COMP201 Software Engineering I
Lecture 14 – Software Design

Lecturer: T Carroll

Email: Thomas.Carroll2@Liverpool.ac.uk

Office: G.14

See Vital for all notes

Software Design Is…

Deriving a solution which satisfies software requirements

The Design Problem

- You have 4 developers
- You have a 500 page requirements specification
- How to control break up the work?

Software Design

- Why?
  - Good software should be:
    - Simple
    - Understandable
    - Re-usable
    - Flexible
    - Portable
- How?
  - Complex to simple
  - Abstraction

Software Design in Reality

- Design mixed with implementation
- Design step 1
  - Implement and add more design
- Design step 2
  - Implement and add more design

- In effect much of software is designed while coded and the design document doesn’t reflect the final product

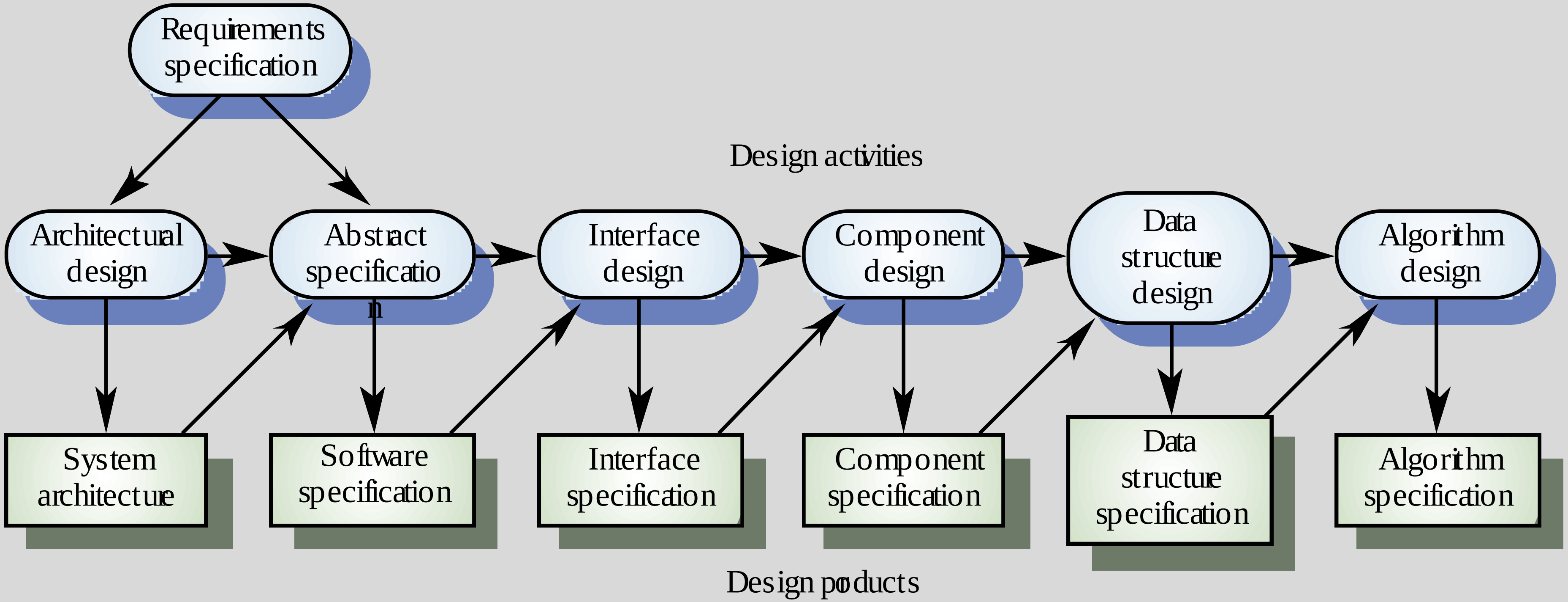
Stages of Design

- Problem understanding
  - Look at the problem from different angles to discover the design requirements.
- Identify one or more solutions
  - Evaluate possible solutions and choose the most appropriate depending on the designer's experience and available resources.
- Describe solution abstractions
  - Use graphical, formal or other descriptive notations to describe the components of the design.
- Repeat process for each identified abstraction
  - until the design is expressed in primitive terms.

The Design Process

- **Any design may be modeled as a directed graph** made up of entities with attributes which participate in relationships.
- The system should be described at several different levels of abstraction.
- **Design takes place in overlapping stages.** It is artificial to separate it into distinct phases but some separation is usually necessary.

Phases in the Design Process



Design Phases

- *Architectural design:*
  - Identify sub-systems.
- *Abstract specification:*
  - Specify sub-systems.
- *Interface design:*
  - Describe sub-system interfaces.
- *Component design:*
  - Decompose sub-systems into components.
- *Data structure design:*
  - Design data structures to hold problem data.
- *Algorithm design:*
  - Design algorithms for problem functions.

Modularity

Design

- Computer systems are not monolithic
  - They are usually composed of multiple, interacting modules.
- **Modularity** has long been seen as a key to cheap, high quality software.
- The goal of system design is to decode:
  - What the modules are;
  - What the modules should do;
  - How the modules interact with one-another

Modular Programming

- In the early days, **modular programming** was taken to mean constructing programs out of small pieces: “subroutines”
- But **modularity** cannot bring benefits unless the modules are
  - autonomous,
  - coherent and
  - robust

Procedural Abstraction

- The most obvious design methods involve **functional decomposition**.
- This leads to programs in which **procedures represent distinct logical functions** in a program.
- Examples of such functions:
  - “Display menu”
  - “Get user option”
- This is called **procedural abstraction**

Programs as Functions

- Another view is programs as functions:

$$x \longrightarrow f \longrightarrow f(x)$$

- The program can be viewed as functions of valid inputs to outputs
- There are programming languages that directly support this view of programming
  - ML
  - Miranda
  - LISP
- Well suited to certain application domains (eg: compilers)
- Not well suited to distributed, non terminating system (eg: Process control systems, Operating Systems, ATMs)

Object-Oriented Design

- The system is viewed as a collection of interacting objects.
- The system state is decentralized and each object manages its own state.
  - Note: use of internal state against functional programming
- Objects may be instances of an object class and communicate by exchanging messages.

Five Criteria for Design Methods

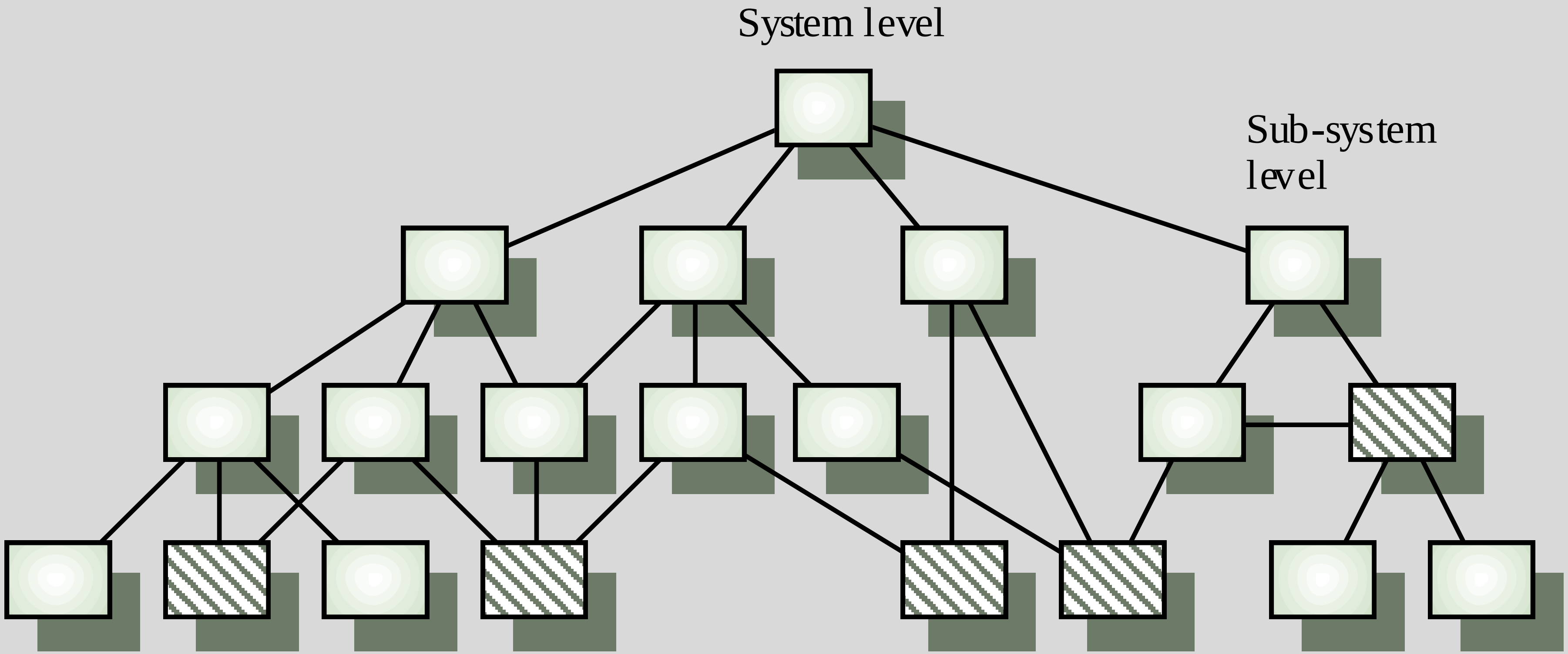
- We can identify **five criteria** to help evaluate modular design methods:
  - Modular decomposability;
  - Modular composability;
  - Modular understandability;
  - Modular continuity;
  - Modular protection.

Modular Decomposability

Modular Decomposability

- **Modular decomposability** - this criterion is met by a design method if the method supports the decomposition of a problem into smaller sub-problems, which can be solved independently.
- **In general, this method will be repetitive**: sub-problems will be divided still further
- **Top-down design methods** fulfil this criterion; stepwise refinement is an example of such a method

Hierarchical Design Structure



Top-Down Design

- In principle:
  - top-down design involves starting at the uppermost components in the hierarchy and working down the hierarchy level by level.
- In practice:
  - large systems design is never truly top-down. Some branches are designed before others. Designers reuse experience (and sometimes components) during the design process.

An Example of Top-Down Design

- Imagine designing a word processor program from scratch.
- What subsystems could be initially found at the top level?
  - File I/O
  - Printing
  - GUI
  - Text processing
- Each subsystem can decompose further
  - File I/O
    - saving documents
    - opening documents

Modular Composability

Modular Composability

- **Modular composability** - a method satisfies this criterion if it leads to the production of modules that may be freely combined to *produce new systems*.
- Composability is directly related to the issue of **reusability**
- Composability is often at odds with decomposability
    - Top-Down design tends to produce modules that may not be composed in the way desired
    - leads to modules which fulfil a specific function, rather than a general one

Examples

- The Numerical Algorithm Group (NAG) libraries contain a wide range of routines for solving problems in linear algebra, differential equations, etc.
- The Unix shell provides a facility called a **pipe**, written “|”, whereby
  - the standard output of one program may be redirected to the standard input of another; this convention favours composability.

Modular Understandability

Modular Understandability

- Modular Understandability - a design method satisfies this criterion if it encourages the development of modules which are easily understandable.
- Abstraction
  - int A=21
  - int age_in_years = 21;   print_out_document(Document d)
- COUNTER EXAMPLE 1. Take a thousand lines program, containing no procedures; it's just a long list of sequential statements. Divide it into twenty blocks, each fifty statements long; make each block a method.
- COUNTER EXAMPLE 2. "Go to" statements.

Modular Understandability

- Related to several component characteristics
  - Can the component be understood on its own?
  - Are meaningful names used?
  - Is the design well-documented?
  - Are complex algorithms used?
- Informally, high complexity means many relationships between different parts of the design.

Modular Continuity

Modular Continuity

- **Modular continuity** - a method satisfies this criterion if it leads to the production of software such that a small change in the problem specification leads to a change in just one (or a small number of) modules.
- EXAMPLE. Some projects enforce the rule that no numerical or textual literal should be used in programs: only symbolic constants should be used
- COUNTER EXAMPLE. Static arrays (as opposed to dynamic arrays) make this criterion harder to satisfy.

Modular Protection

Modular Protection

- Modular Protection - a method satisfied this criterion if it yields architectures in which the effect of an abnormal condition at run-time only effects one (or very few) modules
- EXAMPLE. Validating input at source prevents errors from propagating throughout the program
- COUNTER EXAMPLE. Using **int** types where **short** types are appropriate.

A Real Life Example – Ariane

- The failure of an Ariane 5 space launcher is possibly the most expensive software bug in history at around $370 million.
- Other bugs have been even worse by causing loss of life, for example Therac-25 radiation machines.

The Ariane 5 Space Launcher

- While developing the Ariane 5 space launcher, the designers reused a component (the inertial reference software) which was successfully used in the Ariane 4 launcher
- This component failed 37 seconds into the flight and the ground crew had to instruct the launcher to self-destruct.
- The error was caused by an unhandled numerical conversion exception causing a numeric overflow.
- **Component reuse** is usually a good thing but care must be taken that assumptions made when the component was developed are still valid!

Recap

Recap

- Software design derives a solution to satisfy requirements
- Characteristics of good software
  - Simple
  - Understandable
  - Portable
  - Flexible
  - Re-usable
- Modular Design
- Characteristics of good modular design