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

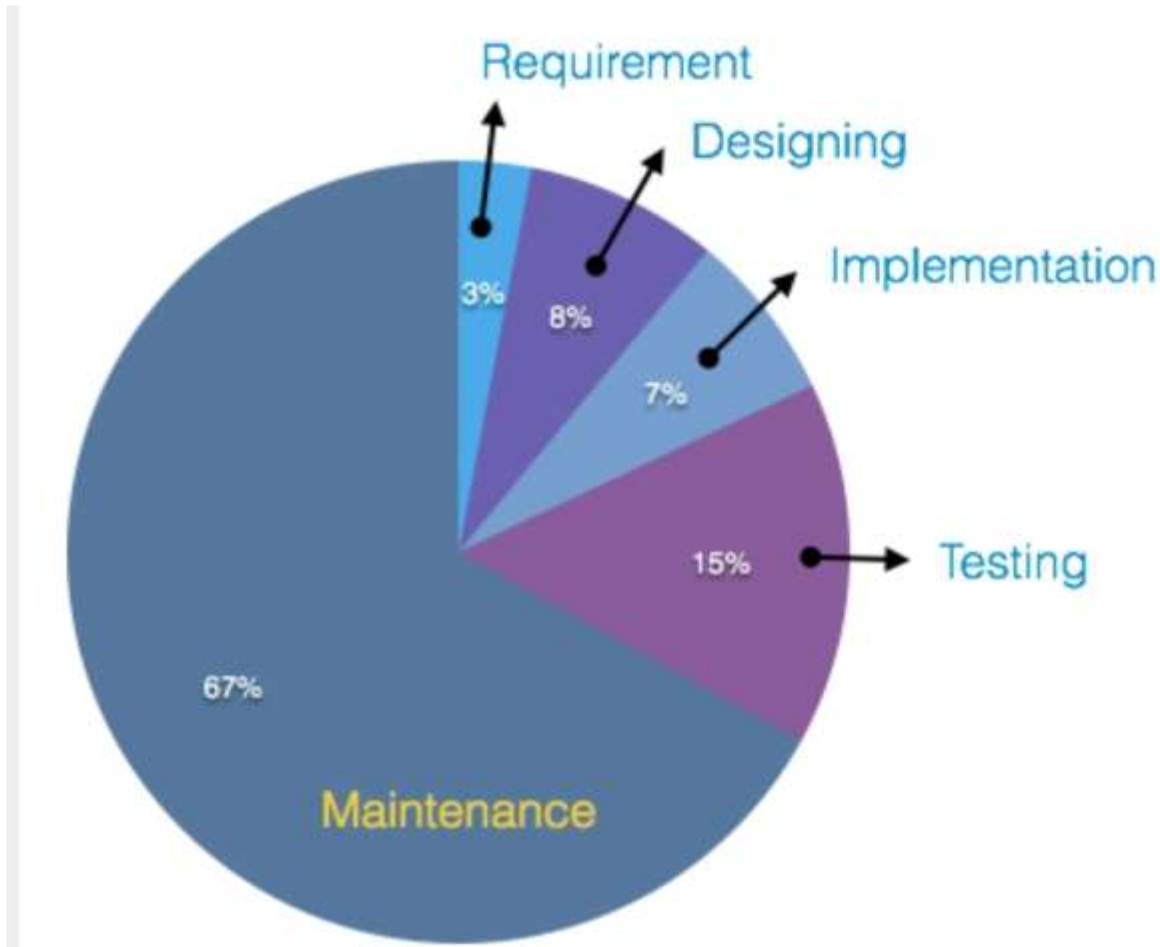
Testing

- 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

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	A	B
Evolution of existing system	C	D

Defining the Problem and the Scope

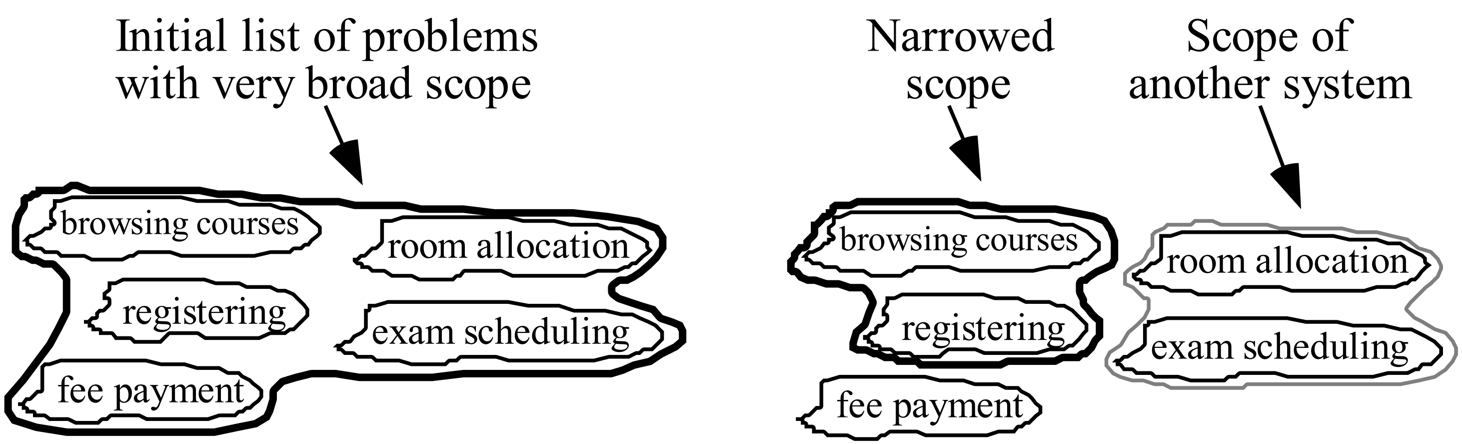
- A problem can be expressed as:
  - A *difficulty* the users or customers are facing,
  - Or as an *opportunity* that will result in some benefit such as improved productivity or sales.

- 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



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

Some Techniques for Gathering and Analysing Requirements

- Observation
- Interviewing
- Brainstorming
- Prototyping

Gathering and Analysing Requirements

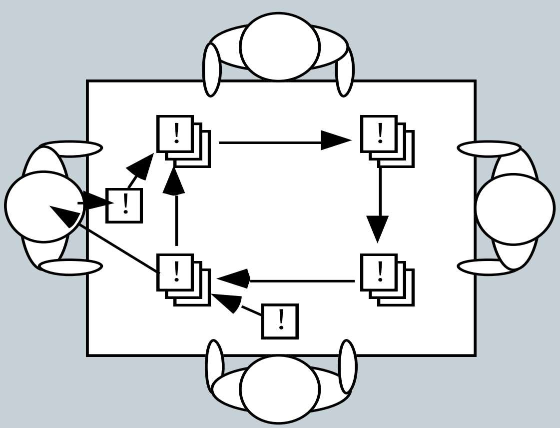
- Observation
  - May help to find details, that user may miss to tell
  - Shadowing important potential users as they do their work
    - ask the user to explain everything he or she is doing
  - Session videotaping
  - Consumes time, best for large projects

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

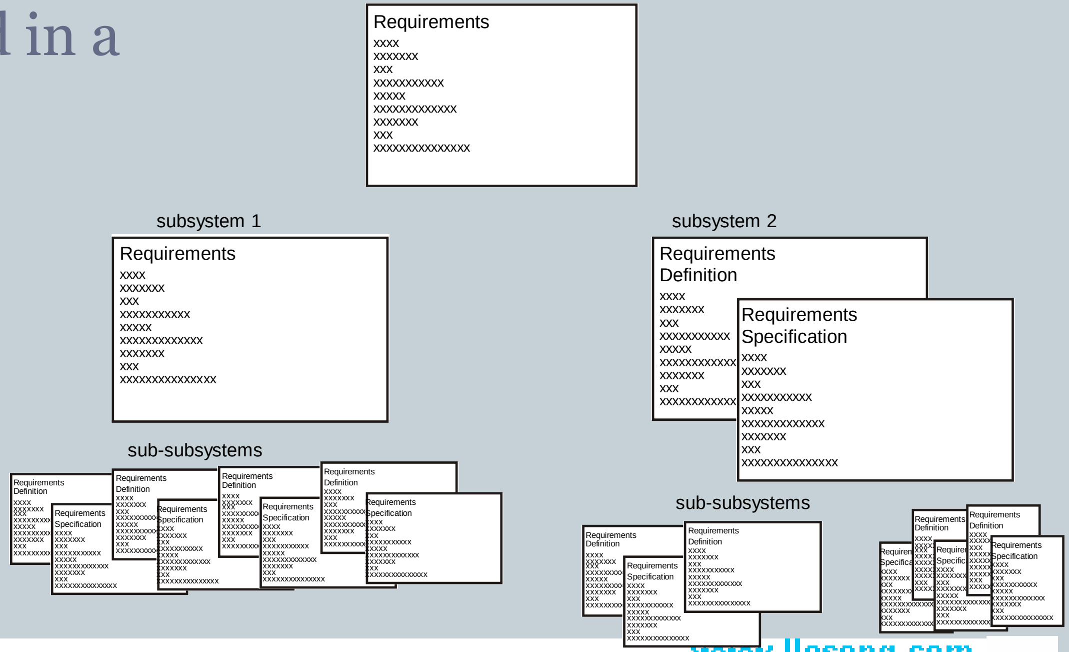
Gathering and Analysing Requirements…

- 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

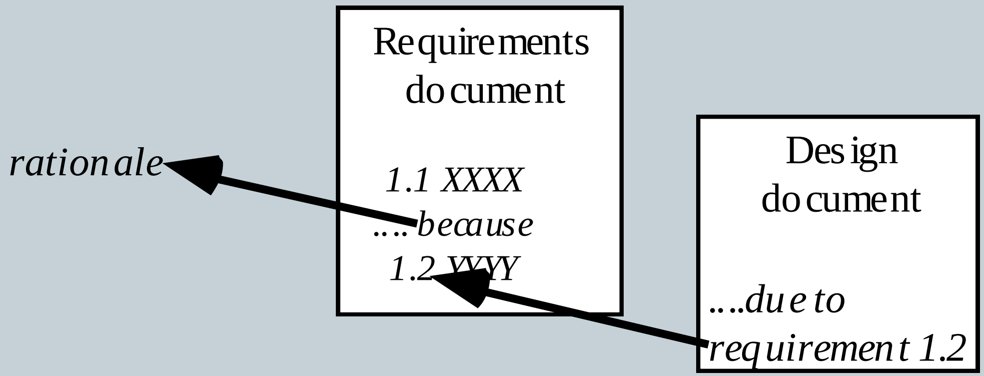
Reviewing Requirements

- **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*

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

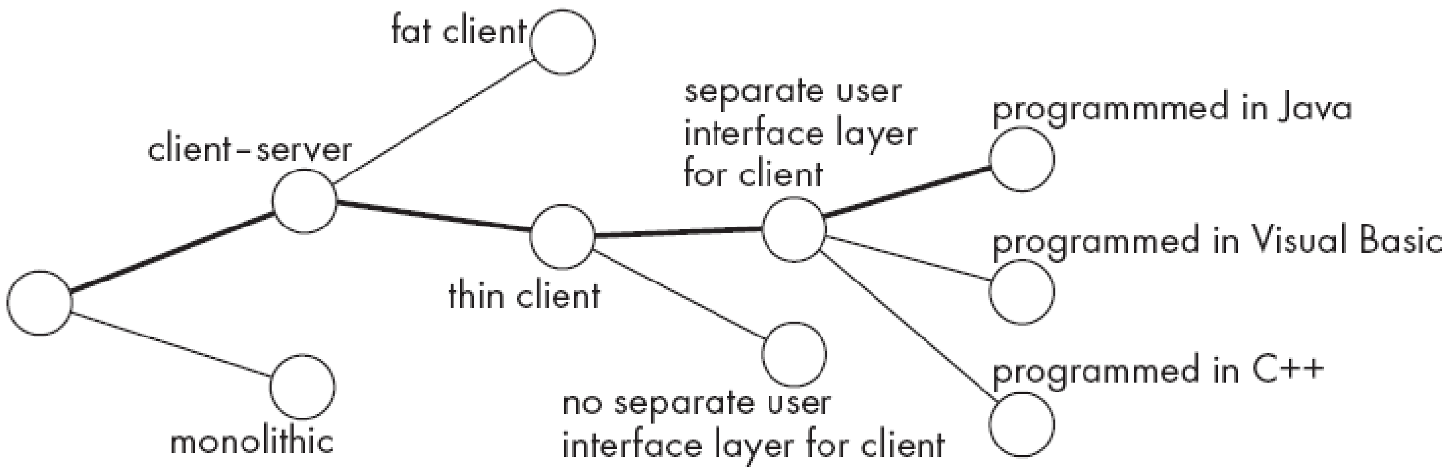
- A designer is faced with a series of *design issues*
  - These are sub-problems of the overall design problem.
  - Each issue normally has several alternative solutions:
    - design *options*.
  - The designer makes a *design decision* to resolve each issue.
    - This process involves choosing the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



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

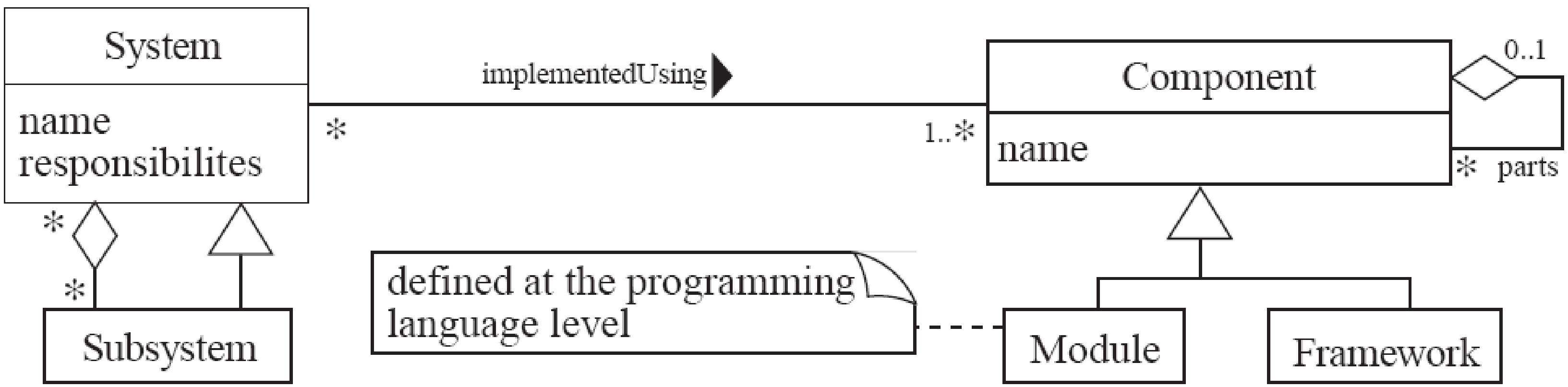
- A component that is defined at the programming language level
  - For example, methods, classes and packages are modules in Java.

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

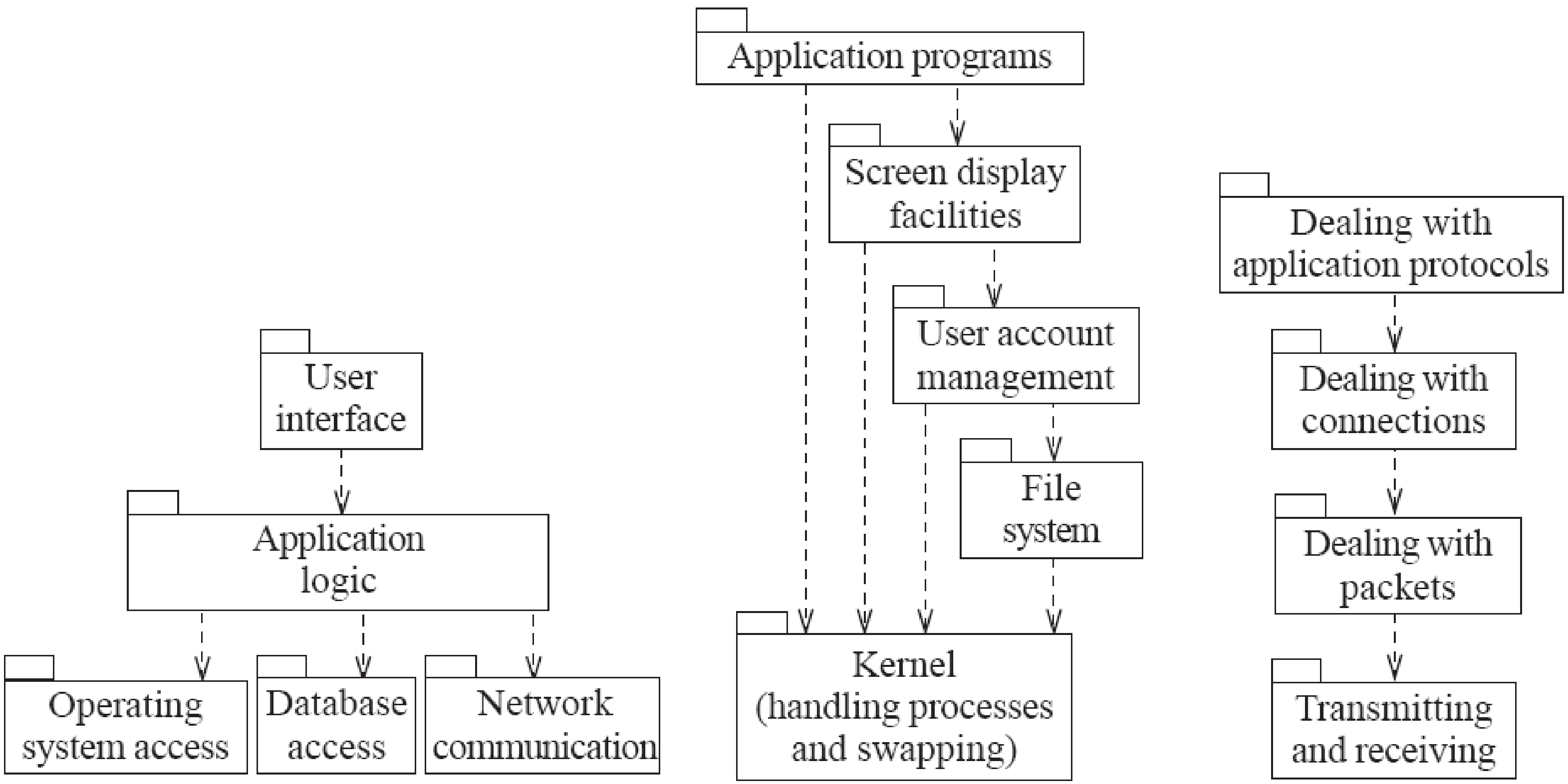
Functional cohesion

- 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

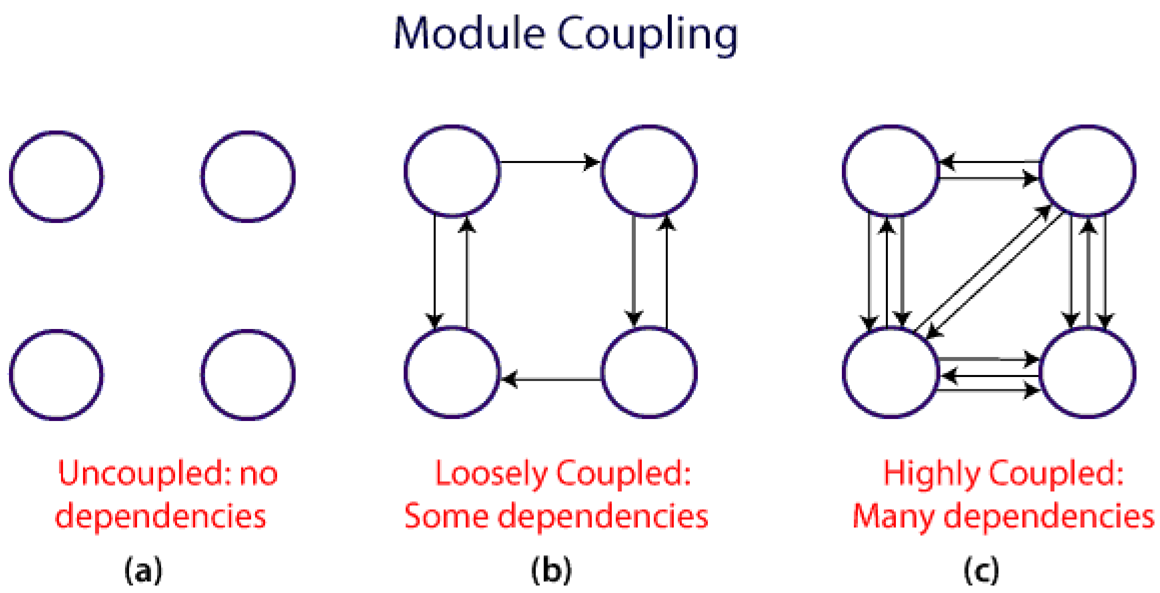
Utility cohesion

- 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



Content coupling:

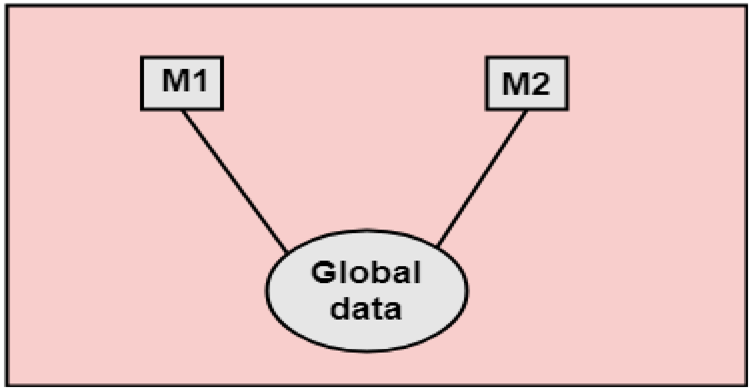
- Occurs when one component *surreptitiously* modifies data that is *internal* to another component .
  - To reduce content coupling you should therefore *encapsulate* all instance variables
    - declare them `private`
    - and provide get and set methods
  - This is the *worst form of coupling* and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

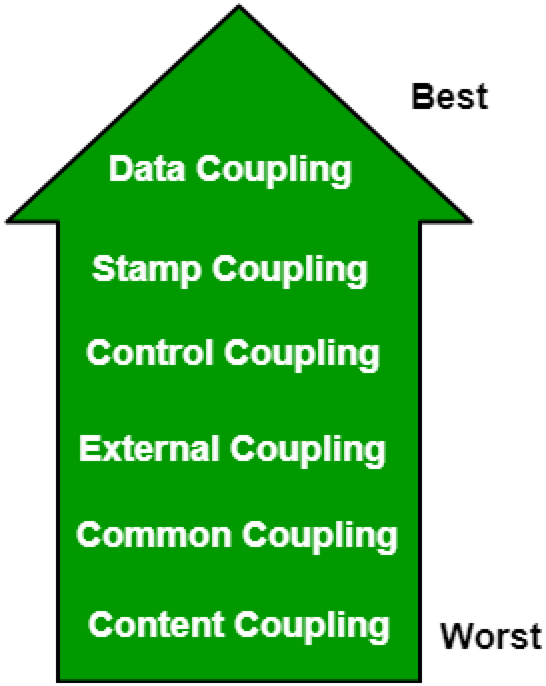
Inclusion or import coupling

- 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