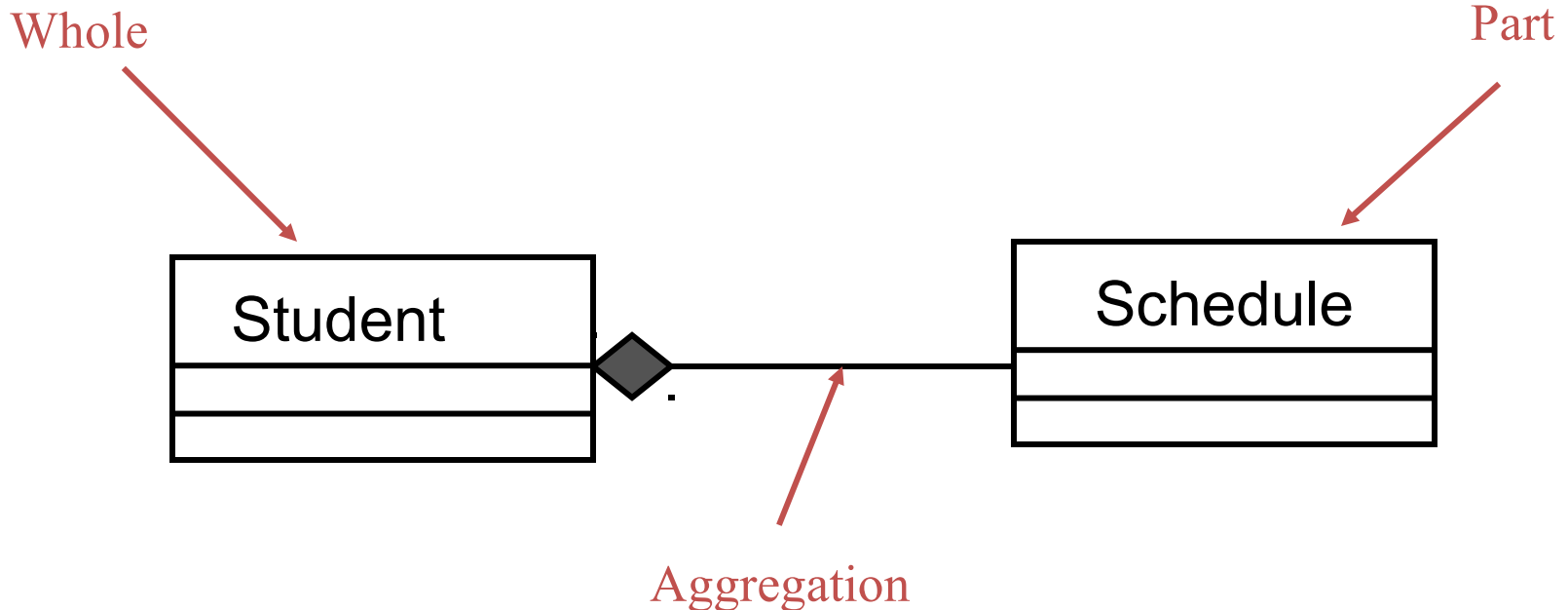


This is sometimes called a 'has\_a' relationship

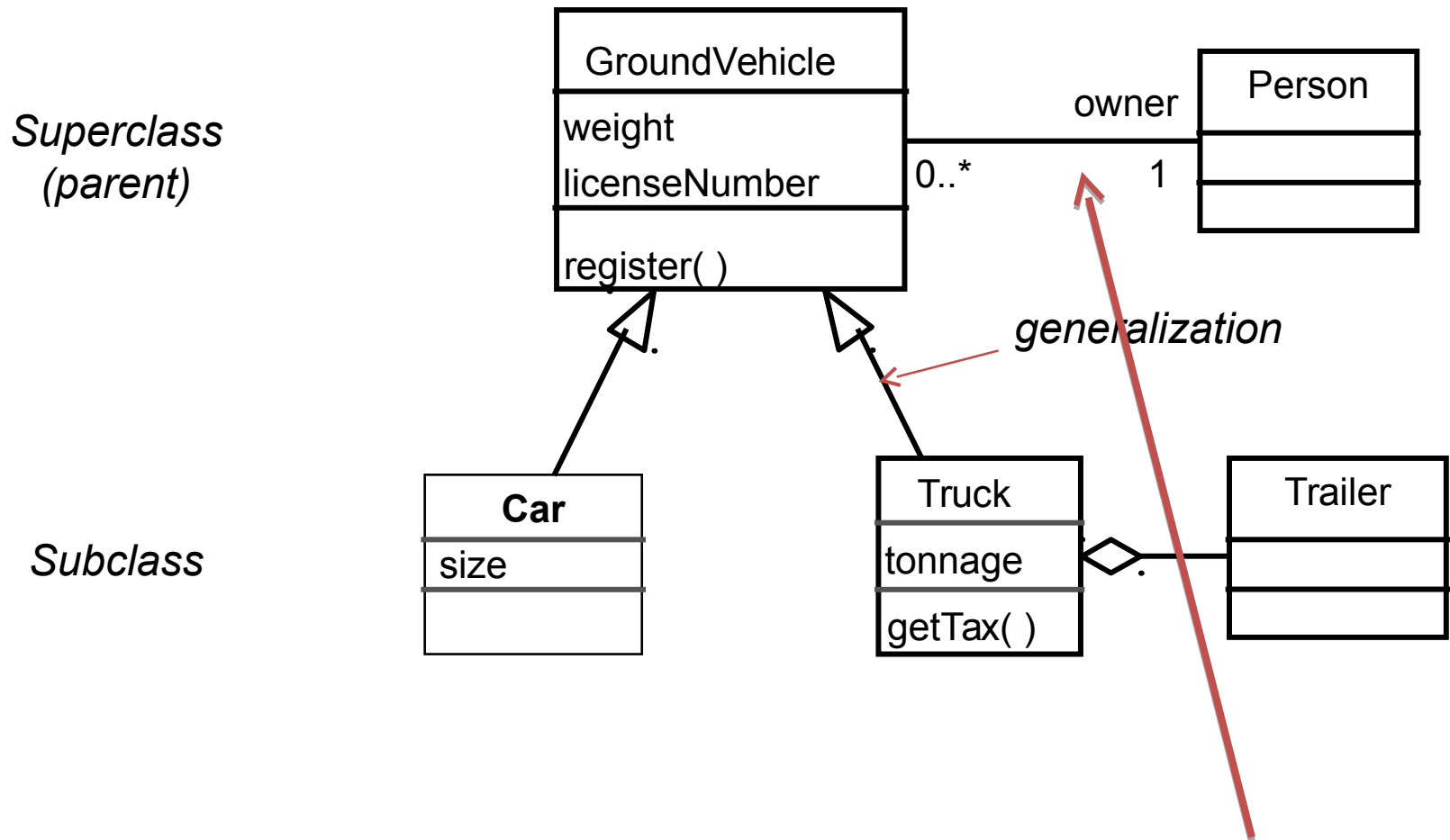
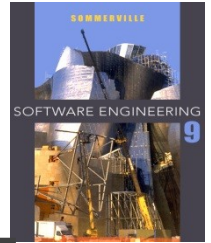
# Relationships: Composition

A form of aggregation with strong ownership and coincident lifetimes

The parts cannot survive the whole/aggregate

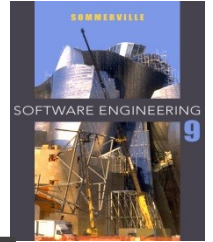


# Example: What Gets Inherited



# Key points

---



- ✂ 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.

## 4. 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:
  - ✂ **A. Data** Some data arrives that has to be processed by the system.
  - ✂ **B. Events** Some event happens that triggers system processing. Events may have associated data, although this is not always the case.

## A. Data-driven Modeling

---

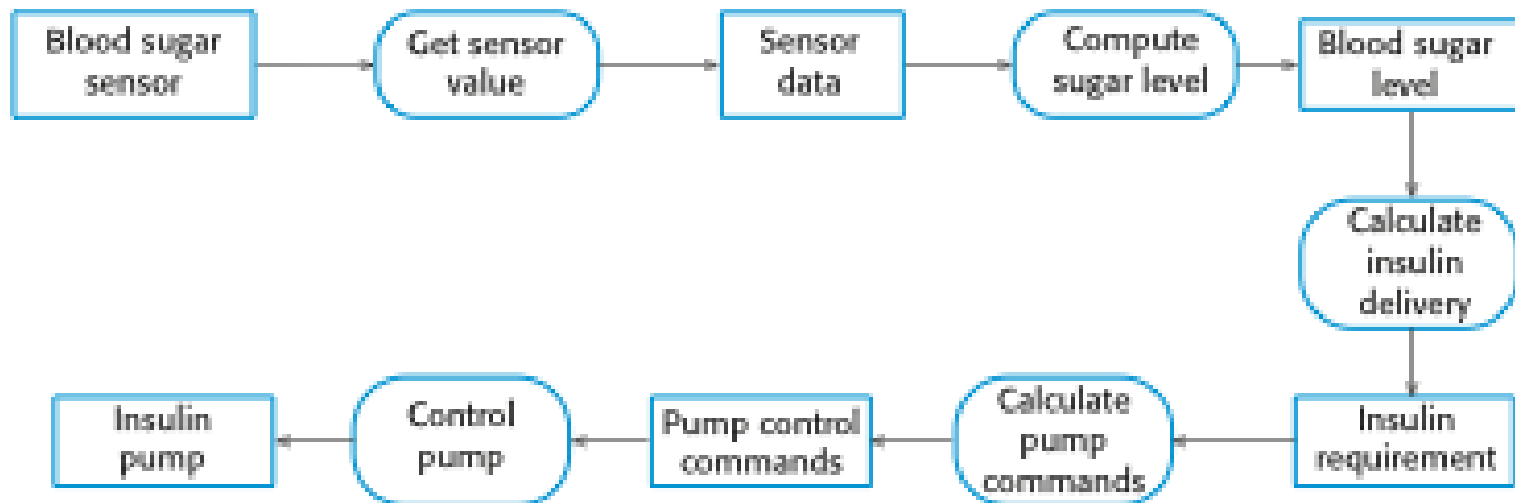
- ✂ Many business systems are data-processing systems that are primarily driven by data.
  - ✂ They are controlled by the data input to the system, with relatively little external event processing.
- ✂ Data-driven models show the sequence of actions involved in processing input data and generating an 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

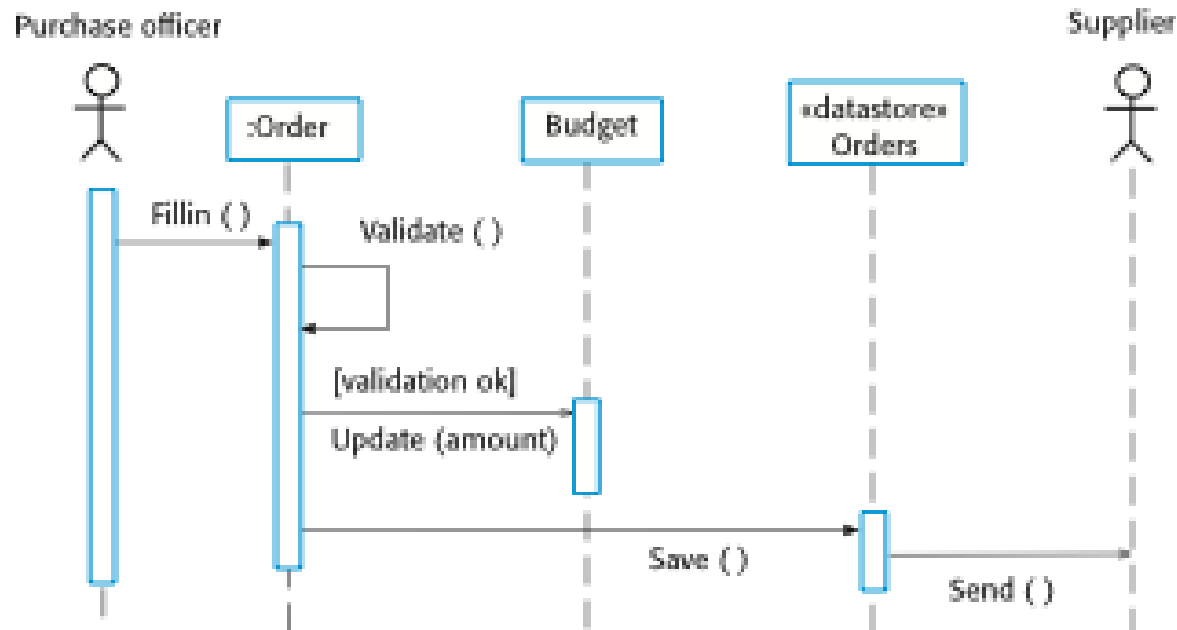
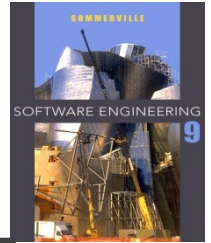
These are used a lot!

They are not flowcharts or flowgraphs; merely 'activities' that need to or must take place.

(This is an example (later) of a Pipe-Filter Architectural Model)



# Order Processing – Sequence Diagram - behavioral



Shows the Sequence of events over time as messages are issued from one object to another.

Terms: life of the object; lifelines, actors, unnamed objects, recursion, and more.

Very Important to show the scenarios in motion. Dynamic!



## B. 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.

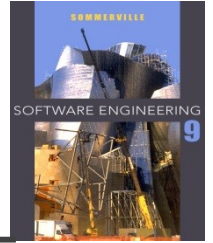
# State Machine Models

---

- ✂ 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 modeling 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.

# Key points

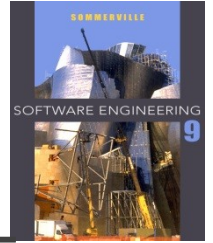
---



- ✂ 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.

# 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.

---

# User Interface Design

# User interface design

---

- Designing effective interfaces for software systems

# Graphical user interfaces

---

- Most users of business systems interact with these systems through graphical interfaces although, in some cases, legacy text-based interfaces are still used

# GUI characteristics

---

Characteristic	Description
Windows	Multiple windows allow different information to be displayed simultaneously on the user's screen.
Icons	Icons different types of information. On some systems, icons represent files; on others, icons represent processes.
Menus	Commands are selected from a menu rather than typed in a command language.
Pointing	A pointing device such as a mouse is used for selecting choices from a menu or indicating items of interest in a window.
Graphics	Graphical elements can be mixed with text on the same display.



# GUI advantages

---

- They are easy to learn and use.
  - Users without experience can learn to use the system quickly.
- The user may switch quickly from one task to another and can interact with several different applications.
  - Information remains visible in its own window when attention is switched.
- Fast, full-screen interaction is possible with immediate access to anywhere on the screen

# Design principles

---

- **User familiarity**
  - The interface should be based on user-oriented terms and concepts rather than computer concepts. For example, an office system should use concepts such as letters, documents, folders etc. rather than directories, file identifiers, etc.
- **Consistency**
  - The system should display an appropriate level of consistency. Commands and menus should have the same format, command punctuation should be similar, etc.
- **Minimal surprise**
  - If a command operates in a known way, the user should be able to predict the operation of comparable commands

# Design principles

---

- Recoverability
  - The system should provide some resilience to user errors and allow the user to recover from errors. This might include an undo facility, confirmation of destructive actions, 'soft' deletes, etc.
- User guidance
  - Some user guidance such as help systems, on-line manuals, etc. should be supplied
- User diversity
  - Interaction facilities for different types of user should be supported. For example, some users have seeing difficulties and so larger text should be available

# User-system interaction

---

- Two problems must be addressed in interactive systems design
  - How should information from the user be provided to the computer system?
  - How should information from the computer system be presented to the user?
- User interaction and information presentation may be integrated through a coherent framework such as a user interface metaphor

# Interaction styles

---

- Direct manipulation
- Menu selection
- Form fill-in
- Command language
- Natural language

# Control panel interface

---

Title	JSD. example	<input type="checkbox"/>	Grid	Busy
Method	JSD			
Type	Network	Units	cm ▶	QUIT
Selection	Process	Reduce	Full ▶	PRINT
NODE LINKS FONT LABEL EDIT				

# Menu systems

---

- Users make a selection from a list of possibilities presented to them by the system
- The selection may be made by pointing and clicking with a mouse, using cursor keys or by typing the name of the selection
- May make use of simple-to-use terminals such as touchscreens

# Advantages of menu systems

---

- Users need not remember command names as they are always presented with a list of valid commands
- Typing effort is minimal
- User errors are trapped by the interface
- Context-dependent help can be provided. The user's context is indicated by the current menu selection



# Form-based interface

---

## NEW BOOK

Title

ISBN

Author

Price

Publisher

Publication  
date

Edition

Number of  
copies

Classification

Loan  
status

Date of  
purchase

Order  
status

# Command interfaces

---

- User types commands to give instructions to the system e.g. UNIX
- May be implemented using cheap terminals.
- Easy to process using compiler techniques
- Commands of arbitrary complexity can be created by command combination
- Concise interfaces requiring minimal typing can be created

# Problems with command interfaces

---

- Users have to learn and remember a command language. Command interfaces are therefore unsuitable for occasional users
- Users make errors in command. An error detection and recovery system is required
- System interaction is through a keyboard so typing ability is required

# Command languages

---

- Often preferred by experienced users because they allow for faster interaction with the system
- Not suitable for casual or inexperienced users
- May be provided as an alternative to menu commands (keyboard shortcuts). In some cases, a command language interface and a menu-based interface are supported at the same time

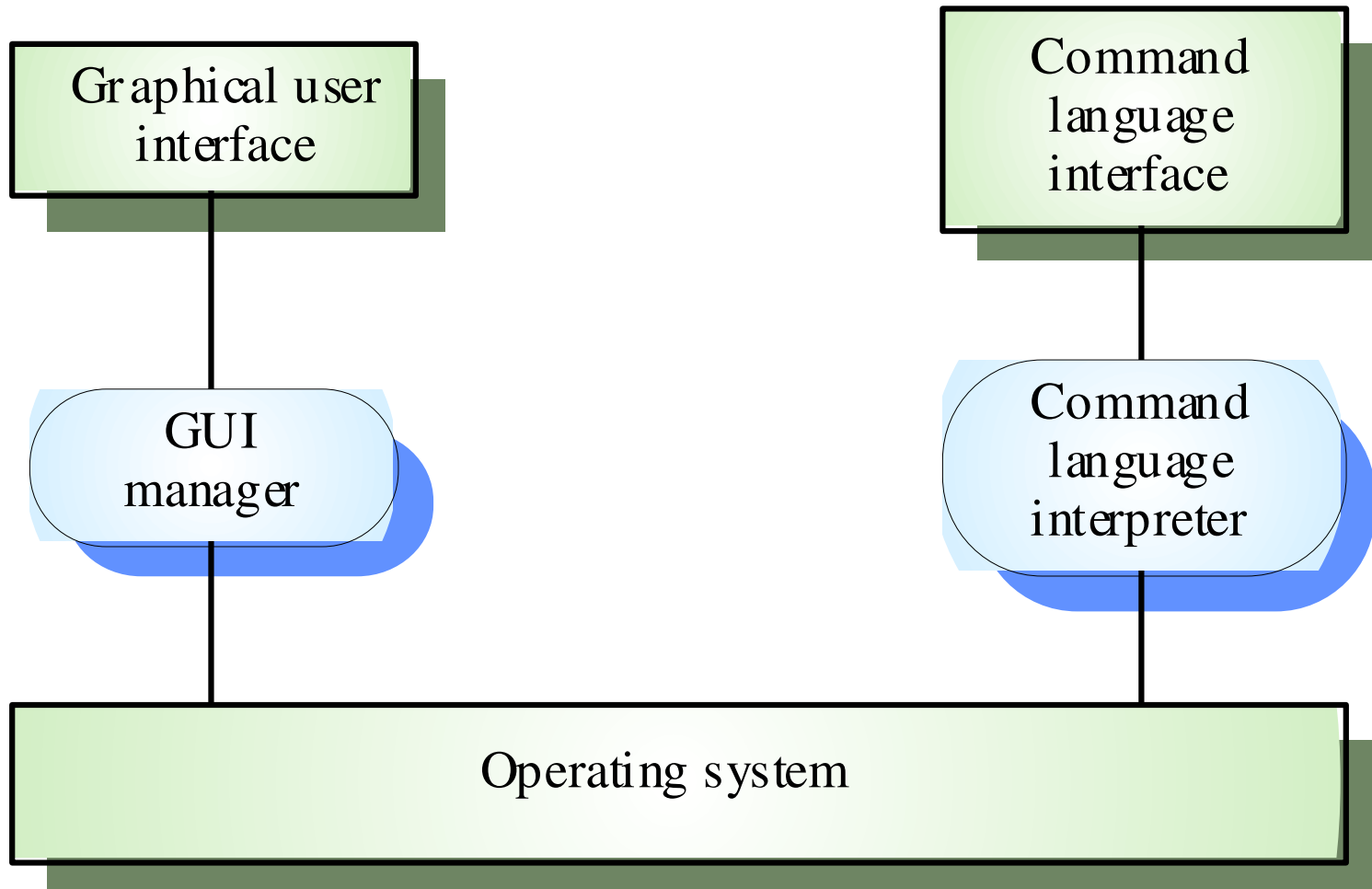
# Natural language interfaces

---

- The user types a command in a natural language. Generally, the vocabulary is limited and these systems are confined to specific application domains (e.g. timetable enquiries)
- NL processing technology is now good enough to make these interfaces effective for casual users but experienced users find that they require too much typing

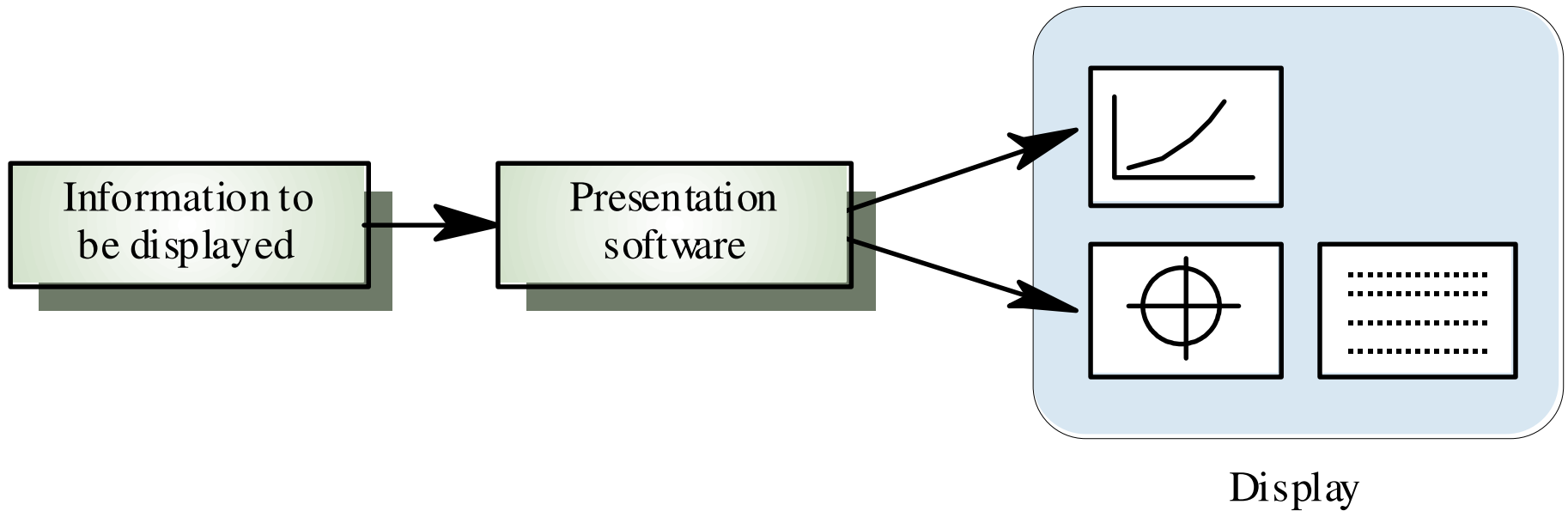
# Multiple user interfaces

---



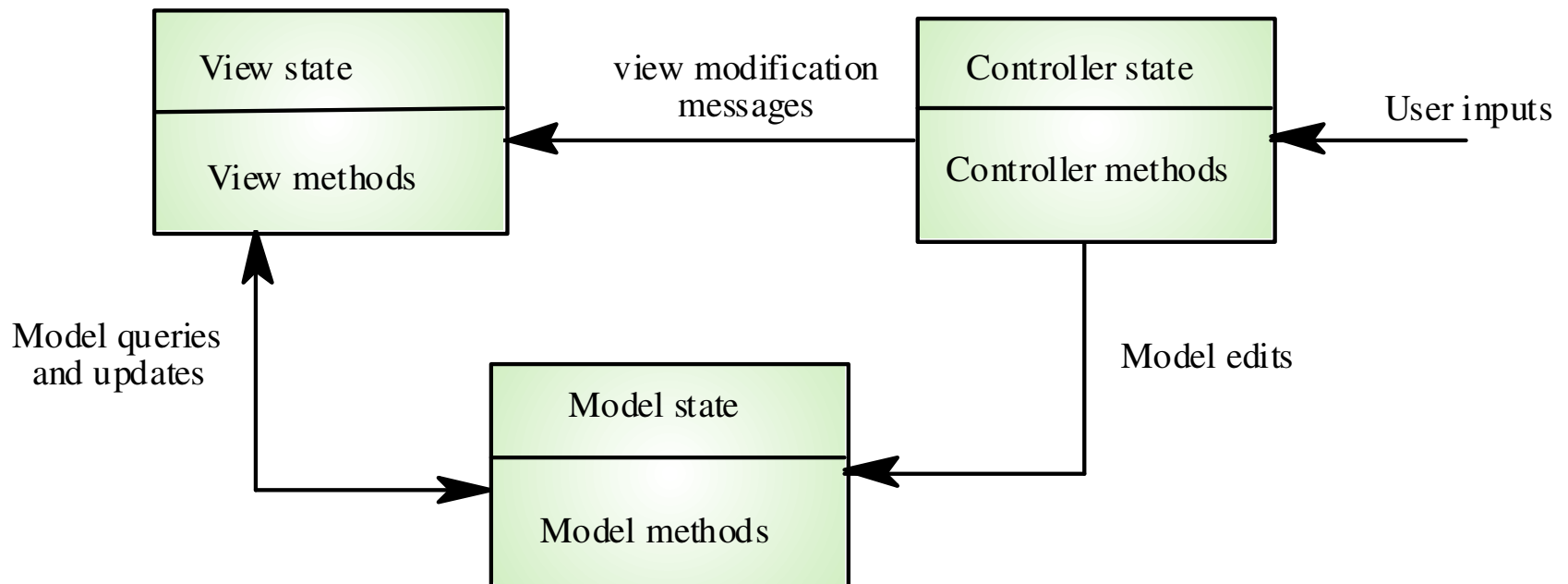
# Information presentation

---



# Model-view-controller

---





# Information presentation

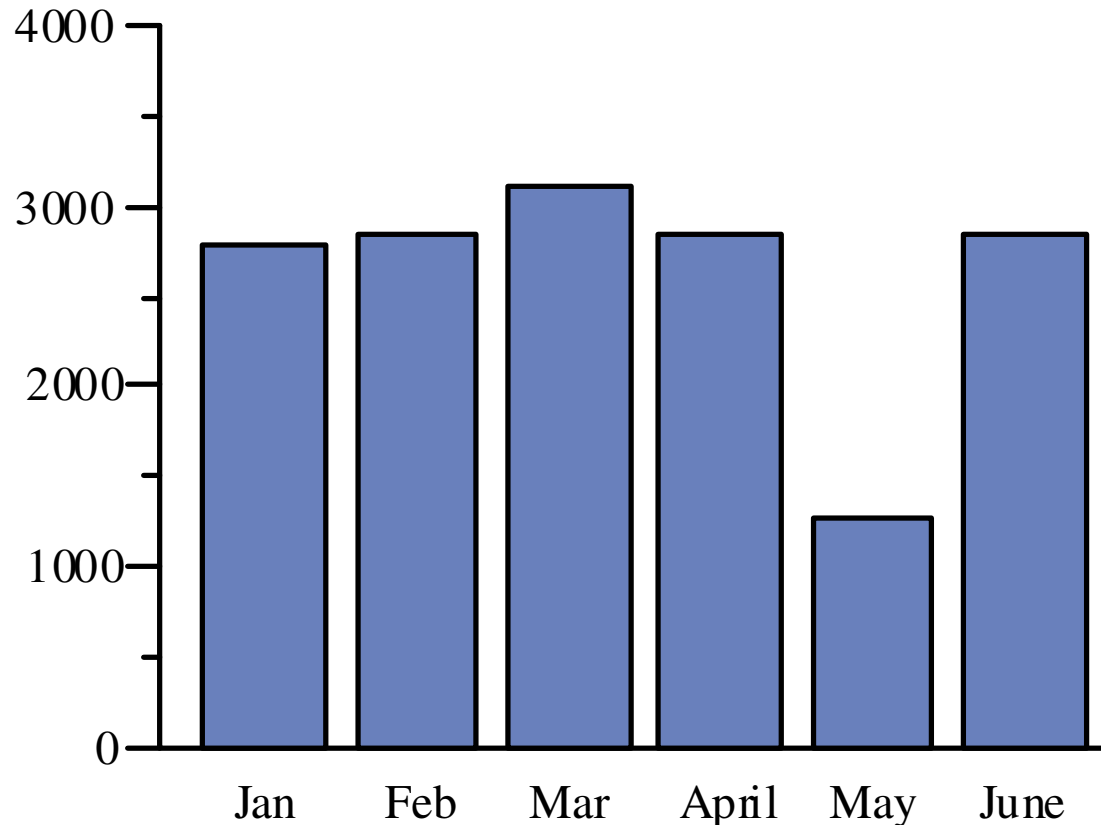
---

- **Static information**
  - Initialised at the beginning of a session. It does not change during the session
  - May be either numeric or textual
- **Dynamic information**
  - Changes during a session and the changes must be communicated to the system user
  - May be either numeric or textual

# Alternative information presentations

---

Jan	Feb	Mar	April	May	June
2842	2851	3164	2789	1273	2835



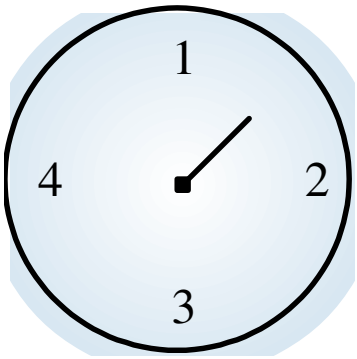
# Analogue vs. digital presentation

---

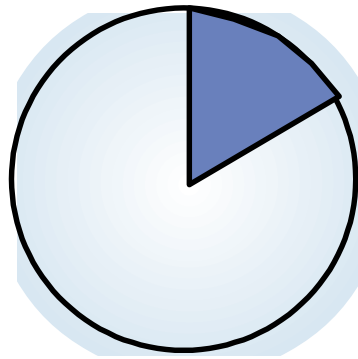
- Digital presentation
  - Compact - takes up little screen space
  - Precise values can be communicated
- Analogue presentation
  - Easier to get an 'at a glance' impression of a value
  - Possible to show relative values
  - Easier to see exceptional data values

# Dynamic information display

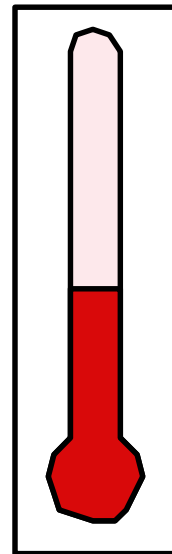
---



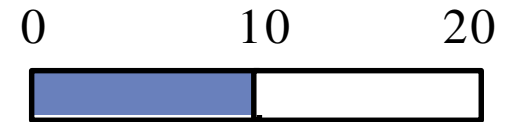
Dial with needle



Pie chart



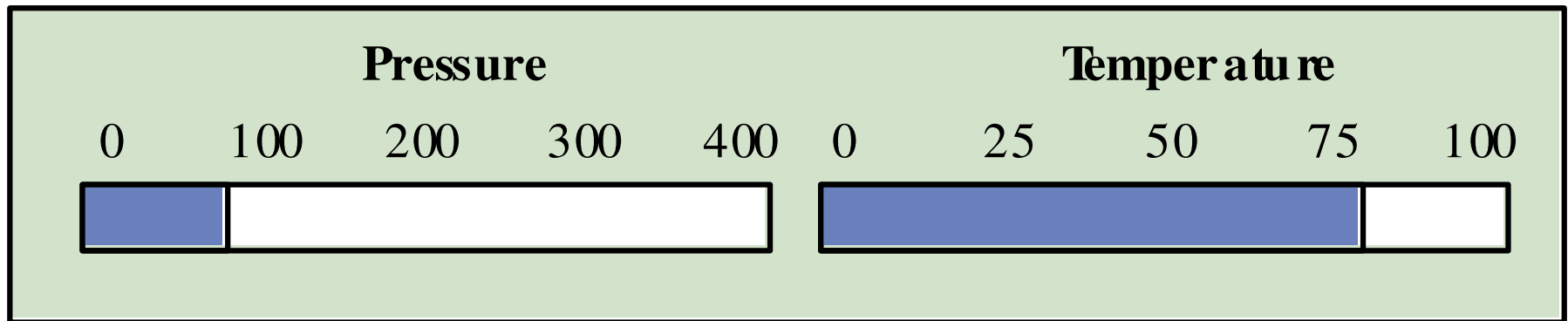
Thermometer



Horizontal bar

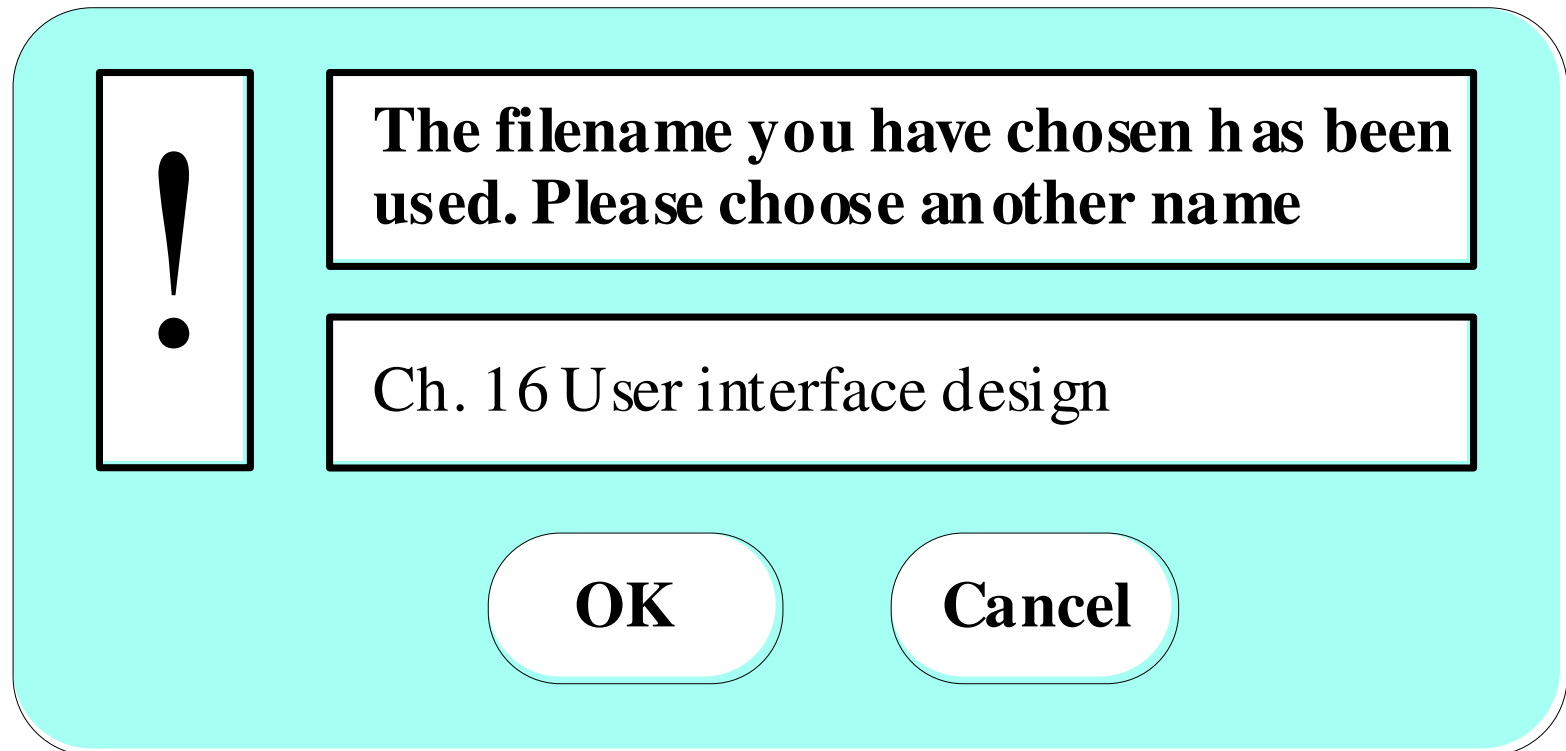
# Displaying relative values

---



# Textual highlighting

---



# Colour use guidelines

---

- Don't use too many colours
- Use colour coding to support use tasks
- Allow users to control colour coding
- Design for monochrome then add colour
- Use colour coding consistently
- Avoid colour pairings which clash
- Use colour change to show status change
- Be aware that colour displays are usually lower resolution

# **Nurse input of a patient's name**

---

Please type the patient name in the box then click ok

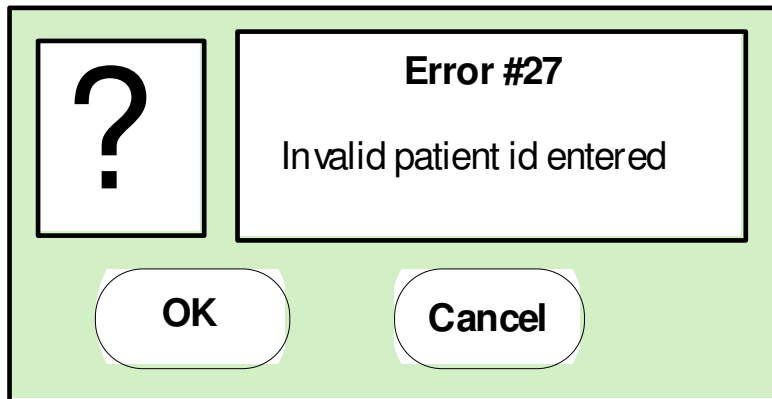
Bates J.

OK Cancel

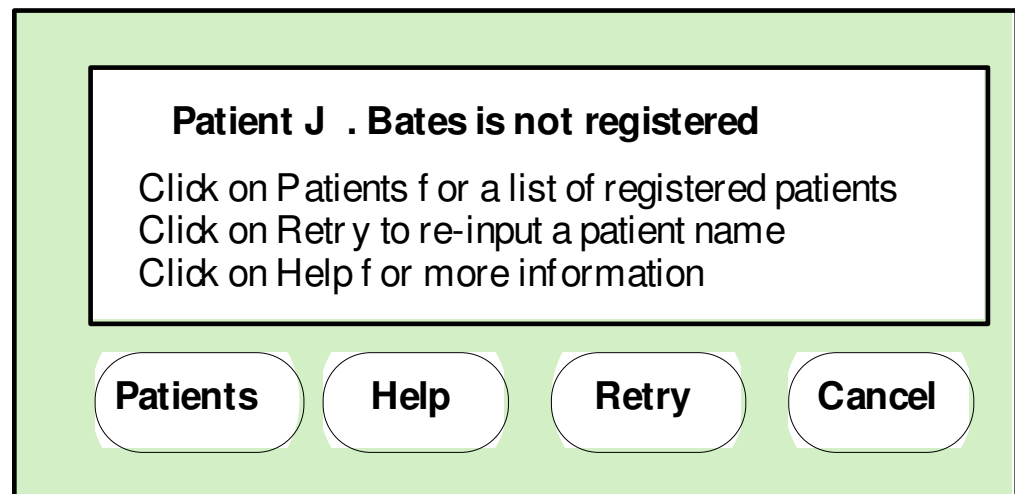


# System and user-oriented error messages

System-oriented error message



User-oriented error message



# Help system design

---

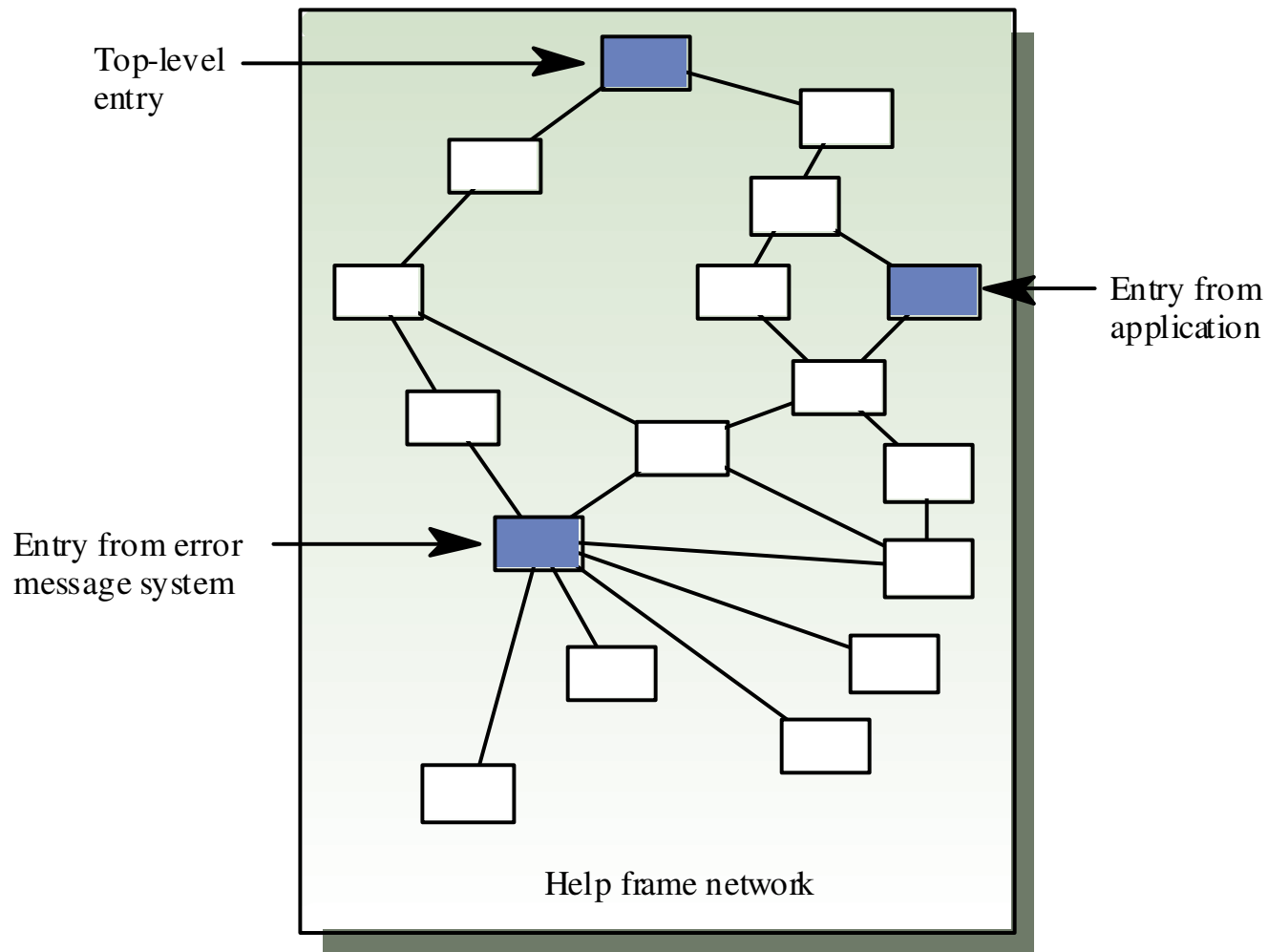
- *Help?* means ‘help I want information’
- *Help!* means “HELP. I'm in trouble”
- Both of these requirements have to be taken into account in help system design
- Different facilities in the help system may be required

# Help system use

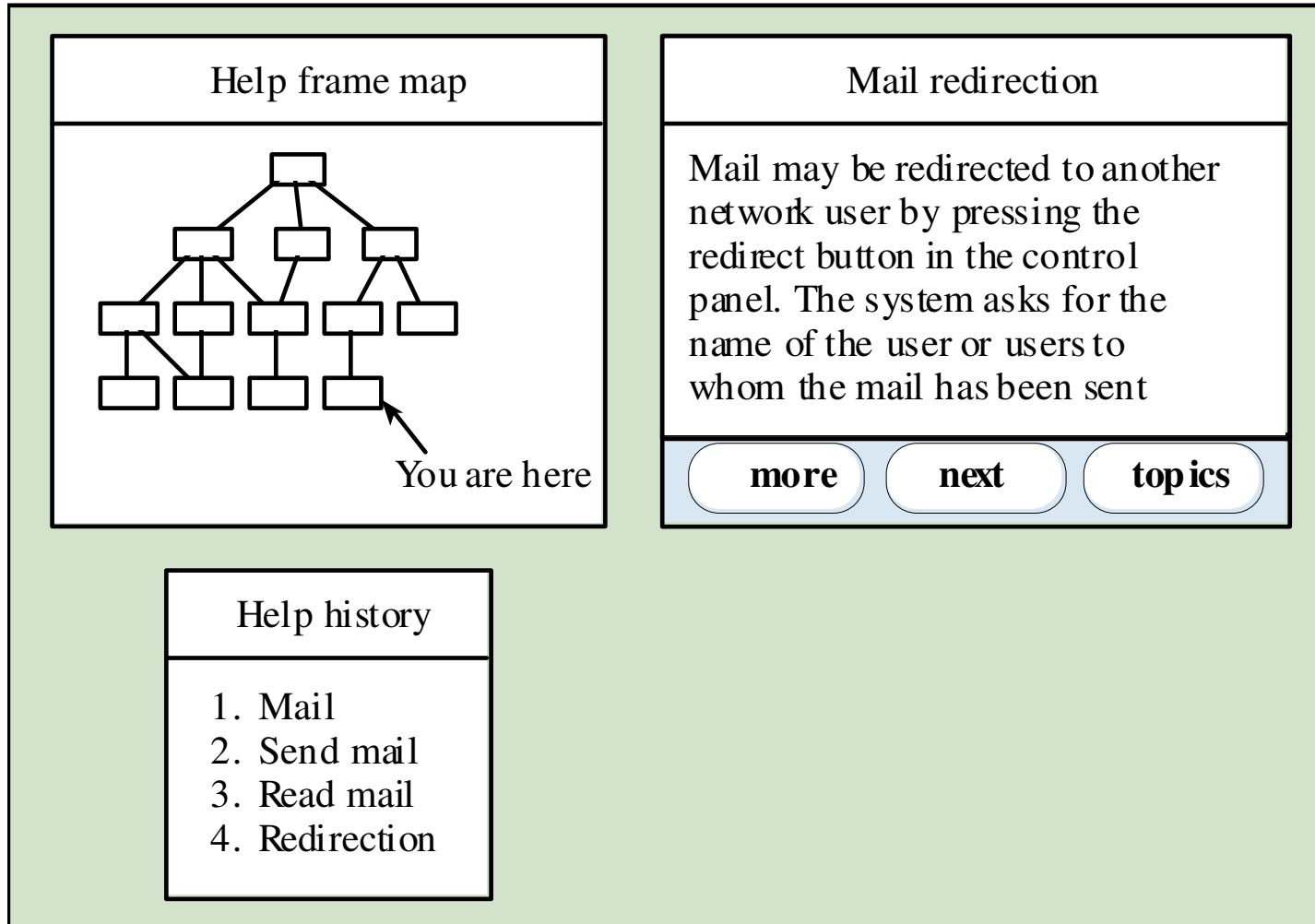
---

- Multiple entry points should be provided so that the user can get into the help system from different places.
- Some indication of where the user is positioned in the help system is valuable.
- Facilities should be provided to allow the user to navigate and traverse the help system.

# Entry points to a help system



# Help system windows



# Usability attributes

---

Attribute	Description
Learnability	How long does it take a new user to become productive with the system?
Speed of operation	How well does the system response match the user's work practice?
Robustness	How tolerant is the system of user error?
Recoverability	How good is the system at recovering from user errors?
Adaptability	How closely is the system tied to a single model of work?

# Key points

---

- Interface design should be user-centred. An interface should be logical and consistent and help users recover from errors
- Interaction styles include direct manipulation, menu systems form fill-in, command languages and natural language
- Graphical displays should be used to present trends and approximate values. Digital displays when precision is required
- Colour should be used sparingly and consistently

# Key points

---

- Systems should provide on-line help. This should include “help, I’m in trouble” and “help, I want information”
- Error messages should be positive rather than negative.
- A range of different types of user documents should be provided
- Ideally, a user interface should be evaluated against a usability specification