B16 Software Engineering

Structured Programming

Lecture 1: Software engineering

Dr Andrea Vedaldi

4 lectures, Hilary Term

For lecture notes, tutorial sheets, and updates see http://www.robots.ox.ac.uk/~vedaldi/teach.html

B16 Part 1: Structured Programming

- **Software engineering principles**
  - Design, modularity, abstraction, encapsulation, etc.

- **Structured programming languages**
  - Interpreted (MATLAB) vs compiled (C) languages
  - **Control flow**
    - Sequencing, alternation, iteration
    - Functions and libraries
  - **Data**
    - Data types: primitive, aggregate, and compound
    - Local and global variables, parameters
    - The heap and the stack

- **Algorithms**
  - Proving algorithm **correctness** by mathematical induction
  - Time and space **complexity**
  - **Recursion**

1. **Structured Programming**
   - Algorithms, data structures, complexity, correctness
   - Writing programs using structure programming languages

2. **Object Oriented Programming**
   - Object-oriented programming
   - Writing programs using object-oriented languages

3. **Computer Communications and Networking**
   - How devices and computers communicate, the Internet
   - Writing programs that can exchange information on a network

4. **Operating Systems**
   - Hardware abstraction and virtualisation
   - Writing programs that access hardware features transparently and concurrently

---

**Texts**

- The C programming language, 2nd edition
  - Kernaghan & Ritchie

- Introduction to algorithms
  - Cormen, Leiserson, Rivest, Stein
Importance and challenges of software engineering
Examples & challenges
Software engineering
The aims and scope of software engineering
Abstraction and modularity
Software processes & their models
Specification, design & implementation, validation, evolution
Waterfall and incremental models
Structured programming
Imperative vs declarative languages
Overview of fundamental abstractions

A world-changing technology

The impact of computing

Control
Digital manufacturing
Digital components, from kitchen appliances to nuclear plants

Design
Numerical simulation
Computer-assisted design

Data processing and analysis
Extremely large datasets (big data)
Sciences, humanities, healthcare

Communication
Digital telephony
Computer networks
Internet, electronic commerce, digital economy

Entertainment
Computer graphics, music
Gaming
Digital arts

Software is the foundation of computing

Design & Control: Sizewell B
Nuclear power station (PWR), on-stream in 1995
Software used extensively in the design
Software used for control

Design & Control: A380
1400 separate programs
There is a software project just to manage all the software!
Safety-critical features (redundancy)
Design & Control: E- and S- Class 2013

- Extensive assisted driving support
- Emergency braking (pedestrians, front & lateral cars)
- Smart cruise control (follow lanes, maintain distances, spots holes/bumps adapting suspensions)
- Parks automatically

Design & Control: E- and S- Class 2013

- Pre-crash braking
- Body control
- Parking
- Adaptive cruise control
- Lane keeping
- Traffic signs
- Adaptive high beam
- Night view
- Attention

Oxford Robotcar

Big Data: NPfIT

- NHS National Plan for IT
- **Goal** provide electronic care records for patients
  - Connect 30,000 GPs and 300 hospitals
  - Provide secure access to records for healthcare professionals
  - Provide access for patients to their own records via “Healthspace”
The "size" of software

**SLOC**: number of source lines of code

```c
#include "fisher.h"
#include "gmm.h"
#include "mathop.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

static void
VL_XCAT(_vl_fisher_encode_, SFX)
  (TYPE * enc,
   TYPE const * means, vl_size dimension, vl_size numClusters,
   TYPE const * covariances,
   TYPE const * priors,
   TYPE const * data, vl_size numData,
   int flags)
{
  vl_size dim;
  vl_index i_cl, i_d;
  TYPE * posteriors ;
  TYPE * sqrtInvSigma;

  posteriors = vl_malloc(sizeof(TYPE) * numClusters * numData);
  sqrtInvSigma = vl_malloc(sizeof(TYPE) * dimension * numClusters);

  memset(enc, 0, sizeof(TYPE) * 2 * dimension * numClusters) ;
  for (i_cl = 0 ; i_cl < (signed)numClusters ; ++i_cl) {
    for(dim = 0; dim < dimension; dim++) {
      sqrtInvSigma[i_cl*dimension + dim] = sqrt(1.0 / covariances[i_cl*dimension + dim]);
    }
  }

  VL_XCAT(vl_get_gmm_data_posteriors_, SFX)(posteriors, numClusters, numData, 
                                          priors, 
                                          /* other parameters */);  
}
```
What 130K lines of code look like

Apollo-11, 1969

Margaret Hamilton, director of software engineering

Code available here: https://github.com/chrislgarry/Apollo-11/

Intrinsic difficulties with software

- Complexity
  - Handled via modularity

- Invisible interfaces
  - Modularity is tricky due to subtle and invisible interactions between software components (esp. concurrency)

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

- 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

- Historical usage information
  - Unlike physical systems, there is a limited amount of experience about standard designs
When software projects go wrong

- A320, Habsheim (1988) and Strasbourg (1992)

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 £12B IT project
- “I fear the whole project will just continue on its slow slide towards becoming the biggest IT disaster ever” (Prof. Ross Anderson, Cambridge)

More
- https://catless.ncl.ac.uk/Risks/ is a great source of others...

Importance and challenges of software engineering
- Examples & challenges

Software engineering
- The aims and scope of software engineering
- Abstraction and modularity

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

Structured programming
- Imperative vs declarative languages
- Overview of fundamental abstractions

Software engineering seeks principles and methodologies to make programs that are:
- Usable
  Meet their requirements, including being acceptable by the users
- Dependable
  Reliable, secure, safe
- Maintainable
  Can be updated with minimal effort
- Efficient
  Run as fast as possible with limited resources

Software engineering is concerned with all aspects of software production. It includes:
- Theory
  Computability, algorithms, correctness, complexity, formal languages
- Tools and best practices
  Specific programming languages, programming environments, developer tools to build, debug, and analyse programs
- Management
  Processes of software creation & maintenance
Abstraction and modularity

Software systems are far too complex to be tackled as a whole. Engineers adopt a reductionist approach based on:

**Modularity**
Decompose the system into smaller components

**Abstraction**
Each component behaviour has a simple description, independent of the other components and of the internal implementation

**Benefits**

- **Understandability**
  Complexity of individual components is commensurate to human brainpower

- **Reuse**
The same component can be used in many applications (e.g. transistors)

- **Isolating changes**
The implementation of a component (e.g. transistor materials) can be changed as long as the behaviour (e.g. electrical properties) does not

**Examples**

- **transistor**
  \[ I_C = \beta I_B, \]
  \[ I_E = I_C + I_B, \]
  \[ V_{EB} = 60 \text{ mV} \]

- **operational amplifier**
  \[ V_+ = V_-, I_+ = 0 \text{ A}, I_- = 0 \text{ A} \]

A hierarchy of increasingly-powerful abstractions

From big to small or vice-versa?

**Hierarchical abstractions**

**Top-down design & implementation**
- Design: start from the high-level requirements and progressively decompose the problem into simpler steps until a solution is generated.
- Implementation: difficult as high-level modules cannot be tested until the low-level ones are completed.

**Bottom-up design & implementation**
- Design: it may not be clear which elementary components are needed and how they should be combined in order to get closer to a solution.
- Implementation: easy as when a module is built it can be immediately tested since the modules it depends on have already been constructed.

In practice, both top-down and bottom-up processes progress concurrently in a programmer's head, often informally.

**Some images from Micro-controller, Murray**
Importance and challenges of software engineering
Examples & challenges

Software engineering
The aims and scope of software engineering
Abstraction and modularity

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

Structured programming
Imperative vs declarative languages
Overview of fundamental abstractions

Software processes

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

Key activities of a software process:

Specification
Design & implementation
Verification & validation
Evolution

Software vs traditional products

Software products are intangible
Manufacturing is almost free (copy a file)
There are no manufacturing defects
The product does not take space, weight, age, corrode, etc.

Software engineering ≠ computer hardware engineering
However, they are coupled

Software engineering defects
Often plagued by design & implementation defects (bugs)
The product can and often does change (software updates)

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

Structured programming
Imperative vs declarative languages
Overview of fundamental abstractions

Specification

Elicit requirements from users
Refine requirements until they are sufficiently precise
Produce a requirement specification document

Example
Elicitation: “The program should calculate the trajectory of a lossy bouncing ball”
Refinement: How is the output represented? How are the initial condition specified? The required numerical accuracy? Should air resistance be accounted for? How fast must the program run?
Requirement specification:
Input: initial position and momentum, ball radius, restitution coefficient, ...
Output: simulated trajectory, 60 Hz sampling rate, 10^-3 m tolerance, ...
Must complete simulation in 1ms
Design & implementation

### Design
- **Modules**
  - Architecture: which components
  - Data flow: how components inter-operate
  - Components: how components work
- **Interfaces**
  - Programmatic interface
  - Text-based interface
  - Graphical interface
- **Databases**
  - Data nature and relationships
  - Storage details
  - Security

### Implementation
- A team of programmers writes each component
- Low-level design details (e.g. specific names of local variables) are left to the programmers
- Writing functionalities for debugging are part of the implementation (e.g. unit testing)

### Evolution
- **Cost of software evolution**
  - **Effort**: Engineers might not understand the system anymore, must study it
  - **Risk**: Changes may have unpredictable effects
- **Reengineering software**
  - **Refactoring**
  - **Rewrite** might be needed to:
    - Switch to a new, more modern basis (e.g. new programming language)
    - Scrap old design for a new one
    - Usually keep the interface or at least the functionality equivalent

### Verification & validation
- **Verification**
  - *Black-box* (from software specification) vs *white-box* (from code inspection)
  - *Top-down* (system testing) vs *bottom-up* (unit/component 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: $\tan(-\pi/2)$, $\tan(0)$, $\tan(\pi/2)$
- **Validation**
  - Check that the program solves the problem (includes checking the requirements)

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

---

<table>
<thead>
<tr>
<th>Requirements definition</th>
<th>Software design</th>
<th>Implementation &amp; unit testing</th>
<th>Integration &amp; system testing</th>
<th>Maintenance &amp; evolution</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Incremental models

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

**Extreme programming**

- Tight requirement implementation-test loops
- Applied at different temporal scales:
  - from seconds (implement a function)
  - to months (establish high-level goals for the next software release)
- Design as you go
- Good for small high-risk/speculative projects, bad for nuclear reactors

[WikiMedia]

**Importance and challenges of software engineering**

- Examples & challenges

**Software engineering**

- The aims and scope of software engineering
- Abstraction and modularity

**Software processes & their models**

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

**Structured programming**

- Imperative vs declarative languages
- Overview of fundamental abstractions

**Imperative languages**

- The most common programming languages are **imperative**.

- An imperative program is a list of elementary steps to be executed in a certain sequence in order to solve a problem.

- Different imperative languages are characterised by different abstractions

**Machine Code (Intel x86)**

<table>
<thead>
<tr>
<th>Code</th>
<th>Hex</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>55</td>
</tr>
<tr>
<td>1</td>
<td>48</td>
</tr>
<tr>
<td>2</td>
<td>89</td>
</tr>
<tr>
<td>3</td>
<td>e5</td>
</tr>
<tr>
<td>4</td>
<td>c7</td>
</tr>
<tr>
<td>5</td>
<td>45</td>
</tr>
<tr>
<td>6</td>
<td>fc</td>
</tr>
<tr>
<td>7</td>
<td>00</td>
</tr>
<tr>
<td>8</td>
<td>00</td>
</tr>
<tr>
<td>9</td>
<td>00</td>
</tr>
<tr>
<td>10</td>
<td>00</td>
</tr>
<tr>
<td>11</td>
<td>00</td>
</tr>
<tr>
<td>12</td>
<td>8b</td>
</tr>
<tr>
<td>13</td>
<td>45</td>
</tr>
<tr>
<td>14</td>
<td>fc</td>
</tr>
<tr>
<td>15</td>
<td>03</td>
</tr>
<tr>
<td>16</td>
<td>45</td>
</tr>
<tr>
<td>17</td>
<td>f8</td>
</tr>
<tr>
<td>18</td>
<td>89</td>
</tr>
<tr>
<td>19</td>
<td>45</td>
</tr>
<tr>
<td>20</td>
<td>f4</td>
</tr>
<tr>
<td>21</td>
<td>8b</td>
</tr>
<tr>
<td>22</td>
<td>45</td>
</tr>
<tr>
<td>23</td>
<td>f4</td>
</tr>
<tr>
<td>24</td>
<td>5d</td>
</tr>
<tr>
<td>25</td>
<td>c3</td>
</tr>
</tbody>
</table>

**Machine Language (x86)**

```
pushq %rbp
movq %rsp, %rbp
movl $1, -4(%rbp)
movl $2, -8(%rbp)
movl -4(%rbp), %eax
addl -8(%rbp), %eax
movl %eax, -12(%rbp)
```

Machine language's main abstraction are mnemonics (readable names for instructions, registers, etc.)

**C Language**

```
int f()
{
    int x = 1;
    int y = 2;
    int z = x + y;
    return z;
}
```
Imperative languages

- **Abstractions** can have a massive impact on the ease of use, understandability, maintainability, power, and efficiency of programming languages.

```c
int f()
{
    int x = 1;
    int y = 2;
    int z = x + y;
    return z;
}
```

Declarative (functional) languages

- A **declarative program** specifies the desired behaviour of the program, but now how this is achieved in term of elementary steps.

```regex
[a-zA-Z]*
```

This means that the program should match any string consisting only of letters from 'a' to 'Z'.

It says what the program should do.

C Language

- Data types: elementary, aggregate and compound.
- Variables.
- Control flow: Blocks, conditionals, loops, switches.
- Statements.
- Procedural languages: Functions, data scoping, encapsulation, recursion.
- Object-oriented programming (Part II): Attach behaviour to data.

An overview of fundamental abstractions