Module 1

Topics

- Introduction to Software Engineering
- Reasons for software project failure
- Similarities and differences between software and other engineering products.
- Software Development Life Cycle (SDLC)
  - Overview of Phases
- Detailed Study of Requirements Phase.

What is a Software

- Computer programs and associated documentation.
- Software products may be developed for a particular customer or may be developed for a general market.

The Nature of Software...

- Software is intangible
  - Hard to understand development effort
- Software is easy to reproduce
  - Cost is in its *development*
    - in other engineering products, manufacturing is the costly stage.
- The industry is labor-intensive
  - Hard to automate

The Nature of Software ...

- Untrained people can hack something together
  - Quality problems are hard to notice
- Software is easy to modify
  - People make changes without fully understanding it
- Software does not ‘wear out’
  - It *deteriorates* by having its design changed:
    - erroneously, or
    - in ways that were not anticipated, thus making it complex

The Nature of Software

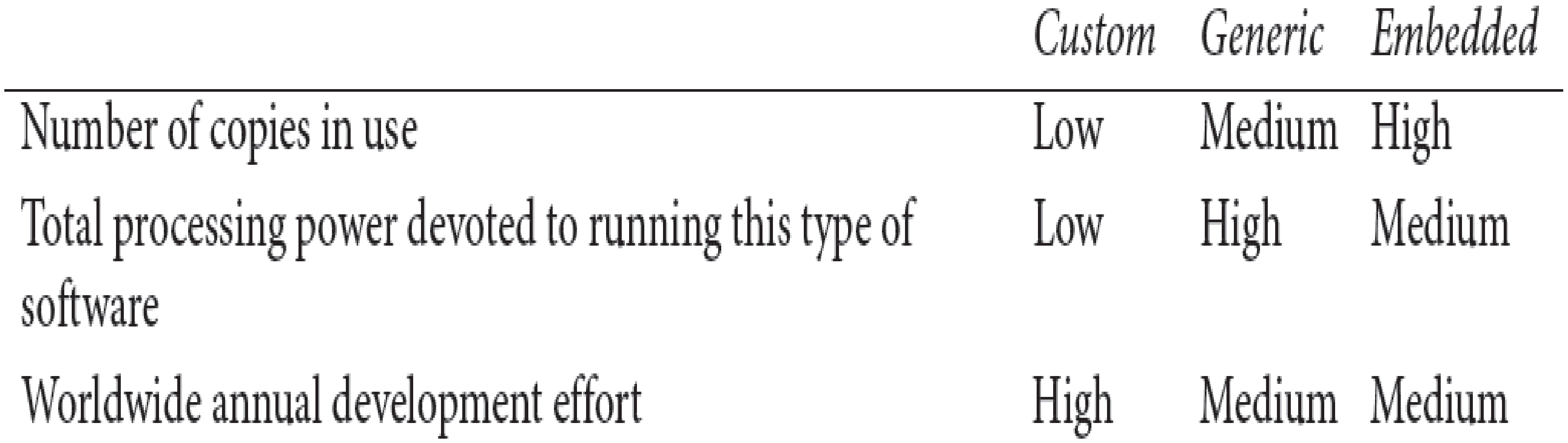
- Conclusions
  - Much software has poor design and is getting worse
  - Demand for software is high and rising
  - We are in a perpetual ‘software crisis’
  - We have to learn to **‘engineer’** software

Types of Software...

- Custom
  - For a specific customer
  - e.g. web sites, air-traffic control systems and software for managing the specialized finances of large organizations.
- Generic
  - Sold on open market
  - Often called
    - COTS (Commercial Off The Shelf)
    - Shrink-wrapped
  - E.g. word processors, spreadsheets, compilers, web browsers, operating systems, computer games
- Embedded
  - Built into hardware
  - Hard to change

Types of Software

Differences among custom, generic and embedded software



	Custom	*Generic*	*Embedded*
Number of copies in use	Low	Medium	High
Total processing power devoted to running this type of software	Low	High	Medium
Worldwide annual development effort	High	Medium	Medium

Types of Software

- Real time software
  - E.g. control and monitoring systems
  - Must react immediately
  - Safety often a concern
- Data processing software
  - Used to run businesses
  - performs functions such as recording sales, managing accounts, printing bills etc
  - Accuracy and security of data are key

- *Some software has both aspects*

What is Engineering

- Engineering is …
  - The application of scientific principles and methods to the construction of useful structures
- Examples
  - Mechanical Engineering
  - Computer Engineering
  - Civil Engineering
  - Electrical Engineering
  - Nuclear Engineering

Engineering…

- Requires well defined approach: repeatable ,predictable
- Large projects requires managing the projects itself
  - Manage people, money(cost),equipment, schedule
  - Scale makes big differences: compare building a hut, storeyed house,50storeyed apartment building
- Quality extremely important: relates to failures, efficiency, usability…
  - People willing to pay for quality

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

- Other definitions:
  - IEEE: (1) the application of a systematic, disciplined, quantifiable approach to the development, operation, maintenance of software; that is, the application of engineering to software. (2) The study of approaches as in (1).
  - The Canadian Standards Association: The systematic activities involved in the design, implementation and testing of software to optimize its production and support.

What is Software Engineering?…

- Solving customers’ problems
  - This is the *goal* of software engineering
  - Sometimes the solution is to *buy, not build*
  - Adding unnecessary features does not help solve the problem
  - Software engineers must *communicate effectively* to identify and understand the problem

What is Software Engineering?…

- **Systematic development and evolution**
  - An engineering process involves applying *well understood techniques* in a organized and *disciplined* way
  - Many well-accepted practices have been formally standardized
    - e.g. by the IEEE or ISO
  - Most development work is *evolution*

What is Software Engineering?…

- **Large, high quality software systems**
  - Software engineering techniques are needed because large systems *cannot be completely understood* by one person
  - Teamwork and co-ordination are required
  - Key challenge: Dividing up the work and ensuring that the parts of the system work properly together
  - The end-product must be of sufficient quality

What is Software Engineering?

- **Cost, time and other constraints**
  - Finite resources
  - The benefit must outweigh the cost
  - Others are competing to do the job cheaper and faster
  - Inaccurate estimates of cost and time have caused many project failures

Difficulties and Risks in Software Engineering

- Complexity and large numbers of details
- Uncertainty about technology
- Uncertainty about requirements
- Uncertainty about software engineering skills
- Constant change
- Deterioration of software design
- Political risks

Software Engineering and the Engineering Profession

- The term Software Engineering was coined in 1968
  - People began to realize that the principles of engineering should be applied to software development
- Engineering is a licensed profession
  - In order to protect the public
  - Engineers design artifacts following well accepted practices which involve the application of science, mathematics and economics
  - Ethical practice is also a key tenet of the profession
- In many countries, much software engineering does not require an engineering license, but is still engineering

Software Engineering and the Engineering Profession

- Ethics in Software Engineering:
- Software engineers shall
  - Act consistently with public interest
  - Act in the best interests of their clients
  - Develop and maintain with the highest standards possible
  - Maintain integrity and independence
  - Promote an ethical approach in management
  - Advance the integrity and reputation of the profession
  - Be fair and supportive to colleagues
  - Participate in lifelong learning

Software Engineering and the Engineering Profession

- Ethics in Software Engineering:

- Software engineers shall
  - Act consistently with public interest
  - Act in the best interests of their clients
  - Develop and maintain with the highest standards possible
  - Maintain integrity and independence
  - Promote an ethical approach in management
  - Advance the integrity and reputation of the profession
  - Be fair and supportive to colleagues
  - Participate in lifelong learning

Stakeholders in Software Engineering

1. **Users**
   – Those who use the software
2. **Customers(Clients)**
   – Those who pay for the software
   – goal is either to increase profits or simply to run their business more effectively
3. **Software developers**
   – develop and maintain the software
4. **Development Managers**

   - people who run the organization that is developing the software

- All four roles can be fulfilled by the same person

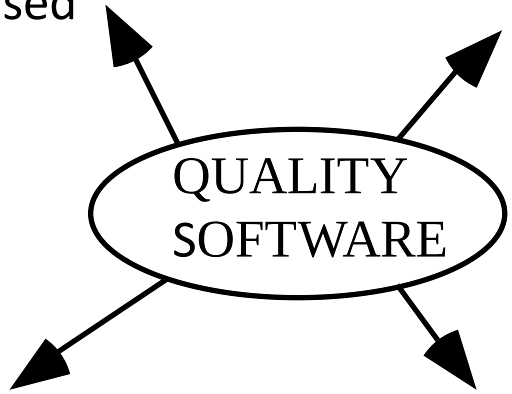
Software Quality...

- **Usability**
  - Users can learn it and fast and get their job done easily
- **Efficiency**
  - It doesn’t waste resources such as CPU time and memory
- **Reliability**
  - It does what it is required to do without failing
- **Maintainability**
  - It can be easily changed
- **Reusability**
  - Its parts can be used in other projects, so reprogramming is not needed

Software Quality and the Stakeholders

Customer:
solves problems at
an acceptable cost in
terms of money paid and
resources used

User:
easy to learn;
efficient to use;
helps get work done



Developer:
easy to design;
easy to maintain;
easy to reuse its parts

Development manager:
sells more and
pleases customers
while costing less
to develop and maintain

Software Quality: Conflicts and Objectives

- The different qualities can conflict
  - Increasing efficiency can reduce maintainability or reusability
  - Increasing usability can reduce efficiency
- Setting objectives for quality is a key engineering activity
  - You then design to meet the objectives
  - Avoids ‘over-engineering’ which wastes money
- Optimizing is also sometimes necessary
  - E.g. obtain the highest possible reliability using a fixed budget

Internal Quality Criteria

- These:
  - Characterize *aspects of the design* of the software
  - Have an effect on the external quality attributes
  - E.g.
    - The amount of commenting of the code
    - The complexity of the code

Short Term Vs. Long Term Quality

- Short term:
  - Does the software *meet the customer's immediate needs*?
  - Is it sufficiently efficient for the volume of data we have *today*?
- Long term:
  - Maintainability
  - Customer's future needs
  - Scalability: Can the software handle larger volumes of data?

Software Engineering Projects

- Three major categories:
  - modifying an existing system;
  - starting to develop system from scratch
  - building most of a new system from existing components, while developing new software only for missing details.

Software Engineering Projects

- Most projects are *evolutionary or maintenance* projects, involving work on *legacy* systems
  - Corrective projects: fixing defects
  - Adaptive projects: changing the system in response to changes in
    - Operating system
    - Database
    - Rules and regulations
  - Enhancement projects: adding new features for users
  - Reengineering or perfective projects: changing the system internally so it is more maintainable

Software Engineering Projects

- ‘Green field’ projects
  - New development
  - The minority of projects
  - a wider freedom to be creative about the design.
  - not constrained by the design decisions and errors made by predecessors

Software Engineering Projects

- Projects that involve building on a *framework* or a set of existing components.
  - A framework is an application that is missing some important details.
    - E.g. Specific rules of this organization.
  - Such projects:
    - Involve plugging together *components* that are:
      - Already developed.
      - Provide significant functionality.
    - Benefit from reusing reliable software.
    - Provide much of the same freedom to innovate found in green field development.

Activities Common to Software Projects...

- Requirements and specification
  - Includes
    - Domain analysis
    - Defining the problem
    - Requirements gathering
      - Obtaining input from as many sources as possible
    - Requirements analysis
      - Organizing the information
    - Requirements specification
      - Writing detailed instructions about how the software should behave

Activities Common to Software Projects...

- Design
  - Deciding how the requirements should be implemented, using the available technology
  - Includes:
    - *Systems engineering*: Deciding what should be in hardware and what in software
    - *Software architecture*: Dividing the system into subsystems and deciding how the subsystems will interact
    - *Detailed design* of the internals of a subsystem
    - *User interface design*
    - *Design of databases*

Activities Common to Software Projects

- Modeling
  - Creating representations of the domain or the software
    - Use case modeling
    - Structural modeling
    - Dynamic and behavioral modeling
- Programming
  - translation of higher-level designs into particular programming languages
- Quality assurance
  - Reviews and inspections
  - Testing
- Deployment
  - distributing and installing the software and any other components of the system such as databases, special hardware etc
- Managing the process
  - Estimating the cost of the system
  - Planning

Reasons for software project failure

Introduction

- Software development projects have not always been successful.
- There are still many reports of software projects going wrong and ‘**software failures**’.
- Software engineering is criticized as inadequate for modern software development

Successful Software Systems…

- When do we consider a software application successful?
  - Development completed
  - It is useful
  - Usable
  - it is used

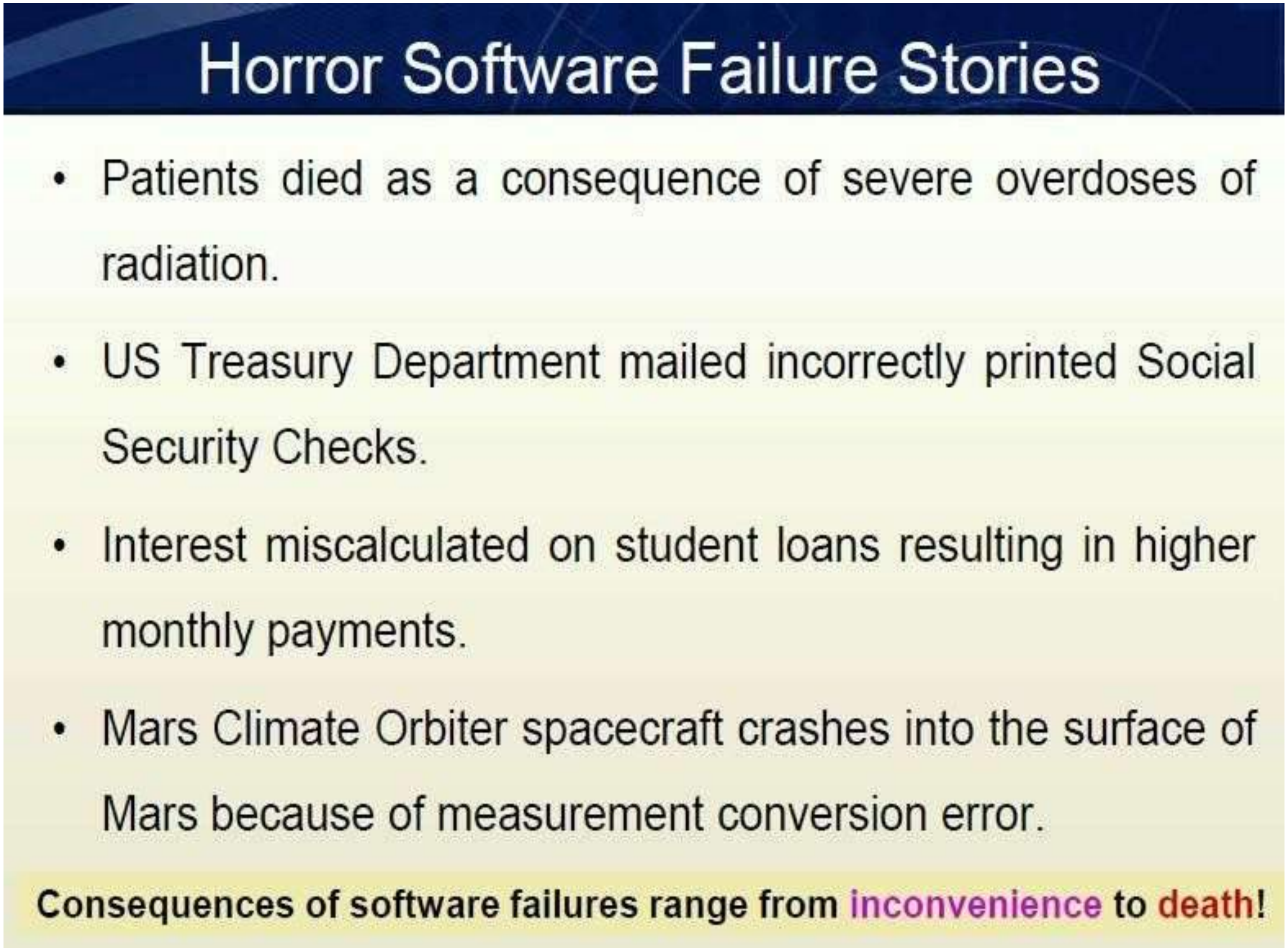
- Cost effectiveness , maintainability implied

Reasons for failures

- Schedule slippage
- Cost over-runs
- Does not solve user’s problem
- Poor quality of software
- Poor maintainability

Reasons for failures…

- Ad hoc software development results in such problems
  - No planning of development work(e.g. no milestones defined)
  - Deliverables to user not identified
  - Poor understanding of user requirements
  - No control or review
  - Technical incompetence of developers
  - Poor understanding of cost and effort by both developer and user



Horror Software Failure Stories

- Patients died as a consequence of severe overdoses of radiation.
- US Treasury Department mailed incorrectly printed Social Security Checks.
- Interest miscalculated on student loans resulting in higher monthly payments.
- Mars Climate Orbiter spacecraft crashes into the surface of Mars because of measurement conversion error.

Consequences of software failures range from inconvenience to death!

Some Software failures

Ariane 5

It took the European Space Agency **10 years and $7 billion** to produce Ariane 5, a giant rocket capable of hurling a pair of three-ton satellites into orbit with each launch and intended to give Europe overwhelming supremacy in the commercial space business.

The rocket was destroyed after 39 seconds of its launch, at an altitude of two and a half miles along with its payload of four expensive and uninsured scientific satellites.

Some Software failures

The Patriot Missile

- First time used in Gulf war
- Used as a defense from Iraqi Scud missiles
- Failed several times including one that killed 28 US soldiers in Dhahran, Saudi Arabia

Reasons:

A small timing error in the system's clock accumulated to the point that after 14 hours, the tracking system was no longer accurate. In the Dhahran attack, the system had been operating for more than 100 hours.

CS3004D Software Engineering

SDLC – Software Development Life Cycle

Introduction

- Software applications can be simple or complex
- Our aim is to build successful software products
- Large software projects are challenging
- Ad hoc software development can result in failures
- Engineering approach is essential

Engineering Approach for Software

- Estimate the cost and effort involved
- Should plan and schedule the work.
- Should involve users in defining requirements, what exactly is expected from the software.
- Should identify the stages in the development.
- Should define clear milestones

Software Process

- Process defines a set of steps
- These steps need to be carried out in a particular order
- Different types of processes in a software domain
  - process for software development
  - Process for managing the project
  - Process for change and configuration management
  - Process for managing the above processes

Step in a Software Process

- Well defined objective
- Well defined inputs and outputs
- Entry and Exit criteria

Software Development Process

- As Software Development Life Cycle.
- So SDLC is the process which helps to develop good quality software products
- SDLC is composed of a number of clearly defined and distinct work steps or phases
- A number of SDLC models or process models have been created such as Waterfall, Spiral etc.

Software Development Life Cycle

- Problem Definition
- Feasibility Study
- Requirements Analysis
- Design
- Implementation
- Testing
- Maintenance
- Archival

Problem Definition

“ What is the problem “

- Ensure there exists a problem to be solved
- Define goal
- Usually short and quick
- Categorizing problem
- Avoid misunderstanding
- Identifying cause of the problem
- Checking cost-effectiveness

Problem Definition Document

- Problem statement
- Project objective
- Preliminary Ideas
- Time and Cost for Feasibility Study

Next Phase --- Feasibility Study

- Understanding of problem and reasons

Try to answer:
- Are there feasible solutions?
- Is the problem worth solving?

Look at cost-benefit analysis; efforts
Thorough review on report
Best time to stop the project

Types of Feasibility Study

- Economical
- Technical
- Operational

Cost – Benefit Analysis

- Types of costs
- Types of benefits
- Estimation in early stage; challenging

Feasibility Report

- A brief statement of the problem; System environment
- Important findings and recommendations
- Alternatives
- System description
- Cost-benefit analysis
- Evaluation of technical risk
- Legal consequences

Requirements Analysis

- Knowing user’s requirement in detail
- Objective is to determine what he system must do to solve the problem (without describing how)
- Produces SRS document
- Incorrect, incomplete, inconsistent, ambiguous SRS often results in project failure

Requirement Analysis

- Challenging
  - Users may not know exactly what is needed
  - Users may change their mind over time
  - Users may have conflicting demands
  - Analyst has no or limited domain knowledge
  - Client may be different from the user
  - Users may not be capable to differentiate between what is possible and what is impractical

Thank You

CS3004D Software Engineering

SDLC – Software Development Life Cycle

Summary

- Need an engineering approach
- Software development life cycle
- Problem definition
- Feasibility Study

Software Development Life Cycle

- Problem Definition
- Feasibility Study
- Requirements Analysis
- Design
- Implementation
- Testing
- Maintenance
- Retirement

Requirements Analysis

- Knowing user’s requirement in detail
- Objective is to determine what he system must do to solve the problem (without describing how)
- Produces SRS document
- Incorrect, incomplete, inconsistent, ambiguous SRS often results in project failure

Requirement Analysis

- Challenging
  - Users may not know exactly what is needed
  - Users may change their mind over time
  - Users may have conflicting demands
  - Users may not be capable to differentiate between what is possible and what is impractical
  - Analyst has no or limited domain knowledge
  - Client may be different from the user

SRS

- First and most important baseline
- What the system will be able to do
- Basis for validation and final acceptance
- Cost increases rapidly after this step
- Should be reviewed in detail by user and other analyst
- Should be adequately detailed
- It identifies all functional and performance requirements

Requirements Analysis Process

- Interviewing clients
- Studying existing things
- Long process – should be organized systematically
- Identifies users and business entities
- Get functional or domain knowledge
- Often goes outside – in

Organizing Findings

- Massive amount of information through study
- Need to be organized, recorded and classified
- Ensure consistency and completeness
- Prepare SRS
- Get it reviewed

Design

- Deals with “ How “
- Consider several technical alternatives
- Input is the SRS
- Prepare for technical management review
- Finally delivers design document

Design Goals

- Processing component
- Data component
- Different design paradigms
- System structure
  - Decomposes the complex system
  - Defines the subsystems or modules

Implementation

- Coding are done
- Translating design specification into the source code
- Source code along with internal documentation
- To reduce the cost of later phases
- Making the program more readable
- General coding standards

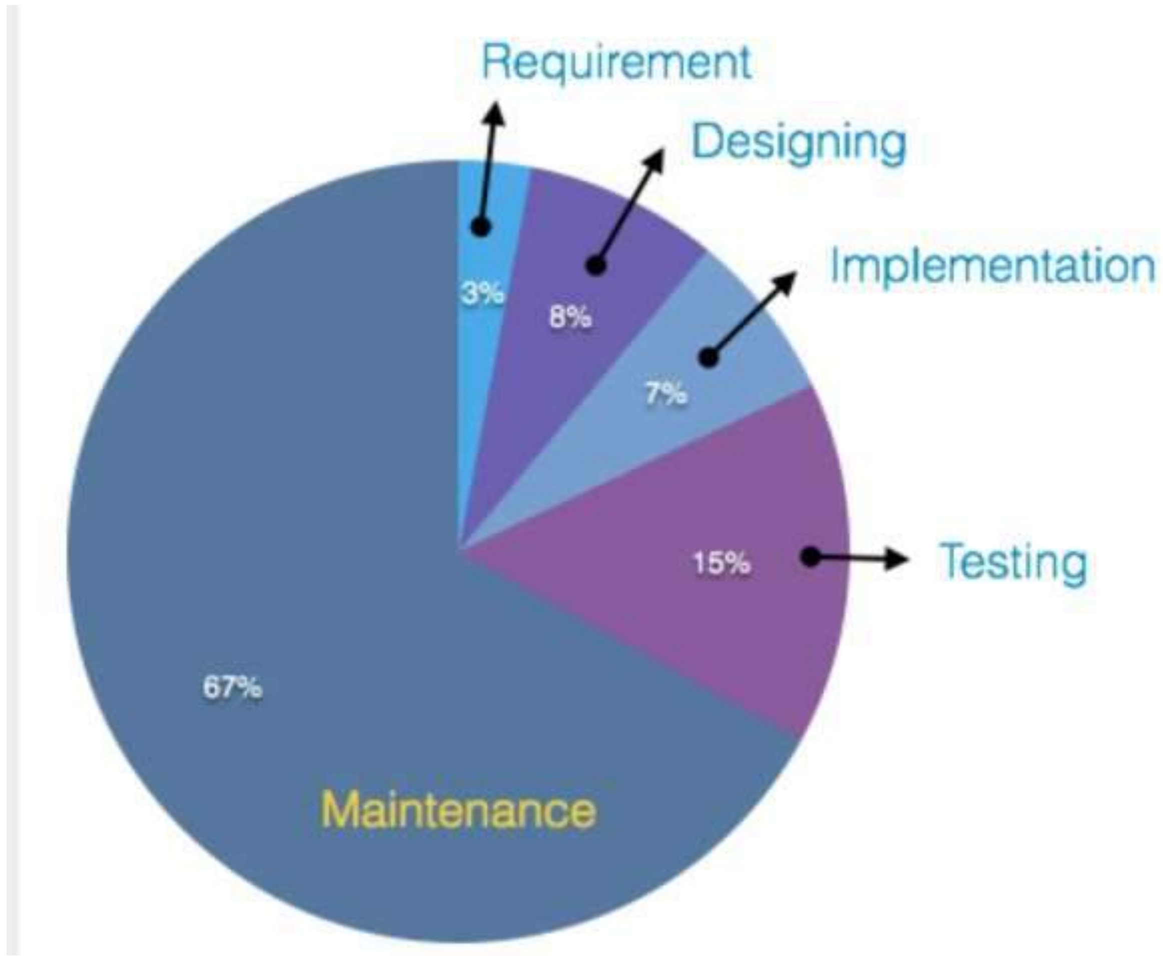
Testing

- Testing is important because software bugs could be expensive or even dangerous.
- Process of evaluating whether the current software product meets the requirements or not.
- Checks for missing requirements, bugs or errors, security, reliability and performance

Maintenance

- Goal is to modify and update software after delivery
  - Correcting errors
  - Improving performance or capabilities
  - Deletion of obsolete features
  - Optimization
- Types of Software Maintenance
  - Corrective
  - Adaptive
  - Preventive
  - Perfective

Cost Comparison over Phases



Software Retirement Process

- Application Decommission or Application sunsetting
- Final stage of life cycle
- Shutting down
- Reasons
  - Replaced
  - Release no longer supported
  - Redundant
  - Obsolete

Thank You

Some Techniques for Gathering and Analysing Requirements

- Observation
- Interviewing
- Brainstorming
- Prototyping

Gathering and Analysing Requirements

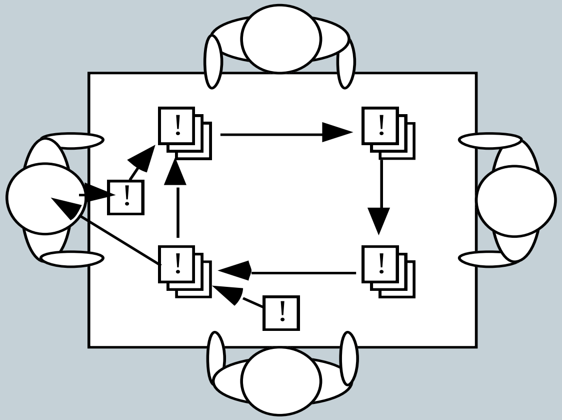
- Observation
  - May help to find details, that user may miss to tell
  - Shadowing important potential users as they do their work
    - ask the user to explain everything he or she is doing
  - Session videotaping
  - Consumes time, best for large projects

Gathering and Analysing Requirements

- Interviewing
  - Conduct a series of interviews
    - Ask about specific details
    - Ask about the stakeholder's vision for the future
    - Ask if they have alternative ideas
    - Ask minimum acceptable solution; ***shelf ware***
    - Ask for other sources of information
    - Ask them to draw diagrams
  - Listening skill and empathy
  - Make clear about interviewer knowledge
  - Don't make promises

Gathering and Analysing Requirements…

- Brainstorming
  - Appoint an experienced moderator
  - Arrange the attendees around a table
  - Decide on a ‘trigger question’
  - Ask each participant to write an answer and pass the paper to its neighbour



Gathering and Analysing Requirements…

- Brainstorming – Advantages
  - Spontaneous new ideas
  - Anonymity ensured
  - Ideas created in parallel
  - No need to wait for turn

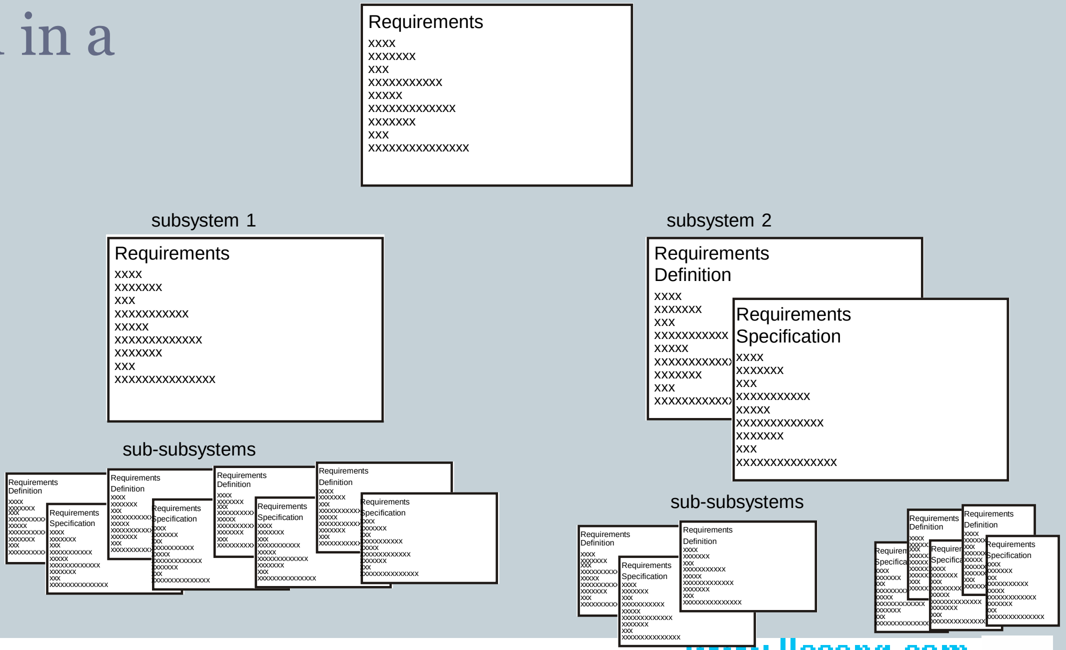
Gathering and Analysing Requirements…

- Prototyping
  - The simplest kind: *paper prototype*.
    - a set of pictures of the system that are shown to users in sequence to explain what would happen
  - The most common: a mock-up of the system’s UI
    - Written in a rapid prototyping language
    - Does *not* normally perform any computations, access any databases or interact with any other systems
    - Only a requirement gather*ing tool*

Types of Requirements Document

Two extremes:

- Requirements documents for large systems are normally arranged in a hierarchy



Level of detail required in a requirements document

- How much detail should be provided depends on:
  - The size of the system
  - The need to interface to other systems
  - The readership
  - The stage in requirements gathering
  - The level of experience with the domain and the technology
  - The cost that would be incurred if the requirements were faulty

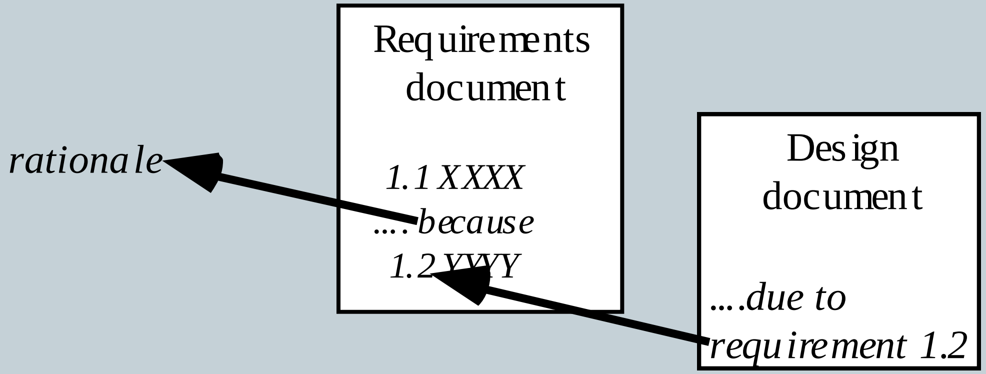
Reviewing Requirements

- **Each individual requirement should**
  - Have **benefits that outweigh the costs** of development
  - Be **important** for the solution of the current problem
  - Be expressed using a **clear and consistent notation**
  - Be **unambiguous**
  - Be **logically consistent**
  - Lead to a system of **sufficient quality**
  - Be **realistic** with available resources
  - Be **verifiable**
  - Be uniquely **identifiable**
  - **Does not over-constrain the design** of the system

Requirements documents...

- The document should be:
  - sufficiently complete
  - well organized
  - clear
  - agreed to by all the stakeholders

- Traceability:



Requirements document…

A. Problem
B. Background information
C. Environment and system models
D. Functional Requirements
E. Non-functional requirements

Managing Changing Requirements

- Requirements change because:
  - Business process changes
  - Technology changes
  - The problem becomes better understood

- Requirements analysis never stops
  - Continue to interact with the clients and users
  - The benefits of changes must outweigh the costs.
    - Certain small changes (e.g. look and feel of the UI) are usually quick and easy to make at relatively little cost.
    - Larger-scale changes have to be carefully assessed
      - Forcing unexpected changes into a partially built system will probably result in a poor design and late delivery
  - Some changes are enhancements in disguise
    - Avoid making the system *bigger*, only make it *better*

Use case Analysis

Use-Cases: describing how the user will use the system

- A ***use case*** is a typical sequence of actions that a user performs in order to complete a given task
  - The objective of ***use case analysis*** is to model the system from the point of view of

    … how users interact with this system

    … when trying to achieve their objectives.

    It is one of the key activities in **requirements analysis**
- **First step in use case analysis is to determine the types of users or other systems that will use the facilities of this system. These are called *actors*.**
  - A *use case model* consists of
    - a set of use cases
    - an optional description or diagram indicating how they are related

Use cases

- **The second step in use case analysis is to determine the tasks that each actor will need to do with the system.**
- Each task is a **use case**
- A use case should
  - Cover the *full sequence of steps* from the beginning of a task until the end.
  - Describe the *user's interaction* with the system …
    - Not the computations the system performs.
  - Be written so as to be as *independent* as possible from any particular user interface design.
  - Only include actions in which the actor interacts with the computer.
    - Not actions a user does manually

Use case Example

- *List a minimal set of use cases for the following actors in a library system:*
  - *Borrower, Checkout Clerk, Librarian*

Borrower:

- ❑ Search for items by title.
- ❑ … by author.
- ❑ … by subject.
- ❑ Check the borrower's personal information and list of books currently borrowed.

Checkout Clerk:

- ❑ All the Borrower use cases, plus
- ❑ Check out an item for a borrower.
- ❑ Check in an item that has been returned.

Use case Example....

- ❑ Renew an item.
- ❑ Record that a fine has been paid.
- ❑ Add a new borrower.
- ❑ Update a borrower's personal information (address, telephone number etc.).

Librarian:

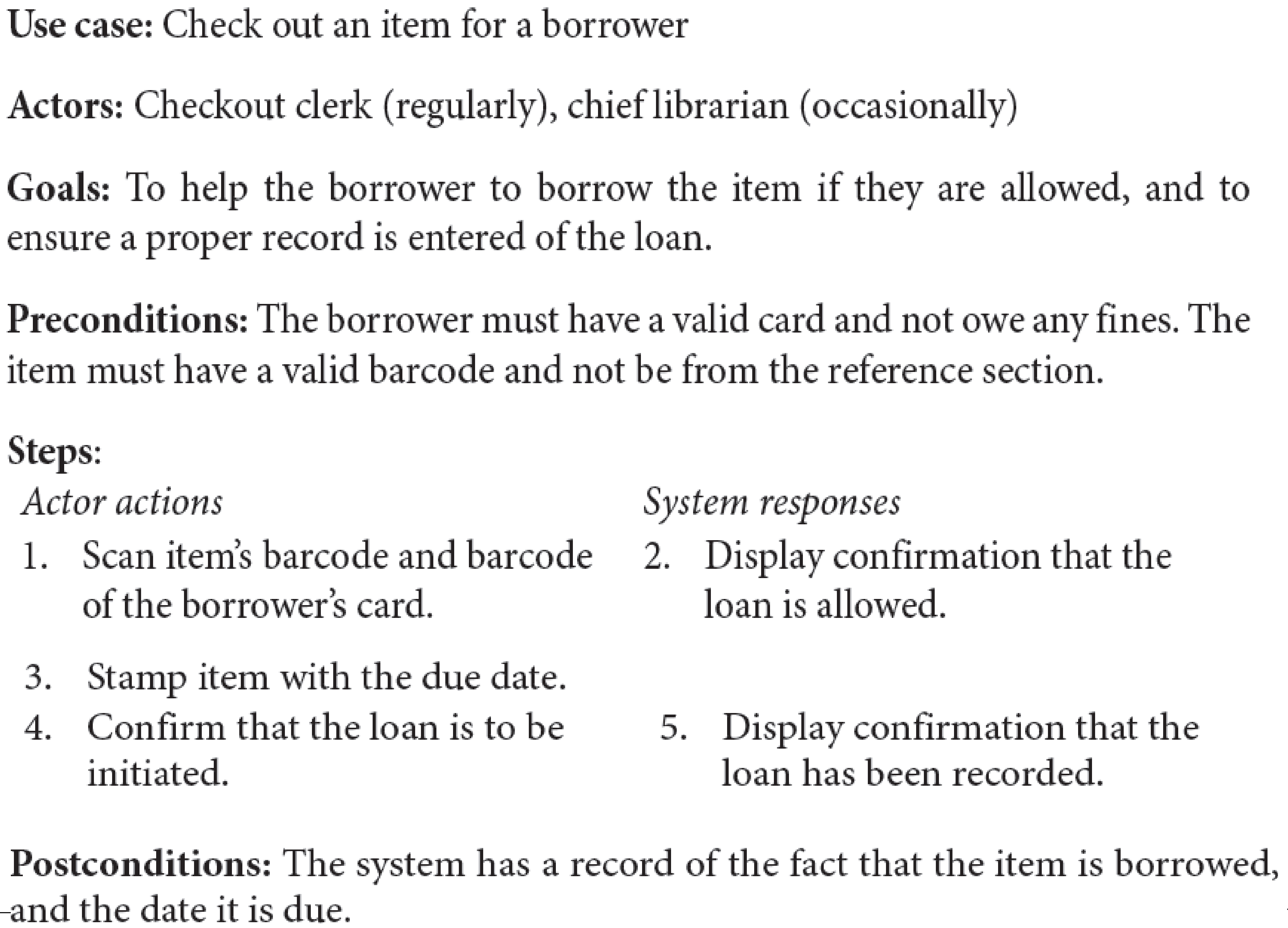
- ❑ All of the Borrower and Checkout Clerk use cases, plus
- ❑ Add a new item to the collection.
- ❑ Delete an item from the collection.
- ❑ Change the information the system has recorded about an item.

How to describe a single use case

- A**. Name**: Give a short, descriptive name to the use case.
- B. **Actors**: List the actors who can perform this use case.
- C. **Goals**: Explain what the actor or actors are trying to achieve.
- D. **Preconditions**: State of the system before the use case.
- E. **Summary**: Give a short informal description.
- F. **Related use cases**.
- G. **Steps**: Describe each step using a 2-column format. The left column showing the *actions taken by the actor*, and the right column showing *the system's responses*.
- H. **Post conditions**: State of the system in following completion.

- ***A and G are the most important***

Use Case Description: Example



Use case: Check out an item for a borrower

Actors: Checkout clerk (regularly), chief librarian (occasionally)

Goals: To help the borrower to borrow the item if they are allowed, and to ensure a proper record is entered of the loan.

Preconditions: The borrower must have a valid card and not owe any fines. The item must have a valid barcode and not be from the reference section.

Steps:

Actor actions	*System responses*
1. Scan item's barcode and barcode of the borrower's card.	2. Display confirmation that the loan is allowed.
3. Stamp item with the due date.	
4. Confirm that the loan is to be initiated.	5. Display confirmation that the loan has been recorded.

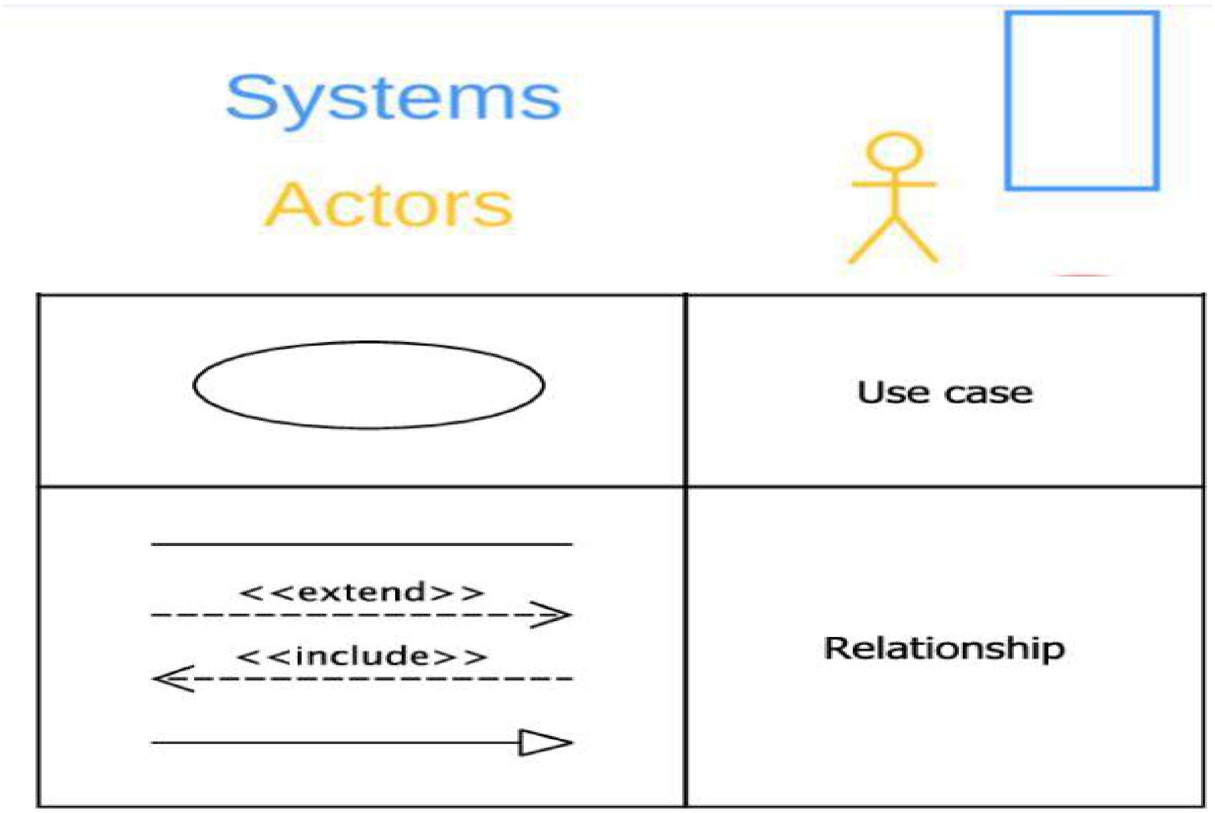
Postconditions: The system has a record of the fact that the item is borrowed, and the date it is due.

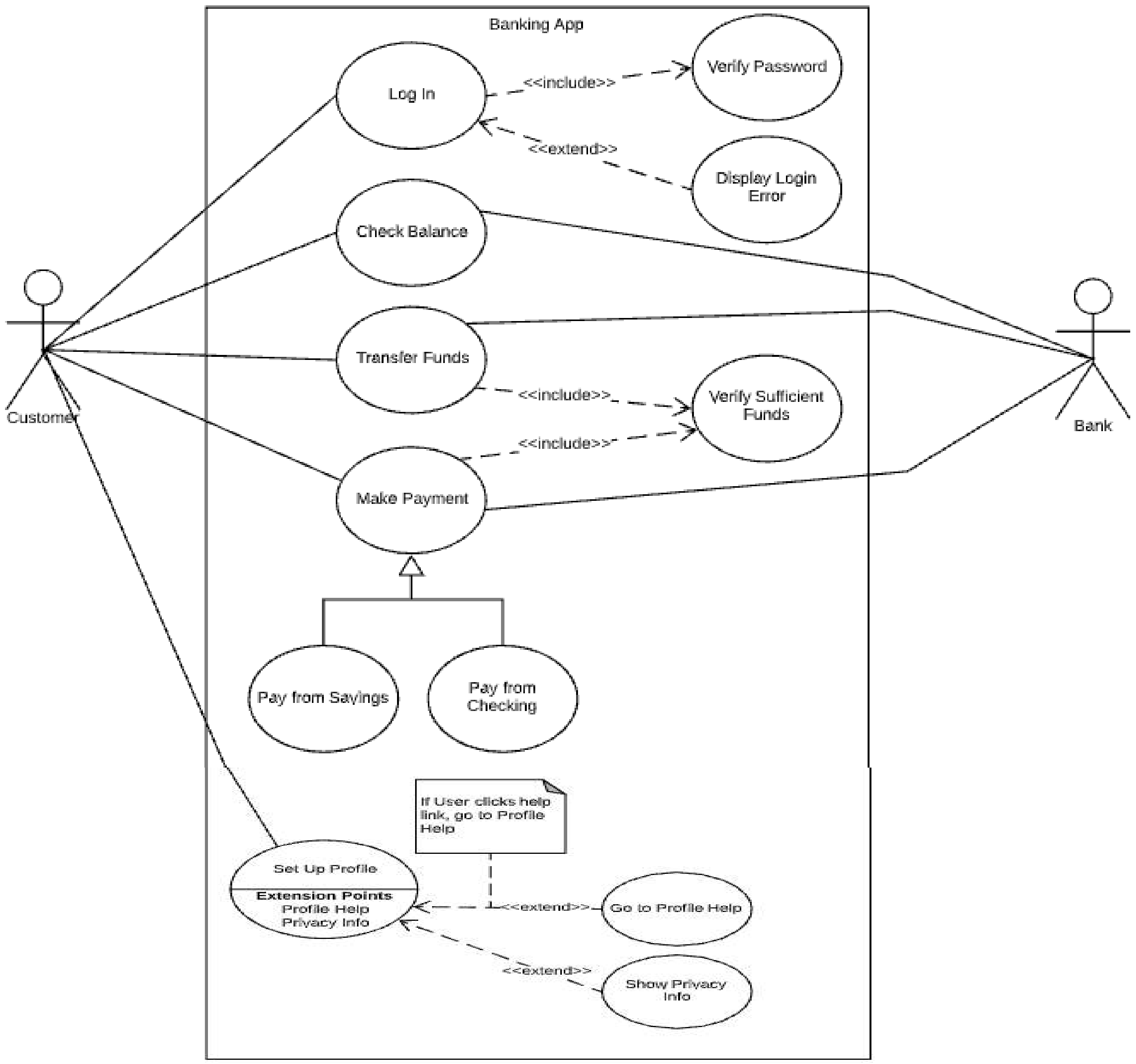
Use case diagrams

- Use case diagrams are UML's notation for showing the relationships among a set of use cases and actors.
- They help a software engineer to convey a high-level picture of the functionality of a system.
- there are two main symbols in use case diagrams:
  - *An actor is shown as a stick person*
  - *a use case is shown as an ellipse.*
- Lines indicate which actors perform which use cases.

Use case diagrams .....

Symbols used in Use case Diagram

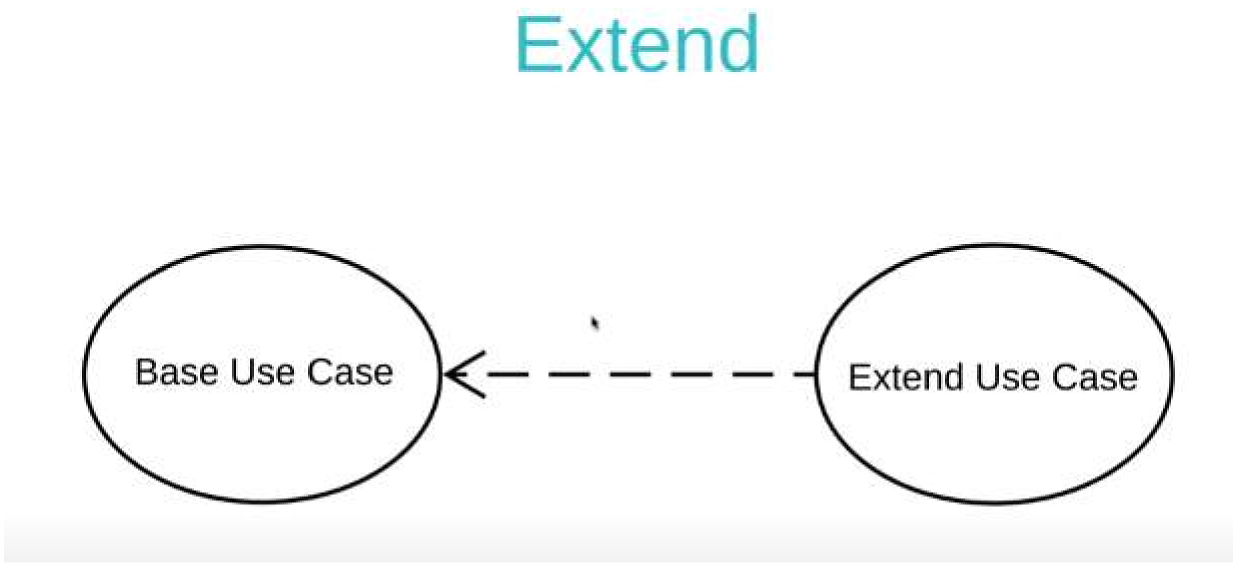



Banking App
Log In
<<include>>
Verify Password
<<extend>>
Display Login Error
Check Balance
Transfer Funds
<<include>>
Verify Sufficient Funds
<<include>>
Make Payment
Pay from Savings
Pay from Checking
If User clicks help link, go to Profile Help
Set Up Profile
Extension Points
Profile Help
Privacy Info
<<extend>>
Go to Profile Help
<<extend>>
Show Privacy Info
Customer
Bank

Use case diagrams….

Extensions

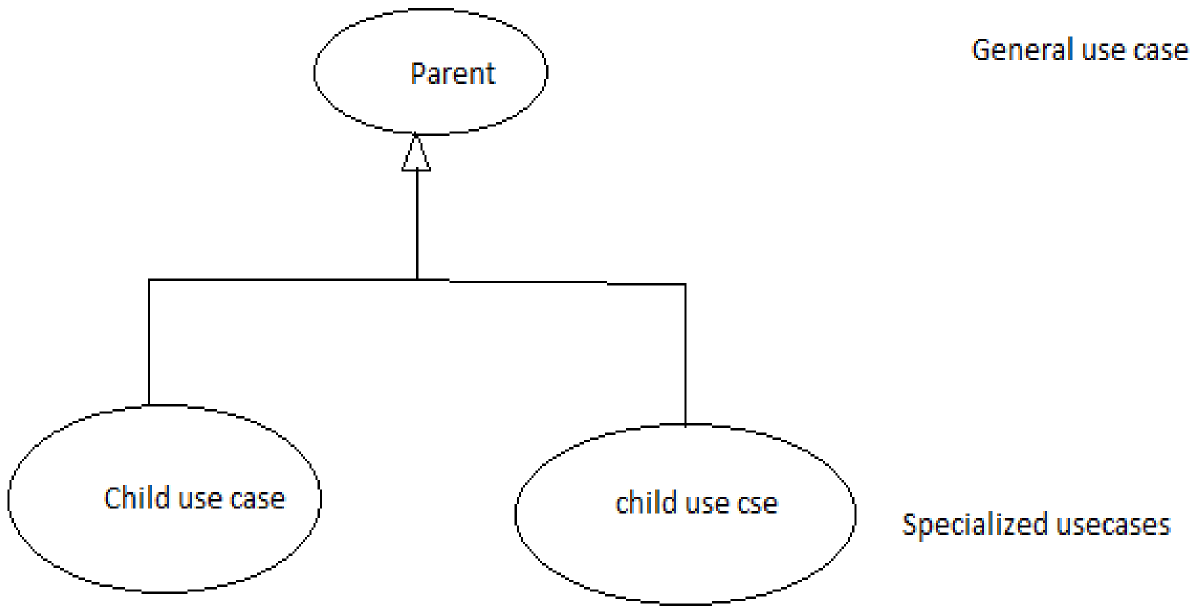
- Used to make *optional* interactions explicit or to handle *exceptional* cases.
- Keep the description of the basic use case simple.



Use case diagrams….

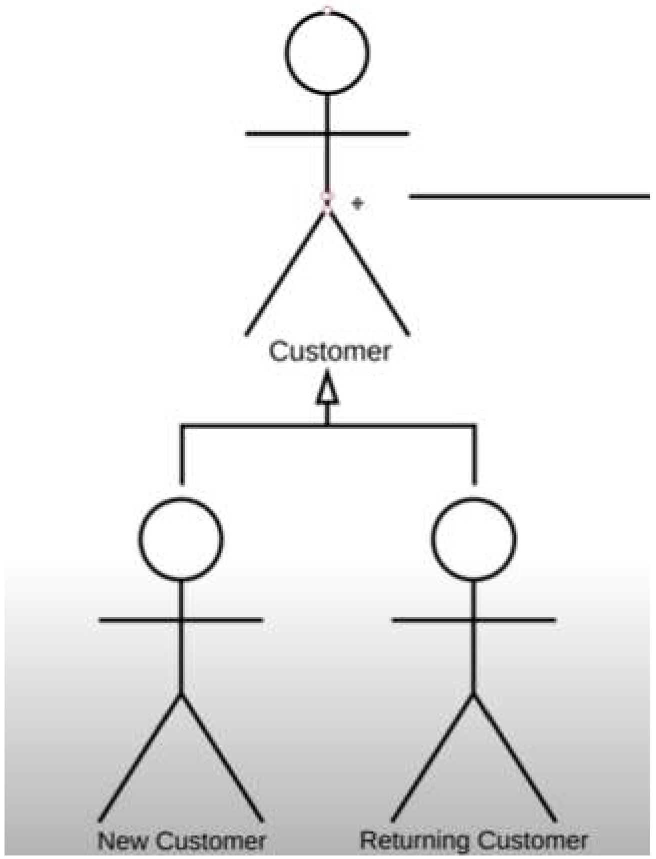
Generalizations

- **Also called inheritance**
- A generalized use case represents *several similar* use cases.
- One or more specializations provides details of the similar use cases.



Use case diagrams….

- Generalization of actors

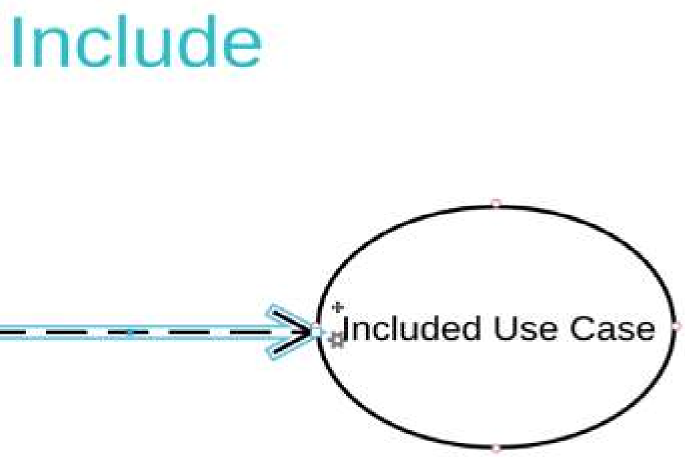


Use case diagrams….

Inclusions

- Show the relationship between base use case and included use case.
- Every time the base use case is executed ,the included use case is executed well.
- Allow one to express *commonality* between several different use cases.

- Are included in other use cases
  - Even very different use cases can share sequence of actions.
  - Enable you to avoid repeating details in multiple use cases.



The benefits of basing software development on use cases

- They can
  - Help to define the *scope* of the system
  - Be used to *plan* the development process
  - Be used to both develop and validate the requirements
  - Form the basis for the definition of test cases
  - Be used to structure user manuals

Use cases must not be seen as a panacea

- The use cases themselves must be validated
  - Using the requirements validation methods.
- Some aspects of software are not covered by use case analysis.
- Innovative solutions may not be considered.

Types of Requirements

What is a Requirement ?

- It is a statement describing either
  - 1) an aspect of what the proposed system must do,
  - or 2) a constraint on the system's development.
  - In either case it must contribute in some way towards *adequately solving the customer's problem*;
  - The set of requirements as a whole represents a negotiated agreement among the stakeholders.

- A collection of requirements is a ***requirements document***.

Requirement Engineering

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

Types of Requirements

- **Functional requirements**
  - Describe *what* the system should do
  - Statements of services the system should provide, how the system should react to particular inputs and how the system should react in particular situation.

- **Non-functional requirements**
  - C*onstraints* that must be adhered to during development
  - Constraint on the services or functions offered by the system such as timing constraints, platform constraints ,constraints on development etc.

Functional Requirements

– What *inputs* the system should accept

– What *outputs* the system should produce

– What data the system should *store* that other systems might use

– What *computations* the system should perform

– The *timing and synchronization* of the above

Non-functional requirements

- *All must be verifiable*
- Three main types

1. **Quality Requirements**

Categories reflecting: usability, efficiency, reliability, maintainability and reusability

- – Response time
- – Throughput
- – Resource usage
- – Reliability
- – Availability
- – Recovery from failure
- – Allowances for maintainability and enhancement
- – Allowances for reusability

Non-functional requirements...

2. **Platform Requirements**

Categories constraining the *environment and technology* of the system.

- Platform
- Technology to be used

3. **Process Requirements**

Categories constraining the *project plan and development methods*

- Development process (methodology) to be used
- Cost and delivery date
  - Often put in contract or project plan instead

Developing Requirements

Domain Analysis

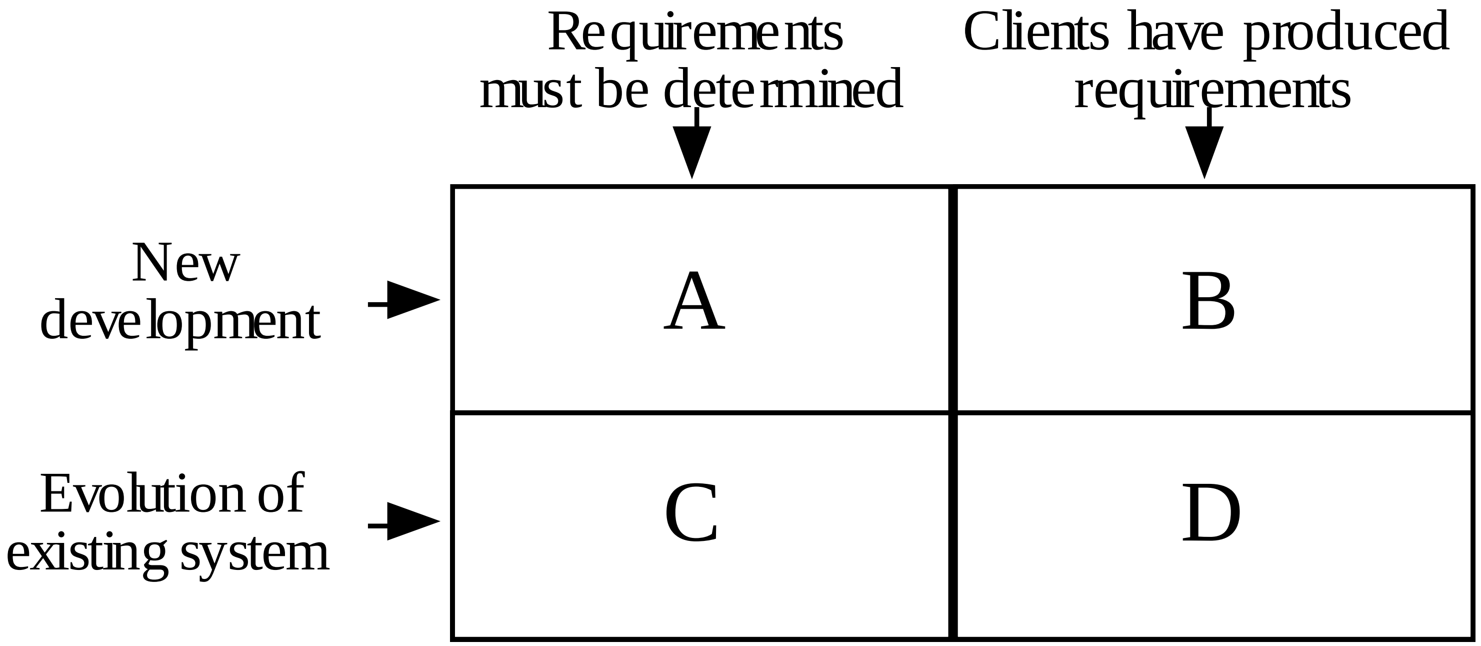
- The process by which a software engineer learns about the domain to better understand the problem:
  - The ***domain*** is the general field of business or technology in which the clients will use the software
  - A ***domain expert*** is a person who has a deep knowledge of the domain

- Benefits of performing domain analysis:
  - Faster development
  - Better system
  - Anticipation of extensions
- It is useful to write a summary of the information found during domain analysis. This is called ***Domain Analysis Document***.

Domain Analysis document

A. Introduction
B. Glossary
C. General knowledge about the domain
D. Customers and users
E. The environment
F. Tasks and procedures currently performed
G. Competing software
H. Similarities to other domains

The Starting Point for Software Projects



Requirements must be determined
Clients have produced requirements
New development
Evolution of existing system
A
B
C
D

Defining the Problem and the Scope

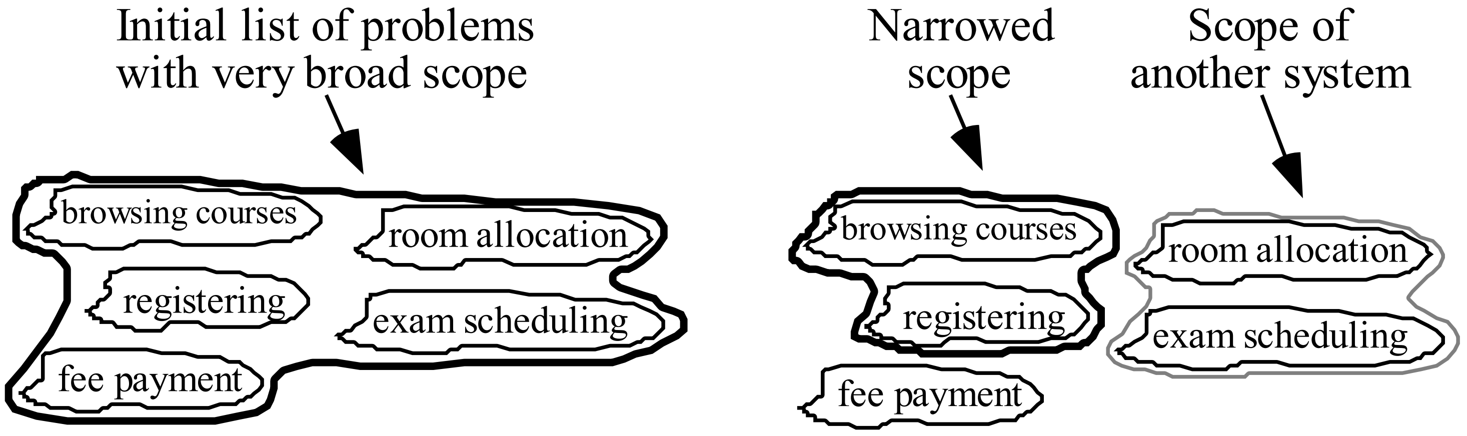
- A problem can be expressed as:
  - A *difficulty* the users or customers are facing,
  - Or as an *opportunity* that will result in some benefit such as improved productivity or sales.

- The solution to the problem normally will entail developing software

- A good problem statement is **short** and **succinct**

Defining the Scope

- Narrow the *scope* by defining a more precise problem
  - List all the things you might imagine the system doing
    - Exclude some of these things if too broad
    - Determine high-level goals if too narrow
- Example: A university registration system



MODULE 2

Principles of Software Design

The Process of Design

- Definition:
  - *Design* is a problem-solving process whose objective is to find and describe a way:
    - To implement the system's *functional requirements…*
    - While respecting the constraints imposed by the *quality, platform and process requirements…*
      - including the budget and deadlines
    - And while adhering to general principles of *good quality*

Design as a series of decisions

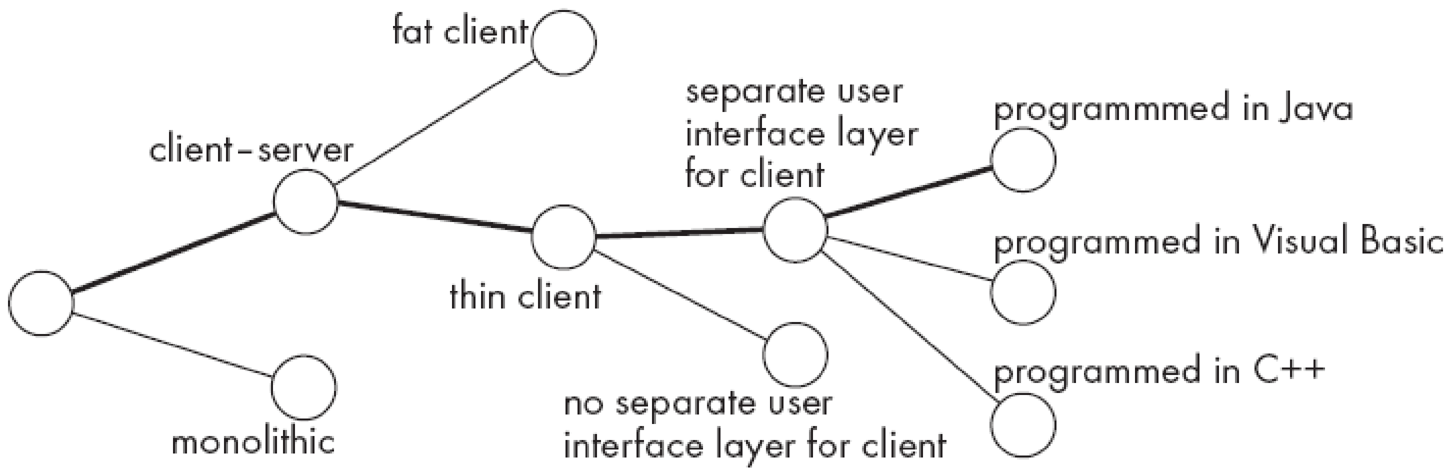
- A designer is faced with a series of *design issues*
  - These are sub-problems of the overall design problem.
  - Each issue normally has several alternative solutions:
    - design *options*.
  - The designer makes a *design decision* to resolve each issue.
    - This process involves choosing the best option from among the alternatives.

Making decisions

- To make each design decision, the software engineer uses:
  - Knowledge of
    - the requirements
    - the design as created so far
    - the technology available
    - software design principles and ‘best practices’
    - what has worked well in the past

Design space

- The space of possible designs that could be achieved by choosing different sets of alternatives is often called the *design space*
  - For example:


fat client
client-server
separate user interface layer for client
programmmed in Java
programmed in Visual Basic
thin client
programmed in C++
monolithic
no separate user interface layer for client

Parts of a system: subsystems, components and modules

Component

- Any piece of software or hardware that has a clear role.
    - A component can be isolated, allowing you to replace it with a different component that has equivalent functionality.
    - Many components are designed to be reusable.
    - Conversely, others perform special-purpose functions.

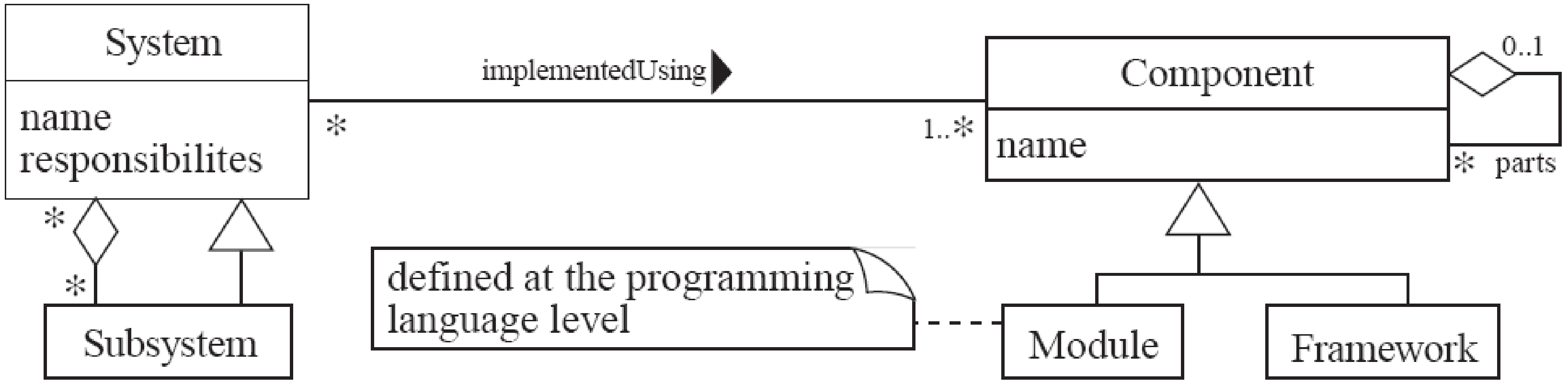
Module

- A component that is defined at the programming language level
  - For example, methods, classes and packages are modules in Java.

System

- A logical entity, having a set of definable responsibilities or objectives, and consisting of hardware, software or both.
  - A system can have a specification which is then implemented by a collection of components.
  - A system continues to exist, even if its components are changed or replaced.
  - The goal of requirements analysis is to determine the responsibilities of a system.
  - **Subsystem**:
    - A system that is part of a larger system, and which has a definite interface

UML diagram of system parts



Top-down and bottom-up design

- Top-down design
  - First design the very high level structure of the system.
  - Then gradually work down to detailed decisions about low-level constructs.
  - Finally arrive at detailed decisions such as:
    - the format of particular data items;
    - the individual algorithms that will be used.

Top-down and bottom-up design

- Bottom-up design
  - Make decisions about reusable low-level utilities.
  - Then decide how these will be put together to create high-level constructs.

- A mix of top-down and bottom-up approaches are normally used:
  - Top-down design is almost always needed to give the system a good structure.
  - Bottom-up design is normally useful so that reusable components can be created.

Different aspects of design

- *Architecture design*:
  - The division into subsystems and components,
    - How these will be connected.
    - How they will interact.
    - Their interfaces.
- *Class design*:
  - The various features of classes.
- *User interface design*
- *Algorithm design*:
  - The design of computational mechanisms.
- *Protocol design*:
  - The design of communications protocol.

Principles Leading to Good Design

- Overall *goals* of good design:
  - Increasing profit by reducing cost and increasing revenue
  - Ensuring that we actually conform with the requirements
  - Accelerating development
  - Increasing qualities such as
    - Usability
    - Efficiency
    - Reliability
    - Maintainability
    - Reusability

Design Principle 1: Divide and conquer

- Trying to deal with something big all at once is normally much harder than dealing with a series of smaller things
  - Separate people can work on each part.
  - An individual software engineer can specialize.
  - Each individual component is smaller, and therefore easier to understand.
  - Parts can be replaced or changed without having to replace or extensively change other parts.

Ways of dividing a software system

- A distributed system is divided up into clients and servers
- A system is divided up into subsystems
- A subsystem can be divided up into one or more packages
- A package is divided up into classes
- A class is divided up into methods

Design Principle 2: Increase cohesion where possible

- A subsystem or module has high cohesion if it keeps together things that are related to each other, and keeps out other things
- Cohesion is also called **Intra-Module Binding**.
  - This makes the system as a whole easier to understand and change
  - Type of cohesion:
    - Functional,
    - Layer
    - Communicational
    - Sequential
    - Procedural
    - Temporal
    - Utility

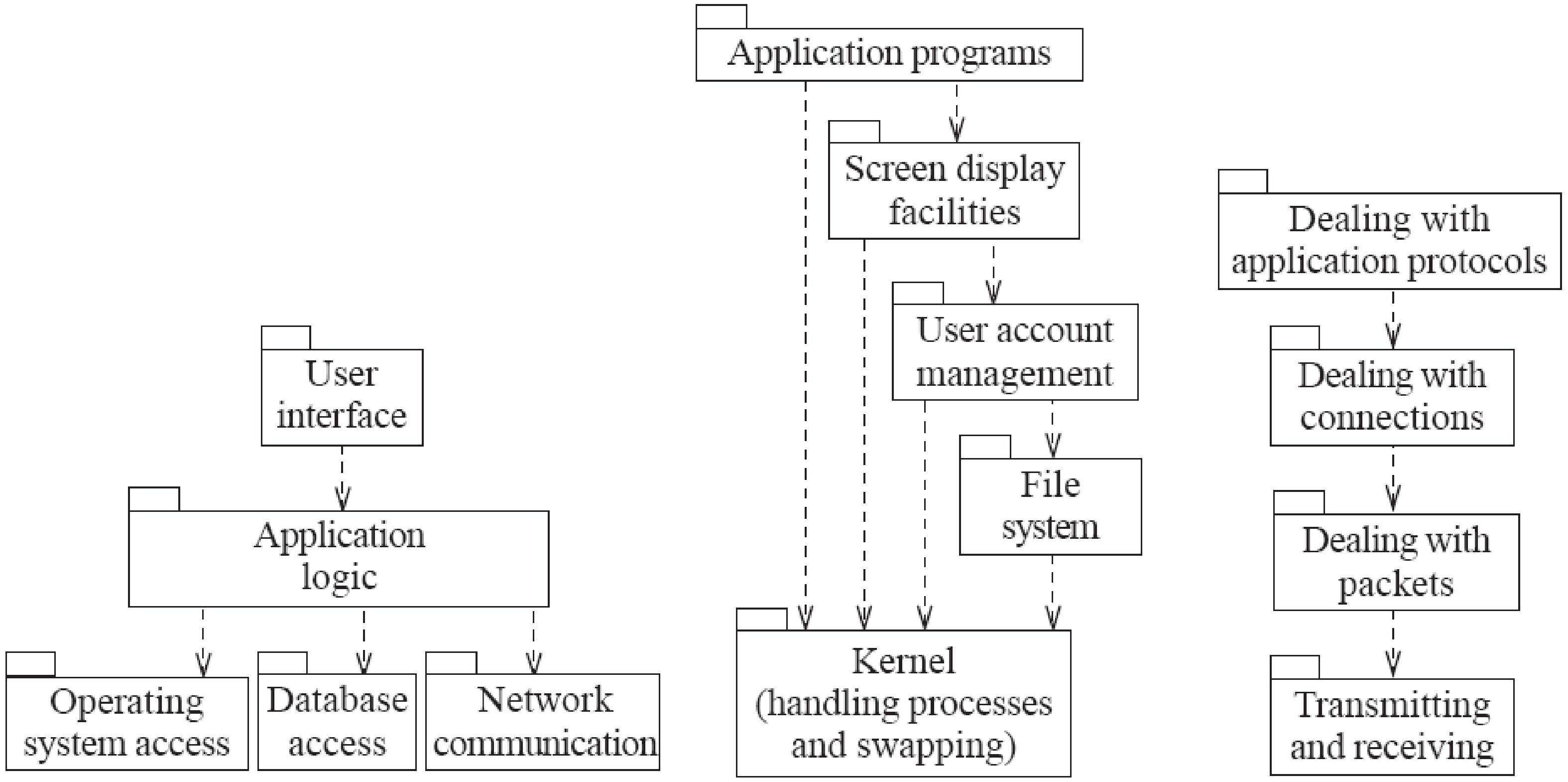
Functional cohesion

- This is achieved when *all the code that computes a particular result* is kept together - and everything else is kept out
    - i.e. when a module only performs a *single* computation, and returns a result, *without having side-effects*.
    - Benefits to the system:
        - Easier to understand
        - More reusable
        - Easier to replace
    - Modules that update a database, create a new file or interact with the user are not functionally cohesive

Layer cohesion

- All the *facilities for providing or accessing a set of related services* are kept together, and everything else is kept out
  - The layers should form a hierarchy
    - Higher layers can access services of lower layers,
    - Lower layers do not access higher layers
  - The set of procedures through which a layer provides its services is the *application programming interface (API)*
  - You can replace a layer without having any impact on the other layers
    - You just replicate the API

Example of the use of layers



(a) Typical layers in an application program

(b) Typical layers in an operating system

(c) Simplified view of layers in a communication system

Communicational cohesion

- All the *modules that access or manipulate certain data* are kept together (e.g. in the same class) - and everything else is kept out
  - A class would have good communicational cohesion
    - if all the system's facilities for storing and manipulating its data are contained in this class.
    - if the class does not do anything other than manage its data.
  - Main advantage: When you need to make changes to the data, you find all the code in one place

Sequential cohesion

- *Procedures, in which one procedure provides input to the next*, are kept together – and everything else is kept out
  - You should achieve sequential cohesion, only once you have already achieved the preceding types of cohesion.

Procedural cohesion

- *Procedures that are used one after another* are kept together
  - Even if one does not necessarily provide input to the next.
  - Weaker than sequential cohesion.
- Example of Procedural Cohesion
  - module write read and edit something
    - use out record
    - write out record
    - read in record
    - pad numeric fields with zeros
    - return in record

Temporal Cohesion

- Elements are involved in activities that are related in time
- *Operations that are performed during the same phase of the execution* of the program are kept together, and everything else is kept out
  - For example, placing together the code used during system start-up or initialization.
  - Weaker than procedural cohesion

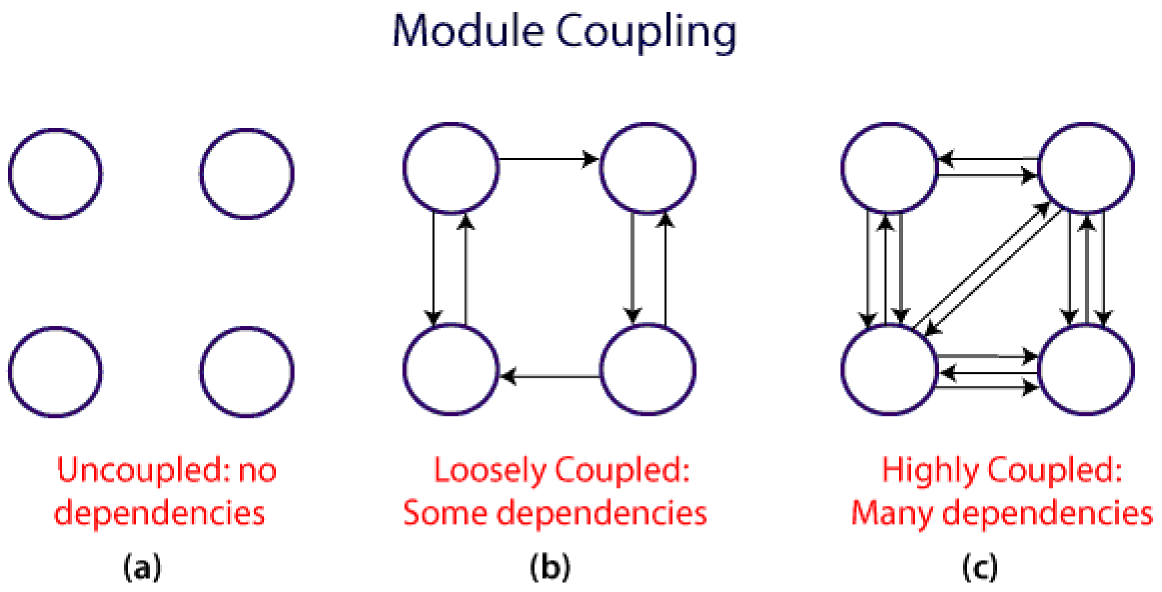
Utility cohesion

- When *related utilities which cannot be logically placed in other cohesive units* are kept together
  - A utility is a procedure or class that has wide applicability to many different subsystems and is designed to be reusable.
  - For example, the **`java.lang.Math`** class.

Design Principle 3: Reduce coupling where possible

- *Coupling* occurs when there are *interdependencies* between one module and another
  - When interdependencies exist, changes in one place will require changes somewhere else.
  - A network of interdependencies makes it hard to see at a glance how some component works.
  - Type of coupling:
    - Content,
    - Common,
    - Control
    - Stamp
    - Data
    - Routine Call
    - Type use
    - Inclusion/Import
    - External

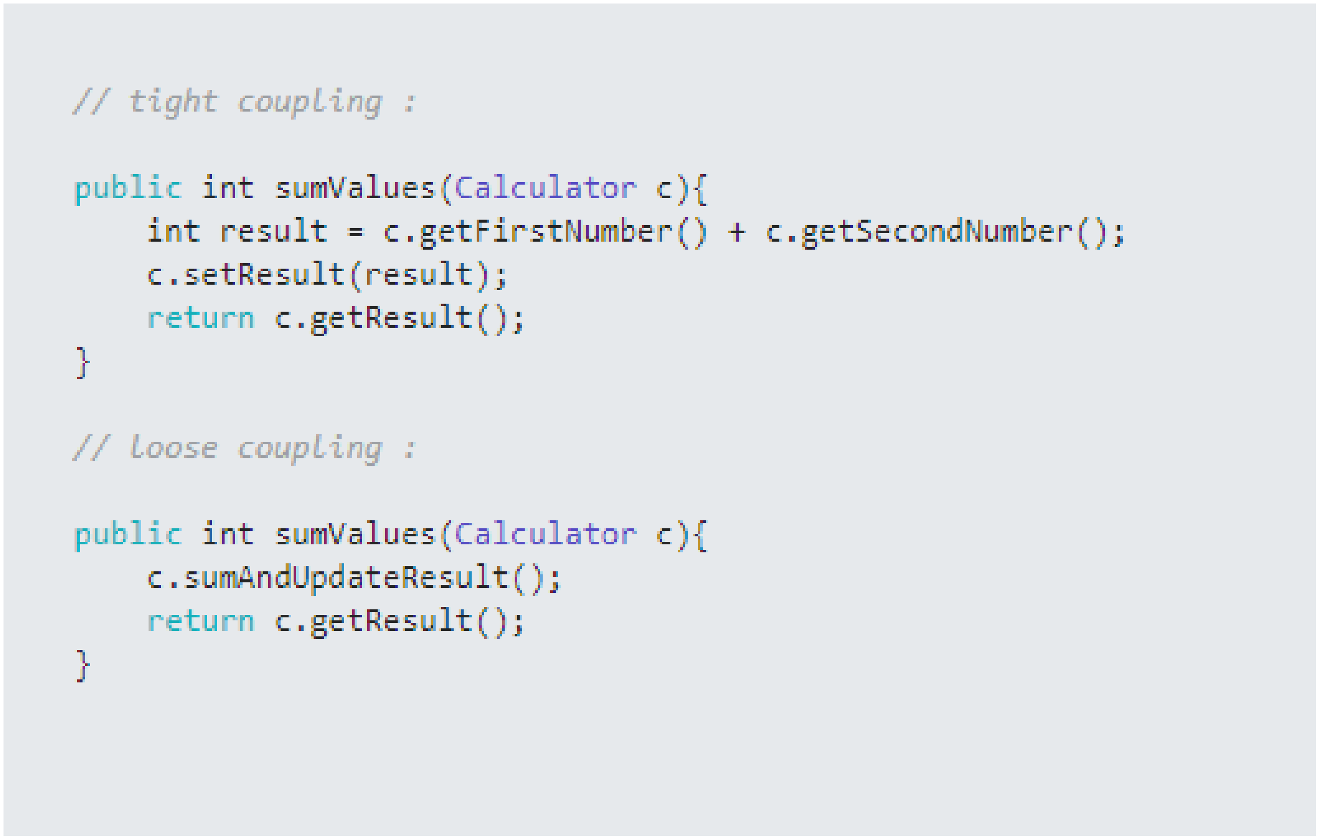
Tightly coupled system and a loosely coupled system


Module Coupling
Uncoupled: no dependencies
(a)
Loosely Coupled: Some dependencies
(b)
Highly Coupled: Many dependencies
(c)

Content coupling:

- Occurs when one component *surreptitiously* modifies data that is *internal* to another component .
  - To reduce content coupling you should therefore *encapsulate* all instance variables
    - declare them `private`
    - and provide get and set methods
  - This is the *worst form of coupling* and should be avoided.

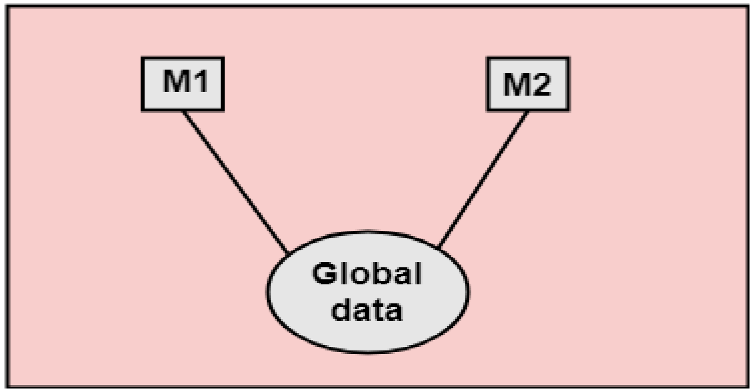
Example….



```
// tight coupling :

public int sumValues(Calculator c){
    int result = c.getFirstNumber() + c.getSecondNumber();
    c.setResult(result);
    return c.getResult();
}

// loose coupling :

public int sumValues(Calculator c){
    c.sumAndUpdateResult();
    return c.getResult();
}
```

Common coupling

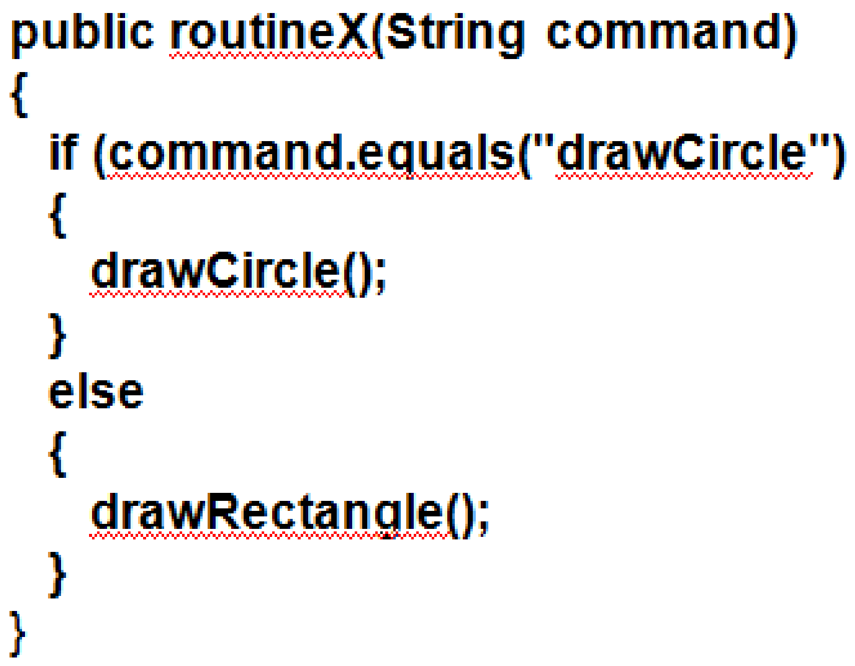
- Occurs whenever you use a *global variable*
  - All the components using the global variable become coupled to each other
  - A weaker form of common coupling is when a variable can be accessed by a *subset* of the system's classes.
    - e.g. a Java package
- Encapsulation reduces the harm of global variables.
    - avoid having too many such encapsulated variables.



Control coupling

- Communication between modules occur by passing control information(or a module control the flow of another)
- Occurs when one procedure calls another using *a 'flag' or 'command'* that explicitly controls what the second procedure does
  - To make a change you have to change both the calling and called method
  - The use of **polymorphic operations** is normally the best way to avoid control coupling
  - One way to reduce the control coupling could be to have **a *look-up table***
    - commands are then mapped to a method that should be called when that command is issued

Example of control coupling



```
public routineX(String  command)
{
   if (command.equals("drawCircle")
   {
      drawCircle();
   }
   else
   {
      drawRectangle();
   }
}
```

Stamp coupling:

- The complete data structure is passed from one module to another module
- Occurs whenever one of your application classes is declared as the *type* of a method argument
- In this case, this other class is tightly coupled to the first class, any change in the first class will affect the other class' function implementation
- Two ways to reduce stamp coupling,
  - using an interface as the argument type
  - passing simple variables

Example of stamp coupling

```
public class Emailer
{
  public void sendEmail(Employee e, String text)
  {...}
  ...
}
```

Using simple data types to avoid it:

```
public class Emailer
{
  public void sendEmail(String name, String email, String text)
  {...}
  ...
}
```

Example of stamp coupling…

Using an interface to avoid it:

```
public interface Addressee
{
  public abstract String getName();
  public abstract String getEmail();
}

public class Employee implements Addressee {…}

public class Emailer
{
  public void sendEmail(Addressee e, String text)
  {…}
  …
}
```

Data coupling

- Two modules exhibit *data coupling* if one calls the other directly and they communicate using “parameters" .
- Data passed using parameters.
- This coupling occurs when a function has got too many parameters.
- The downside of such coupling is that the callers of the function should pass all the arguments, even the ones that does not matter to them.
- **The more arguments a method has, the higher the coupling**
- **You should reduce coupling by not giving methods unnecessary arguments**

  - There is a trade-off between data coupling and stamp coupling
    - *Increasing one often decreases the other*

Routine call coupling

- Occurs when one routine (or method in an object oriented system) calls another
  - The routines are coupled because they depend on each other's behaviour
  - Routine call coupling is always present in any system.

  - If you repetitively use a sequence of two or more methods to compute something
    - then you can reduce routine call coupling by writing a single routine that encapsulates the sequence.

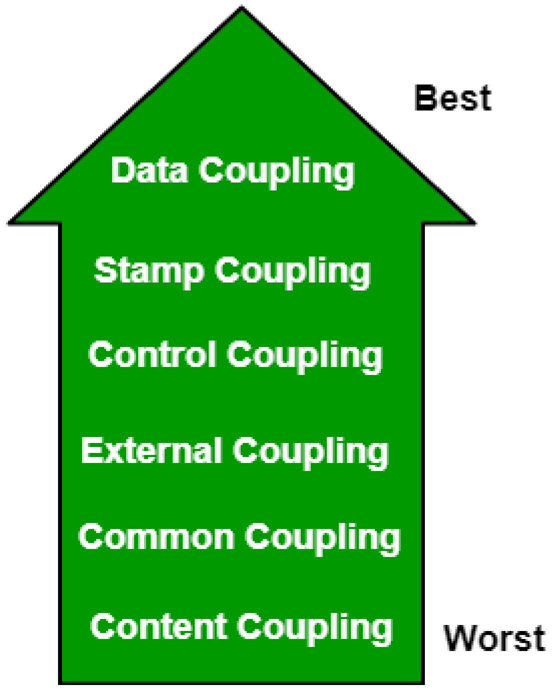
Inclusion or import coupling

- Occurs when one component imports a package
  - (as in Java)
- or when one component includes another
  - (as in C++).
  - The including or importing component is now exposed to everything in the included or imported component.
  - If the included/imported component changes something or adds something.
    - This may raises a conflict with something in the includer, forcing the includer to change.
  - An item in an imported component might have the same name as something you have already defined.

External coupling

- When a module has a dependency on such things as the operating system, shared libraries or the hardware
  - It is best to reduce the number of places in the code where such dependencies exist.
  - The Façade design pattern can reduce external coupling

Levels of Coupling



Design Principle 4: Keep the level of abstraction as high as possible

- An abstraction is a tool that enables a designer to consider a component at an abstract level without bothering about the internal details of the implementation.
- Ensure that your designs allow you to hide or defer consideration of details, thus reducing complexity
  - A good abstraction is said to provide *information hiding*
  - Abstractions allow you to understand the essence of a subsystem without having to know unnecessary details

Abstraction and classes

- Classes are data abstractions that contain procedural abstractions
  - Abstraction is increased by defining all variables as private.
  - The fewer public methods in a class, the better the abstraction
  - Superclasses and interfaces increase the level of abstraction
  - Attributes and associations are also data abstractions.
  - Methods are procedural abstractions
    - Better abstractions are achieved by giving methods fewer parameters

Design Principle 5: Increase reusability where possible

- Design the various aspects of your system so that they can be used again in other contexts
- strategies for increasing reusability are as follows:
  - Generalize your design as much as possible
  - Follow the preceding three design principles
  - Design your system to contain hooks
  - Simplify your design as much as possible

Design Principle 6: Reuse existing designs and code where possible

- Design with reuse is complementary to design for reusability
  - Actively reusing designs or code allows you to take advantage of the investment you or others have made in reusable components
    - *Cloning* should not be seen as a form of reuse

Design Principle 7: Design for flexibility

- Also known as adaptability
- Actively anticipate changes that a design may have to undergo in the future, and prepare for them.
  - Reduce coupling and increase cohesion
  - Create abstractions
  - Do not hard-code anything
  - Leave all options open
    - Do not restrict the options of people who have to modify the system later
  - Use reusable code and make code reusable

Design Principle 8: Anticipate obsolescence

- Plan for changes in the technology or environment so the software will continue to run or can be easily changed
- The following are some rules that designers can use to better anticipate obsolescence:
  - Avoid using early releases of technology
  - Avoid using software libraries that are specific to particular environments
  - Avoid using undocumented features or little-used features of software libraries
  - Avoid using software or special hardware from companies that are less likely to provide long-term support
  - Use standard languages and technologies that are supported by multiple vendors

Design Principle 9: Design for Portability

- Have the software run on as many platforms as possible
  - Avoid the use of facilities that are specific to one particular environment
  - E.g. a library only available in Microsoft Windows

Design Principle 10: Design for Testability

- Take steps to make testing easier
  - Design a program to automatically test the software
    - Ensure that all the functionality of the code can by driven by an external program, bypassing a graphical user interface
  - In Java, you can create a main() method in each class in order to exercise the other methods

Design Principle 11: Design defensively

- Never trust how others will try to use a component you are designing
  - Handle all cases where other code might attempt to use your component inappropriately
  - Check that all of the inputs to your component are valid: the *preconditions*
    - Unfortunately, over-zealous defensive design can result in unnecessarily repetitive checking

Design by contract

- A technique that allows you to design defensively in an efficient and systematic way
  - Key idea
    - each method has an explicit *contract* with its callers
  - The contract has a set of assertions that state:
    - What *preconditions* the called method requires to be true when it starts executing
    - What *postconditions* the called method agrees to ensure are true when it finishes executing
    - What *invariants* the called method agrees will not change as it executes

Design –
Object Oriented Approach

We have seen …

- Requirement Specification – SRS
- Design Principles
  - Cohesion
  - Coupling
  - Modularity

Design Approach

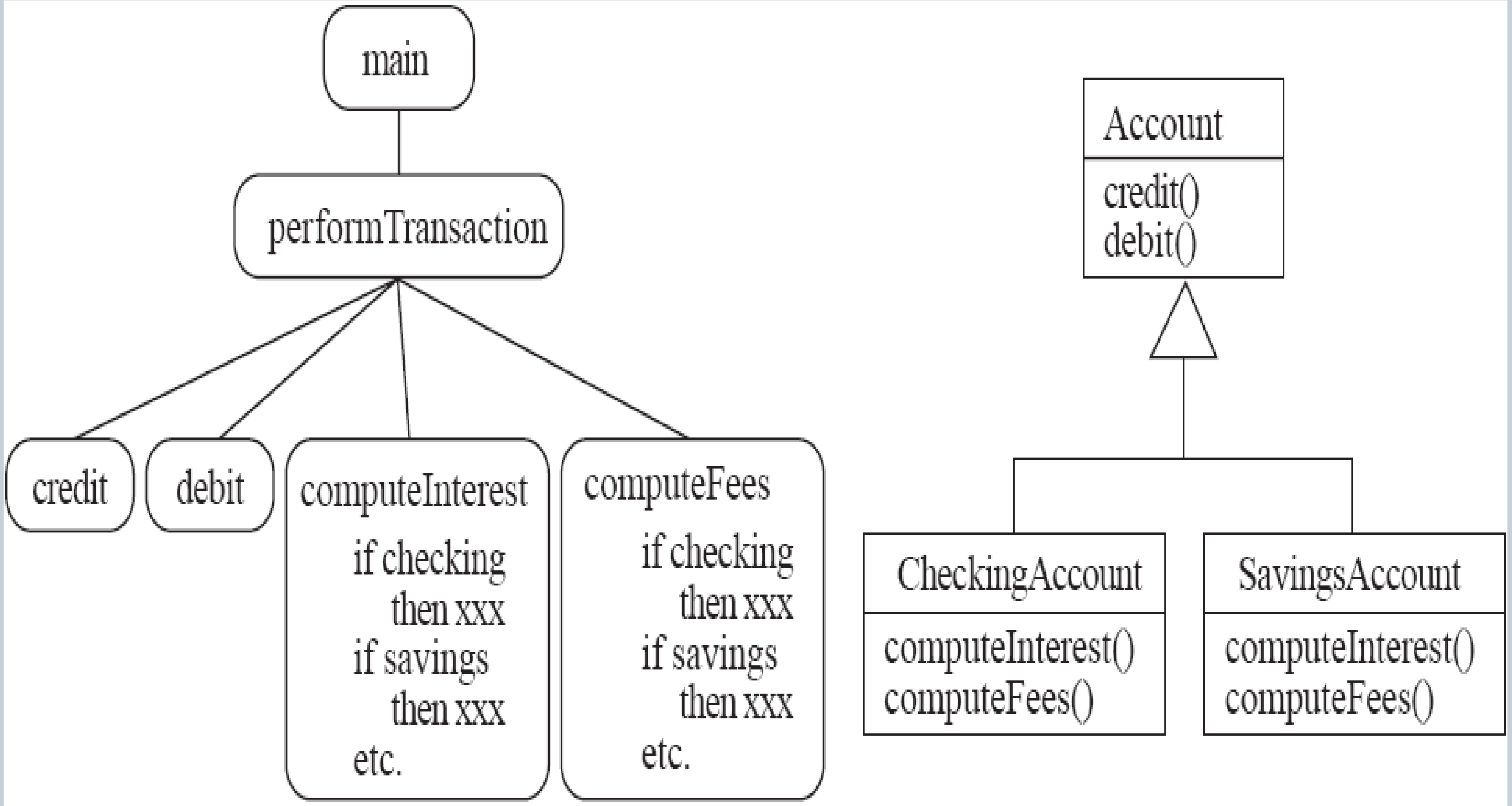
- Modelling functionalities – Starting point of design
- Design can be approached in different ways
  - Conventional functional oriented or procedural approach
    - Top-down approach
    - Structured Analysis
    - Data flow diagrams, structure chart
  - Object oriented approach
    - Bottom up approach
    - UML for modelling

- Procedural paradigm:
  - Software is organized around the notion of *procedures*
  - *Procedural abstraction*
    - Works as long as the data is simple
  - *Adding data abstractions*
    - Groups together the pieces of data that describe some entity
    - Helps reduce the system's complexity.
      - Such as *Records* and *structures*
- Object oriented paradigm:
  - Organizing procedural abstractions in the context of data abstractions

Object Oriented paradigm

- An approach to the solution of problems in which all computations are performed in the context of objects.
  - The objects are instances of classes, which:
    - are data abstractions
    - contain procedural abstractions that operate on the objects
  - A running program can be seen as a collection of objects collaborating to perform a given task

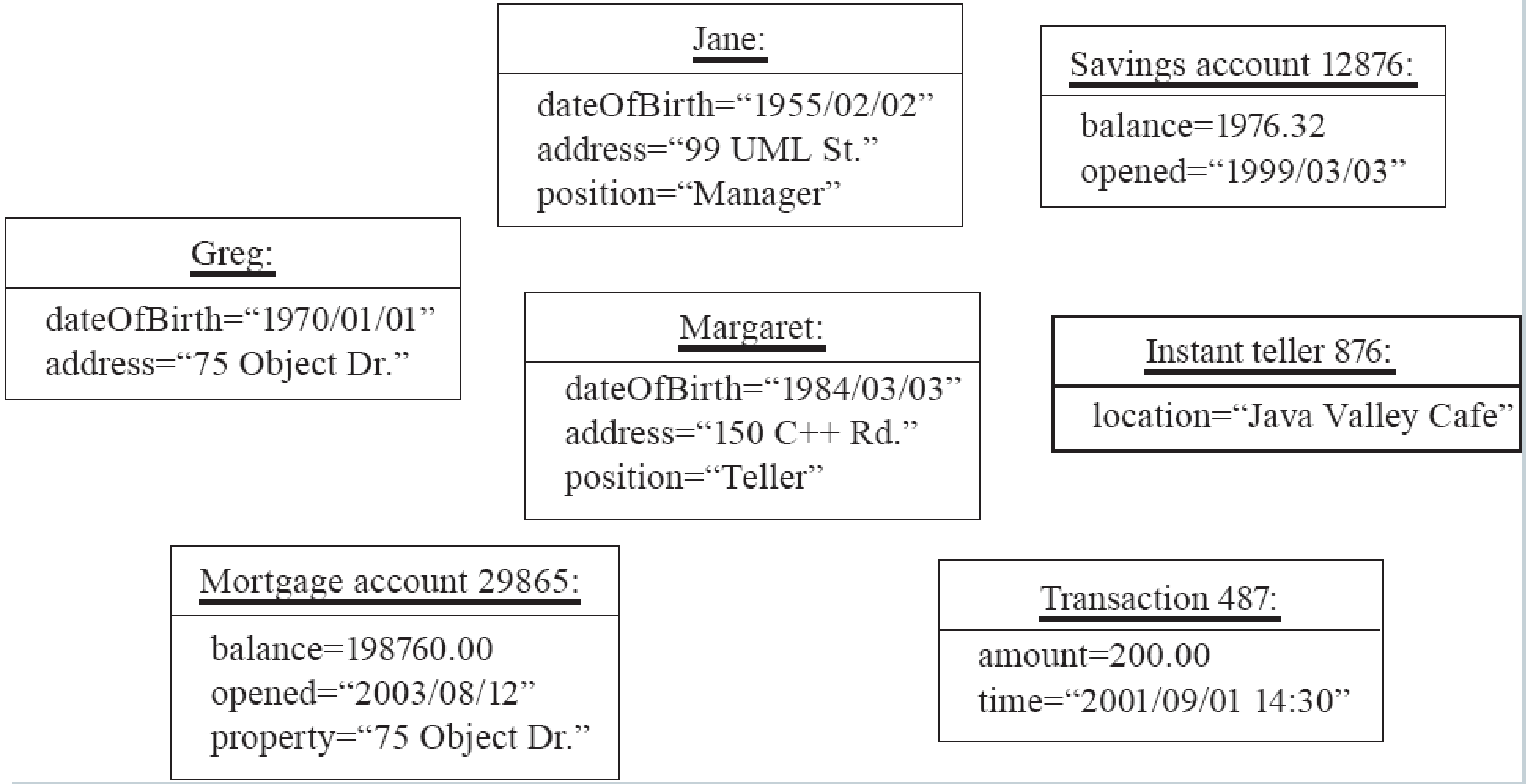
A View of the Two paradigms



Classes and Objects

- Object
  - A chunk of structured data in a running software system
  - Has *properties*
    - Represent its state
  - Has *behaviour*
    - How it acts and reacts
    - May simulate the behaviour of an object in the real world

Objects


Jane:
dateOfBirth="1955/02/02"
address="99 UML St."
position="Manager"
Savings account 12876:
balance=1976.32
opened="1999/03/03"
Greg:
dateOfBirth="1970/01/01"
address="75 Object Dr."
Margaret:
dateOfBirth="1984/03/03"
address="150 C++ Rd."
position="Teller"
Instant teller 876:
location="Java Valley Cafe"
Mortgage account 29865:
balance=198760.00
opened="2003/08/12"
property="75 Object Dr."
Transaction 487:
amount=200.00
time="2001/09/01 14:30"

Classes

- A class:
  - A unit of abstraction in an object oriented (OO) program
  - Template or blueprint of objects
  - A kind of software module
    - Describes its object structure (properties)
    - Contains *methods* to implement their behaviour

Is Something a Class or an Instance?

- Something should be a *class* if it could have instances
- Something should be an *instance* if it is clearly a *single* member of the set defined by a class

- *Student*
  - Class; instances are individual students.
- Course
  - Class; instance is CS3004 SE
- *Teacher with Employee ID – CSED100*
  - Instance of `Class Teacher`

Instance Variables

- Variables defined inside a class corresponding to data present in each instance
  - Also called *fields* or *member variables*
  - Attributes
    - Simple data
    - E.g. `name`, `dateOfBirth`
  - Associations
    - Relationships to other important classes
    - E.g. `supervisor`, `coursesTaken`

Variables vs. Objects

- A variable
  - *Refers* to an object
  - May refer to different objects at different points in time
  - Eg:

```
Student s1 = new Student();
s1.name = "ABC";
s1.computeGrade();
```

- An object can be referred to by several different variables at the same time

```
Student s2 = new Student();
s2 = s1;
```

- *Type* of a variable
  - Determines what classes of objects it may contain

Methods and Polymorphism

- Method
  - A procedural abstraction used to implement the behaviour of a class
  - Several different classes can have methods with the same name
    - They implement the same abstract operation in ways suitable to each class
    - E.g. calculating area in a rectangle is done differently from in a circle

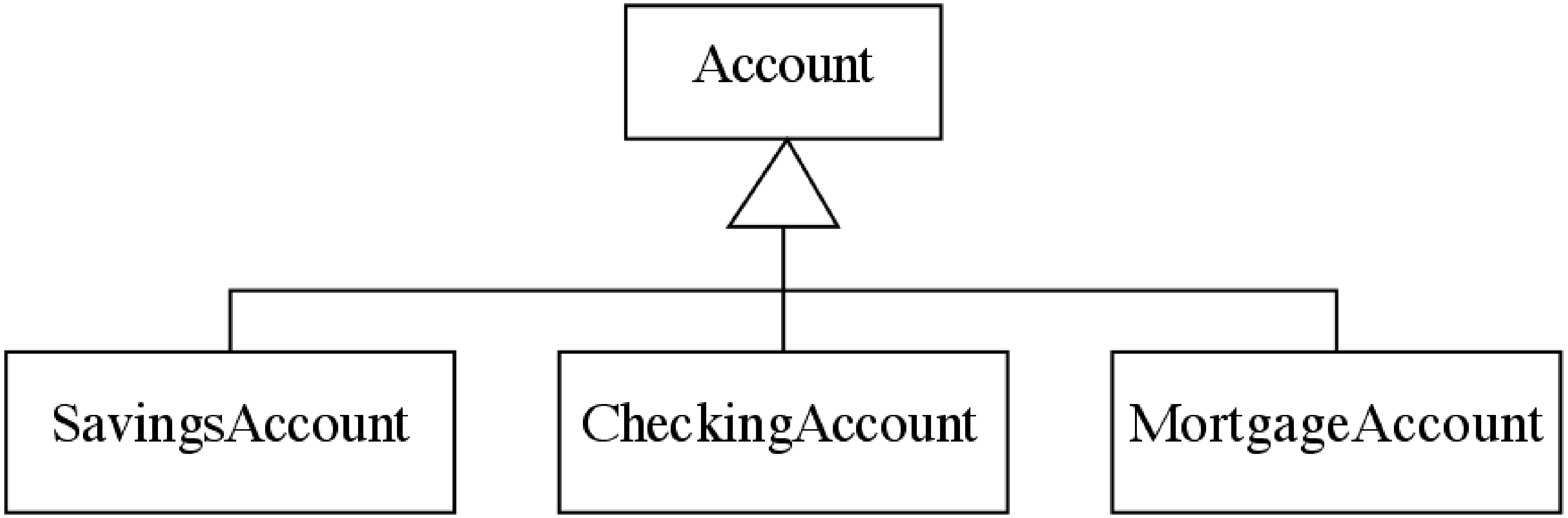
Polymorphism

- A property of object oriented software by which an *abstract operation may be performed in different ways* in different classes.
  - Requires that there be *multiple methods of the same name*
  - The choice of which one to execute depends on the object that is in a variable
  - Reduces the need for programmers to code many `if-else` or `switch` statements

Organizing Classes into Inheritance Hierarchies

- Superclasses
  - Contain features common to a set of subclasses
- Inheritance hierarchies
  - Show the relationships among superclasses and subclasses
  - A triangle shows a *generalization* △
- Inheritance
  - Subclasses implicitly have the features defined in its superclasses

An Example Inheritance Hierarchy

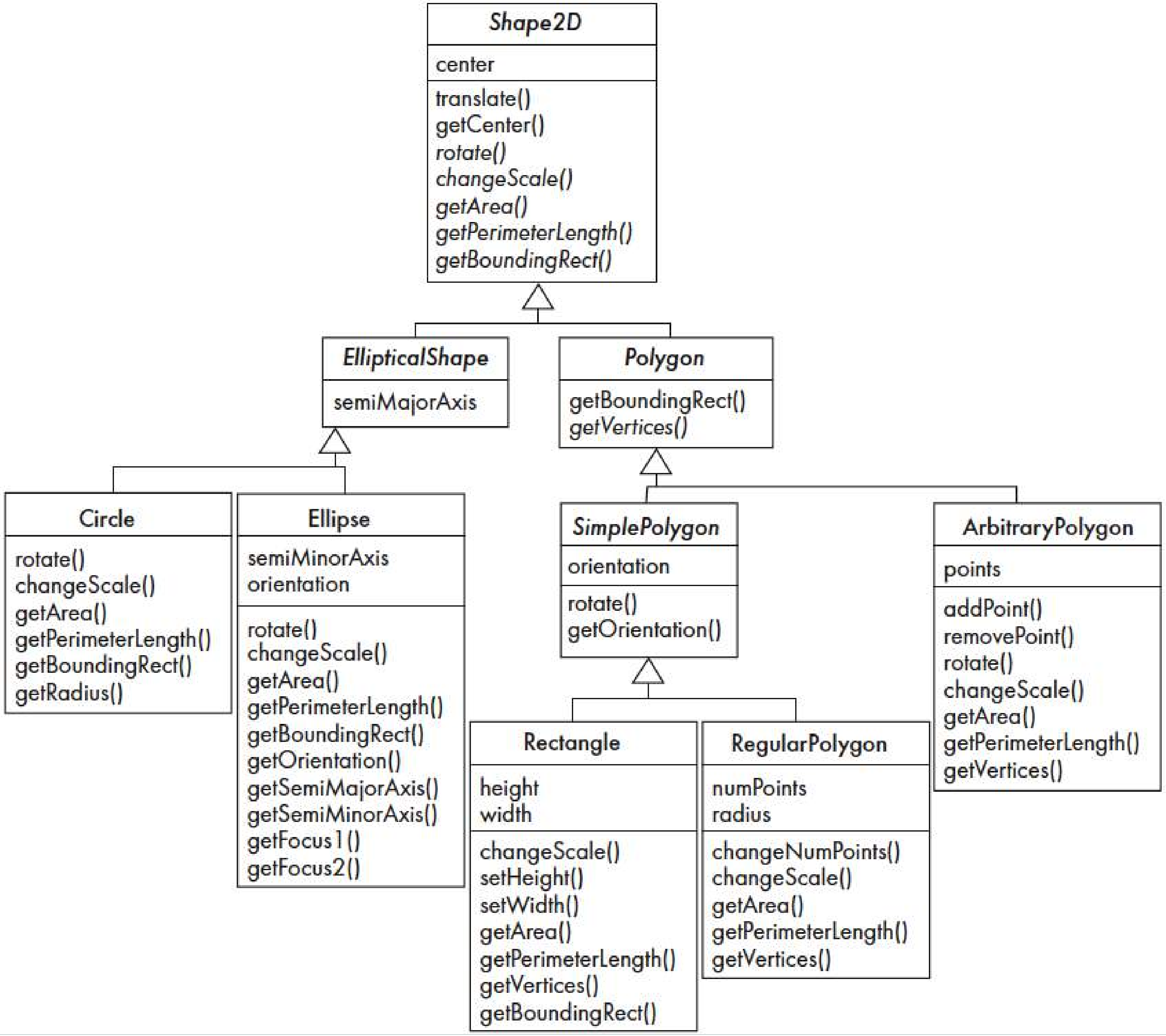


- Inheritance
  - Subclasses implicitly have the features defined in its superclasses

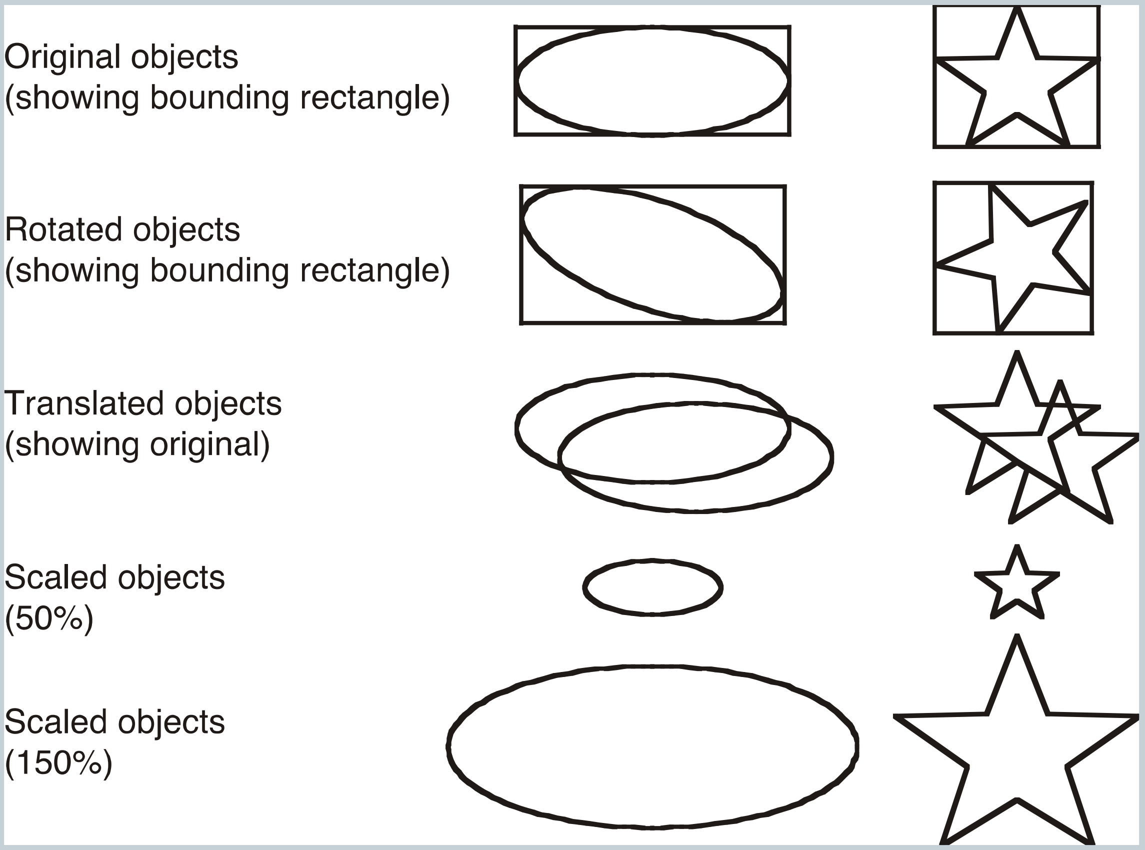
The Is A Rule

- Always check generalizations to ensure they obey the isa rule
  - "A checking account ***is an*** account"
  - "A student is a person"
- Is 'State' a subclass of 'Country'?
  - No, it violates the is a rule

Inheritance, Polymorphism and Variables



Some Operations in the Shape Example



Abstract Classes and Methods

- An operation should be declared to exist at the highest class in the hierarchy where it makes sense
  - The *operation* may be *abstract* (lacking implementation) at that level
  - If so, the *class* also <u>must</u> be *abstract*
    - No instances can be created
    - The opposite of an abstract class is a *concrete* class
  - If a superclass has an abstract operation then its subclasses at some level must have a concrete method for the operation
    - Leaf classes must have or inherit concrete methods for all operations
    - Leaf classes must be concrete

Overriding

- A method would be inherited, but a subclass contains a new version instead
  - For restriction
  - For extension
  - For optimization

How a decision is made on which method to run?

```
Rectangle obj = new Rectangle();
obj.getBoundingRect();
Shape2D obj= new Rectangle();
 obj.getBoundingRect();
```

1. If there is a concrete method for the operation in the current class, run that method.
2. Otherwise, check in the immediate superclass to see if there is a method there; if so, run it.
3. Repeat step 2, looking in successively higher superclasses until a concrete method is found and run.
4. If no method is found, then there is an error

Dynamic binding

- Occurs when decision about which method to run can only be made at *run time*
  - Needed when:
    - A variable is declared to have a superclass as its type, and
    - There is more than one possible polymorphic method that could be run among the type of the variable and its subclasses

Concepts that Define Object Orientation

- The following are necessary for a system or language to be OO
  - Identity
    - Each object is *distinct* from each other object, and *can be referred to*
    - Two objects are distinct *even if they have the same data*
  - Classes
    - The code is organized using classes, each of which describes a set of objects
  - Inheritance
    - The mechanism where features in a hierarchy inherit from superclasses to subclasses
  - Polymorphism
    - The mechanism by which several methods can have the same name and implement the same abstract operation.

- Java basics – video uploaded