

COMP647

# Software Design Methodologies

Lecture 1

Review of Software Engineering

## **Software Life Cycle**

### **Feasibility study WHY?**

- cost-benefit analysis
- Is it worthwhile doing the project?

### **Requirements analysis and specification WHAT?**

- What should the software do?
- product: specification and requirements document

### **Design and specification HOW?**

- How should the software do it?
- architectural design
  - overall structure and organization of modules
  - product: architectural design, module interface specification
- detailed design
  - choice of data structures and algorithms for module internals
  - product: internal module design

### **Coding and module testing REALIZE components!**

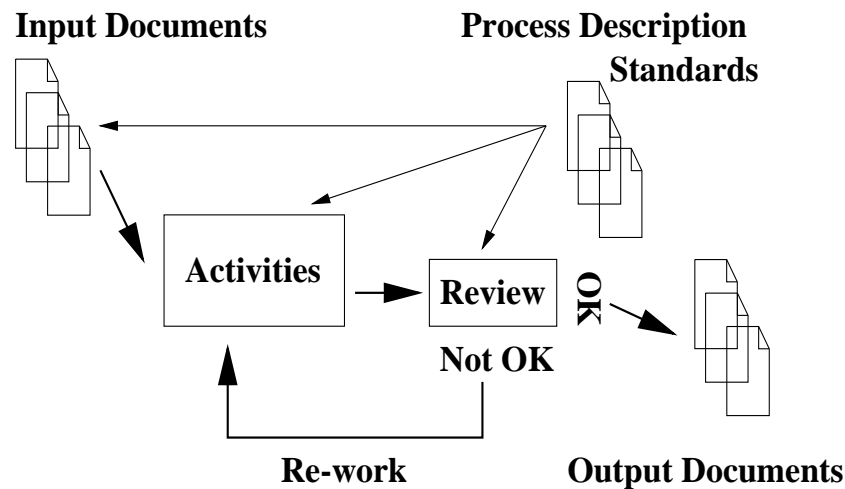
- code and test individual modules
- product: software, test result description

### **Integration and system testing REALIZE system!**

- test whether several modules work together
- test system as a whole
- product: test result description

### **Delivery and maintenance EVOLVE!**

## A Lifecycle Phase



For each phase of the life-cycle you should note:

- **PRODUCTS/DELIVERABLES**
  - what is the aim of the phase that is, what does it aim to produce
- **ACTIVITIES**
  - what steps are taken in the process of producing the deliverables
- **AUDIENCE** for each deliverable
  - who will use each deliverable
  - how will they use each deliverable
  - what information will they extract from each deliverable
  - is it easy for them to find/understand this information

NB there are often many different users of each deliverable

- **HOW TO REVIEW QUALITY** of each deliverable
  - what steps can you take to ensure the quality of each deliverable
    - \* during the production of the deliverable
    - \* after the production of the deliverable

## **Context of Design**

Req. Analysis — UNDERSTAND THE PROBLEM

Design — PROPOSE, REFINE, DETAIL  
A MECHANISM  
THAT SOLVES THE PROBLEM

Implementation — REALIZE THE MECHANISM

## Definitions for Software Engineering

**Product** — what we are trying to build.

**Process** — the methods we use to build the product.

**Method** — a guideline that describes an activity.  
Methods are general, abstract, widely applicable.  
Example: top-down design.

**Technique** — a precise rule that defines an activity.  
Techniques are precise, particular, and limited.  
Example: loop termination proof.

**Tool** — a mechanical/automated aid to assist in the application of a methodology.  
Examples: editor, compiler, . . .

**Methodology** — a collection of techniques and tools.

**Rigor** — careful and precise reasoning.  
Use *rigor* as much as possible.

## **Principles of Software Engineering**

**People are Human**

**Separation of Concerns**

**Modularity**

**Incrementality**

**Abstraction**

**Generality**

**Anticipation of Change**

**Rigor and Formality**

## **Main Obstacles to Software Engineering**

Re-work

Complexity

Change

## Software Process Models

A **process model** is a description of a way of developing software.

A process model may also be a methodology for software development.

- code-and-fix (obsolete — only for small scale)
- waterfall — documentation driven
- evolutionary — increment driven
- transformational — specification driven
- spiral — risk driven

A process is divided into **phases** or stages

Each phase has

- *activities* to be performed, including quality reviews, collecting cost and time data, collecting defect data from quality reviews
- *input* products, including process description, and standards
- *output* products (internal and external) to be produced  
(NB the audience for each document)
- *entry condition*  
under what circumstances can phase begin  
i.e. do you have enough reliable input to begin
- *exit condition*  
under what circumstances is this phase finished  
i.e. do deliverables pass quality control

## Comparison on Models

- Waterfall development provides: good management of the process; poor response to clients; a large final product; a short test phase.
- Spiral development provides: short development time; good response to changes in requirements; a small final product.
- Consensus: the spiral is better than the waterfall, especially for products that are not well understood.
- “In the old days, we **wrote** software; then, for a while, we **built** software; nowadays, we **grow** software” (Brooks, 1978)

OO methodologies adopt a spiral process model based on either evolutionary prototypes (for novel products) or incremental prototypes (for well-understood products)

# Overview of Design

Design process

- Design phases

- Deliverables

- Activities

## DESIGN FOR CHANGE

- The nature of change

- Information hiding, Modularization

How to review designs

*Design* produces a solution meeting the functional and non-functional constraints of the requirements.

This is called “*fitness for purpose*”

The solution describes **how** to do the task.

A “good” design should provide

- fitness for purpose (correct, reliable, robust)
- maintainable (design for change)
- “positive” qualities important to user

## **Design Phases**

**Architectural Design** looks at structural issues

- organization into subsystems and modules
- assignment of functionality to components
- distribution of control
- protocols for communication, synchronization, and data access
- physical allocation of components to processors

Deliverables: *Architectural Design (AD)*;  
*Module Interface Specifications (MIS)*

**Detailed Design** provides internal details of each module in the design, including

- each routine of interface,
- parameters for each routine,
- format of any input/output,
- the data structures and algorithms used,

Deliverables: *Internal Module Description (IMD)*

## Terminology

*Subsystem* is subset of the modules making up a system.

Can think of a subsystem as a high-level module, so a subsystem is a provider of services

*Module* is a provider of computational resources or services

*Unit* is an individual routine, procedure, function

*USES relation*: M1 uses M2 if, in order for M1 to provide its services, M1 uses the services of M2

*IS\_COMPONENT\_OF relation*: M1 *is\_component\_of* M2 if M1 is physically part of M2

also called *is\_part\_of* relation

inverse of *aggregation* relation

## Design Deliverables

*Architectural Design* (AD) provides

- a **module diagram**,
- a brief description of the **role** of each module, and
- **traceability** information between requirements and the module functionality.

*Module Interface Specifications* (MIS) describes each service provided by each module.

To specify a function, give:

- name;
- argument types;
- a **requires clause** — a condition that must be true on entry to the function (*pre-condition*);
- an **ensures clause** — a condition that will be true on exit from the function (*post-condition*);
- further comments as necessary.

The requires and ensures clause constitute a *contract* between the user and implementor of the function.

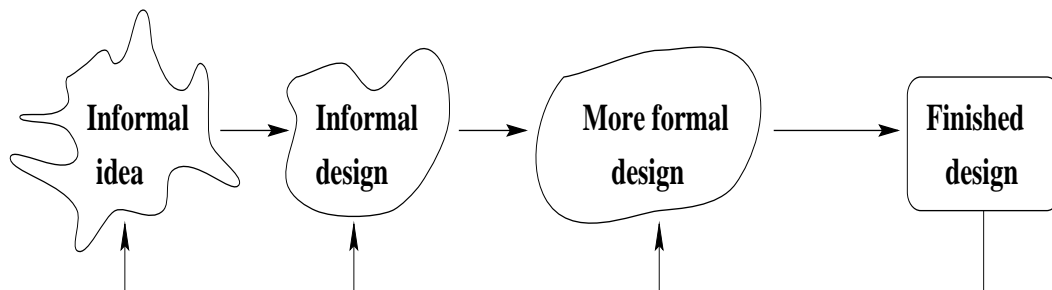
*Internal Module Description* (IMD) has the same structure as the MIS, but adds:

- data descriptions (e.g. binary search tree);
- data declarations (types and names);
- a description of how each function will work (pseudocode, algorithm, narrative, . . .).

## Design Activities

**Basics:** *iterate* the following steps

1. propose a tentative design
2. describe as rigorously as necessary
3. trace scenarios to find pitfalls in proposal
4. propose solutions to pitfalls, discuss pros and cons



*Brainstorming* is team discussion to toss around ideas  
— free unconstrained flow of ideas  
— no criticism of ideas allowed  
— collect as many ideas as possible

Follow-up meeting examines, criticises, and selects ideas

*Issue-driven design* carefully documents **rationale** for design by examining *issues*

For each issue/problem with proposed design:

- describe issue fully
- list alternative solutions
- argue pros and cons of each solution
- propose new solution combining best ideas from all solutions offered
- iterate
- decide on solution to the issue/problem

Sometimes issue “remains on the table”  
while further investigation is done, eg prototypes

## Design Activities Overview

decompose system into subsystems and modules  
“execute” scenarios of software use cases, to clarify

- which module does what
- which modules are needed
- the use they make of each other
- their raison d’etre
- their responsibilities

**iterate** until AD document is stable

develop a tentative design of module interfaces

“execute” scenarios of software use cases, to clarify

- what is the sequence of calls to the interface
- what information is passed (in either direction)
- are interfaces complete?  
ie each task can be performed
- are interfaces elegant?  
ie can each task be performed easily

**iterate** until MIS has a complete elegant interface for each module

(may have to change AD, redistribute responsibilities)

choose the module internal design to meet design trade-offs using knowledge of data structure and algorithm trade-offs

(may impact MIS, require more info passed in interfaces)

May do all activities at once

— work at different levels of abstraction

## Design for Change

A **maintainable** system should be

**understandable** A design must be understood before it can be changed.

Keep things as simple as possible.

Document well. Document top-down. Document rationale.

**modular** Want a loosely coupled system of cohesive modules.

Ideal: A (anticipated) change to a module should not impact other modules at all.

Cohesion is a property of modules.

A *cohesive* module provides a small number of closely related services.

Coupling is a property of systems.

Modules are *loosely coupled* if the communication between them is simple.

**traceable** Design changes may come from changes in requirements, or from defect reports of code.

It must be easy to find all parts of the design that correspond to a requirement, or to a piece of code.

## The Nature of Change

What can change? Everything and Anything!

**change of algorithms** — best understood type of change  
eg, sort algorithm

**change of data representation** — about 17  
eg, linked list to hash table  
Use abstract data types!

**change of underlying abstract machine**  
eg machine defined by system interface to OS, DBMS, Windows  
Use layers!

**change of peripheral devices** — especially for embedded systems  
eg, device drivers for terminals, disks, LAN  
eg, image recognition input devices  
eg, sensor input to process control system

**change of social environment**  
eg, changes in tax legislation, accounting obligations  
eg, changes in interest rates  
eg, changes in “business rules”

## Information Hiding

Information hiding is the main strategy for design for change.

**Encapsulate** a module by only allowing access to module services via the *interface*.

The *implementation* details are **hidden**.

No other module depends on these details, so they can change without impacting other modules.

What should be visible in interface?

— as little as possible

What should be hidden?

— as much as possible, especially things likely to change

Module secrets

— data representation, eg symbol table module

— details of abstract machine

    eg, interface to X windows, or to file system

    eg, details of input formats, syntaxes

— etc

## **A Recipe for Module Design**

1. Decide on a secret.
2. Review implementation strategies to ensure feasibility.
3. Design the interface.
4. Review the interface.  
Is it too simple or too complex? Is it cohesive?
5. Plan the implementation.  
E.g. choose representations.
6. Review the module.
  - Is it self-contained?
  - Does it use many other modules?
  - Can it accommodate likely changes?
  - Is it too large (consider splitting) or too small (consider merging)?

## How to Review Designs

Aim: to discover errors, not fix them

### Design Review

- panel members study the design document(s) (or sections)
- mark items on checklist that seem incorrect or need clarification
- panel meets with designers and discuss marked items

### Design Walkthrough

- designer explains logic of the design step by step to a panel of peers
- panel asks questions, point out errors, seek clarification

more informal than review

much benefit for designer in the process of articulating and explaining design

Automated cross checking

eg compiler type-checking procedure calls

Metrics

- number of modules
- fan-in, fan-out
- number of variables, routines, and parameters in interface