B16 Software Engineering

Structured Programming

Lecture 1

Dr Andrea Vedaldi
4 lectures, Michaelmas Term

For lecture notes, tutorial sheets, and updates see
http://www.vlfeat.org/~vedaldi/teach.html

In this first installment
B16 Software Engineering: Structured Programming

• Software engineering principles
  – Mostly about concepts/principles of design, modularity, abstraction, encapsulation, etc.

• Structured programming
  – Revision, coding in C and MATLAB
  – Control flow, variables, data types
    ▪ How variables are implemented in hardware
  – Functions
    ▪ Code re-use, parameters, libraries
  – Data structures
    ▪ Structures/classes, arrays

• Algorithms

• Recursion

B16 Software Engineering

Meta Learning Outcomes

• Understand the importance of good design practice in software, and the role that structured and object oriented programming ideas play in this.

• Understand how to develop applications that can interact with the outside world (i.e. programs as part of bigger engineering systems) via the operating system and via inter-computer and computer-device communications

• Understand how ideas in 4th year options and projects can actually be implemented in software.
Learning Outcomes

The course will aim to give a good understanding of basic software design methods, and emphasize the need to produce well-structured maintainable computer software. The course will concentrate on principles, but these will be reinforced with examples in MATLAB and C/C++ programming languages. Specifically, by the end of this course students should:

- Understand concepts of basic program design techniques that can be applied to a variety of programming languages.
- Understand the need for structured programming in software projects.
- Understand the mechanics of function calls and of recursion.
- Understand the role, uses and advantages of compound data structures.
- Be able to recognise, produce and/or maintain well structured programs.

Lecture 1 outline

- Importance and challenges of software engineering
  - Examples

- Software engineering
  - What it shares and how it differs from other engineering
  - Abstraction and modularity

- Software processes & their models
  - Specification, design & implementation, validation, evolution
  - Waterfall and incremental models

- Structured programming
  - Key ingredients
  - Language support

- Design tools
  - Flow charts, pseudo-code, data flow diagrams, state diagrams

The Role of Computing in Engineering

Computing is ubiquitous in engineering. Why?

- Awesome speed of modern, everyday computers makes complicated analysis and simulation possible across all domains.
- Applications in design and modelling. Far beyond the reach of the mortal human engineer. Indeed many modelling problems are utterly infeasible without modern computers and software.
- In embedded systems, computers can provide a level of power, speed, flexibility and control not otherwise possible (e.g., mobile phone).
- Computing is “cheap” (but exercise this argument with care).
- Software is the key!

Some examples ...
Example: mobile gadgets

- Modern mobile phones run complex software
- A simple metric
  - Number of *source lines of code* (SLOC)
- Android SLOC: 12 M
  - Or about 170 reams of paper

Example: Sizewell B

- Nuclear power station (PWR), on-stream in 1995
- Software used extensively in the **design**
- Software used for **control**

Example: A380

- 1400 separate programs
- There is a software project just to manage all the software!
- Clearly safety-critical features of the software

Example: NPfIT

- NHS National Plan for IT
- **Goal** provide electronic care records for patients
  - Connect 30000 GPs and 300 hospitals
  - Provide secure access to records for healthcare professionals
  - Provide access for patients to their own records via “Healthspace”
Intrinsic difficulties with software

- **Analog vs discrete**
  - Analysis of analog systems can often be interpolated (e.g., if a bridge stands a certain load, it will stand smaller ones)
  - Not so for software

- **The “curse” of flexibility**
  - Can encourage unnecessary complexity
  - Redefinition of tasks late in development – shifting goal-post

- **Complexity and invisible interfaces**
  - Standard way of dealing with complexity is via modularity
  - But this alone is not enough because interfaces can be subtle and invisible, and here too there is a need to control complexity

- **Historical usage information**
  - Unlike physical systems, there is a limited amount of experience about standard designs

When software projects go wrong

- **London Ambulance Service**
  - 1992, computerised ambulance despatch system fails

- **Therac-25**
  - 2 people died and several others exposed to dangerous levels of radiation because of software flaws in radiotherapy device

- **OSIRIS**
  - £5M University financial package
  - Expenditure to date more like £20-25M

- **NPfIT?**
  - NHS £12 billion IT project

- **comp.risks** is a great source of others...

NHS National programme for IT: NPfIT

- Plan to provide electronic care records for patients
  - Connect 30000 GPs and 300 hospitals
  - Provide secure access to records for healthcare professionals
  - Provide access for patients to their own records via “Healthspace”

- **Laudable?**

- **Realistic?**
  - Software Engineering specialists have their doubts
  - Ross Anderson (Prof of Security Engineering, Cambridge Computing Laboratory) writes in his blog “I fear the whole project will just continue on its slow slide towards becoming the biggest IT disaster ever”.
Software Engineering

Software Engineering is more than just programming/coding.

- It is an engineering discipline that is concerned with all aspects of software production.

- It seeks principles and methodology that yield programs that are:
  - *usable* (meet their requirements, are acceptable by the users)
  - *dependable* (reliable, secure, safe)
  - *maintainable* (can be updated with minimal effort)
  - *efficient*

- It covers:
  - theory of computation
  - programming tools (e.g. languages)
  - design tools (e.g. software architectures)
  - processes of software creation & management
  - ...

Software vs “Other” Engineering

- How is software engineering *similar* to other engineering?

  - **Modularity and Abstraction**
    - Modularity: decompose a system into components
    - Abstraction: separate a component behavior from its implementation

  - **Benefits** of modularity and abstraction
    - better understanding
    - knowledge reuse
    - isolate local changes

  - **Examples**
    - Thevenin / Norton equivalent of a linear circuit
    - Transistors, flip-flops, registers, execution pipelines, chips, computers

Abstraction and modularity

- How is software engineering *different* from other engineering?

  - **Software products**
    - can incorporate massive complexity
    - can be easily modified
    - are weightless
    - do not age, corrode, etc.
    - do not have *manufacturing* defects

  - **Cost** of software
    - cheap to manufacture (transfer a file)
    - expensive to produce (design, implement, validate, run, maintain, evolve)

  - Note, computer *hardware* engineering ≠ *software* engineering
    - however, software must account for hardware limitations
Software processes

- A software process is a set of related activities that leads to the production of a software product [Sommerville].

- Fundamental activities (lifecycle)
  1. Specification
  2. Design & implementation
  3. Validation
  4. Evolution

Specification

The process of understanding and defining what services are required from the system and identifying the constraints on the system’s operation and development [Sommerville].

- Requirements must be elicited, specified, and validated
  - Identify services and constraints by talking to the user
  - Refine these iteratively
  - Produce a requirement specification (document)

- Example
  - User requirement: “calculate the n-bounce trajectory of a lossy bouncing ball”
  - Must be refined:
    - How will the output be represented?
    - How will the initial condition be specified?
    - Which approximations are tolerated in the numerical integration?
    - Should the air resistance be accounted for?
    - ...

Design & implementation

- Design
  - architectural design (define components)
  - interface design (define how components inter-operate)
  - component design (define how components work)
  - database design
  - user interface design
  - ...

- Implementation
  - programming each component
  - unit testing & debugging
  - note, some fine-grained design decisions are left to the programmer

Top-down design

- Here want to keep in mind the general principles
  - Abstraction
  - Modularity

- Architectural design
  - identifying the building blocks (modules)
  - describe the data, functions, and their constraints

- Interfaces
  - define how the modules fit together

- Component design
  - recursively design each block
**Validation**

- **Validation ⊃ Verification**
  - Verification: does the software conform to its specifications?
  - Validation: does the software actually do what it was supposed to do?

- **Designing tests** for verification
  - *Black-box* (from software specification) vs *white-box* (from code inspection)
  - *Top-down* (system testing) vs *bottom-up* (unit testing)

- **Coverage of the tests**
  - Exhaustive testing is impossible
  - Pick representative examples of “normal” inputs as well as “corner cases”

- **Example:** Test a function to compute the tangent
  - normal input: $\tan(1.1)$
  - corner cases: $\tan(-\pi/2)$, $\tan(0)$, $\tan(\pi/2)$

---

**Evolution**

- Software evolution may be triggered by changing business requirements, by reports of software defects, or by changes to other systems in a software system’s environment [Sommerville].

- Example goals of software evolution:
  - fix a bug, add a functionality, improve efficiency

- Evolving software is costly
  - require a good understanding of the system, which often must be re-acquired
  - effects of changes are difficult to predict and risky

- **Reengineering** may be needed to accommodate changes
  - *refactoring*: rewrite part of it with modern tools, a better design, etc
  - rewrite the documentation
  - sometimes rewrite from scratch (but keep the functionality equivalent)

---

**Software process models**

- A **software process model** is a manner of organising the fundamental activities
  1. Specification
  2. Design & implementation
  3. Validation
  4. Evolution

- Two competing models
  - **waterfall model**
  - incremental models
    - e.g. extreme programming

---

**Waterfall model**

- Ideally, activities are distinct stages to be signed off chronologically
- In practice, activities partially overlap and interact

**Fundamental activities**

1. Specification
2. Design & implementation
3. Validation
4. Evolution
**Incremental models**

Develop the software by increments, exposing them to the user and elicit changes, and go back to incorporate them.

- **Extreme programming**
  - interleave frequent software releases with user validation
  - development is *test based*
    - tests encode the specification
    - tests are written before the program
    - all releases must pass all tests
  - do not try to predict changes; changes reactively (refactoring)

- Extreme programming vs waterfall
  - much more lightweight
  - usually preferable for small-medium size projects

**Structured Programming**

- **Modules**
  - Group related functionalities together
  - Clearly separate behavior from implementation
    - encapsulation, information hiding, ...

- **Control flow**
  - *for*, *repeat*, *while*
    - *Go To Statement Considered Harmful* [Dijkstra]
  - procedures (or functions)
    - e.g., `sin()`, `addAppointmentToCalendar()`

- **Data**
  - compound type (e.g. `struct``
  - objects

  - Most computer languages have features that help structured programming

  - However, it is fundamentally a matter of *how you organise your code*

**Modularity is recursive**

Components are recursively decomposed into simpler components, until machine instructions are generated. There are two ways of constructing such a hierarchy:

- **Top-down**
  - Code high level parts using “stubs” with assumed functionality for low-level dependencies
  - Iteratively descend to lower-level modules

- **Bottom-up**
  - Code and test each low-level component
  - Need “test harness” so that low-level can be tested in its correct context
  - Integrate components

  - Not hard-fast rules; combination often best

**Modularity: Algorithm and Data Structures**

- Different languages encourage decomposing programs in different ways
- Procedural languages: decompose algorithms
- Object-oriented languages: decompose data

[Wirth 75]
### Flow chart

- **Start**
- `count ← 0`
- `Wait for alarm`
- `count ← count + 1`
- `Hit snoozer`
- `ready or count ≥ 3` (false)
- `Climb out of bed`
- **Stop**

### Pseudo-code

```plaintext
count = 0

Wait for alarm
Hit snoozer
count = count + 1

while (not ready() and count <= 3)
  Climb out of bed
```

### State Diagram

- **Full Power**
  - **Set Time**
    - **Enable**
      - **Do: Display 'Ready'**
    - **Operation**
      - **Do: Operate Oven**
    - **Door Open**
    - **Door Closed**
    - **Cached**
  - **Waiting**
    - **Do: Display Time**
    - **Half Power**
      - **Set Time**
        - **Enable**
          - **Do: Display 'Ready'**
        - **Operation**
          - **Do: Operate Oven**
        - **Door Open**
        - **Door Closed**
        - **Cached**
    - **Disabled**
      - **Do: Display Waiting**

- **Controller**
  - **trust**
  - **Simulator**
    - **state**
    - **Display**

- **VTOL (Vertical Take-Off and Land)**
  - **velocity**
  - **gravity**
  - **trust**
  - **height**

---

[Sommerville]