COMP61511 (Fall 2017)
Software Engineering Concepts
In Practice
Week 4
Bijan Parsia & Christos Kotselidis
<bijan.parsia, christos.kotselidis@manchester.ac.uk>
(bug reports welcome!)
Preliminaries
What Is Construction?

A definition:

Software *construction* is the creation, assembly, or modification of *executable programs* typically via modification of the *source code*.
Abstraction Hierarchy Of A System

Not the only formulation of such a hierarchy!
Architecture Vs. Construction

System Software Architecture

1. Software system
2. Division into subsystems/packages
3. Division into classes within packages
4. Division into data and routines within classes
5. Internal routine design

Software construction
Coding As Problem Solving

- **Software engineering** is problem solving
  - Hence, the foundational nature of **problem definition**
- **Writing** or modifying code
  - Is also a form of problem solving
    - We hope **smaller** problems.

*Pro tip: Always know the problem you're solving!*
The Big Four (Plus Two)

• Four primary activities
  1. Creating
     ■ We need functionality
  2. Debugging
     ■ We need correctness
  3. Refactoring (last week!)
     ■ We need comprehensibility
  4. Optimising
     ■ We need efficiency (wrt to some resource)

• Plus two
  ■ Testing & Reading
Testing Is Everywhere

- All **primary** activities involve testing
  - Whether **formal** or **informal**
  - E.g., **Creation** (whether test first or not)

<table>
<thead>
<tr>
<th>External Qualities</th>
<th>Functional</th>
<th>Non-Functional</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Correctness</td>
<td>Usability</td>
</tr>
<tr>
<td></td>
<td>Accuracy</td>
<td>Efficiency</td>
</tr>
<tr>
<td></td>
<td>Adaptability</td>
<td>Reliability</td>
</tr>
<tr>
<td><strong>For Modification</strong></td>
<td>Maintainability</td>
<td>Integrity</td>
</tr>
<tr>
<td></td>
<td>Flexibility</td>
<td>Robustness</td>
</tr>
<tr>
<td></td>
<td>Portability</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Reusability</td>
<td></td>
</tr>
<tr>
<td><strong>For Comprehension</strong></td>
<td></td>
<td>Readability</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Understandability</td>
</tr>
<tr>
<td><strong>Testability</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Internal Qualities**
Reading Is Everywhere

- Reading **code** is a key skill
  - **Other** people's code
    - that you are **using**
    - that you are **modifying**
  - **Your** own code!
    - whether **using** or **modifying**
- "Reading" (understanding) **systems** is a key skill
  - **Grasping** the problem, requirements, architecture
  - **Relating** code to those
10 lines of code = 10 issues. 500 lines of code = "looks fine." Code reviews.

— I Am Devloper (@iamdevloper) November 5, 2013
Project Effects On Product Qualities
Although it might seem that the best way to develop a high-quality product would be to focus on the product itself, in software quality assurance you also need to focus on the software-development process.
— McConnell, 20.2

Poor quality processes raises the risk of poor quality products
A Key Point (2)

The General Principle of Software Quality is that improving quality reduces development costs. McConnell, 20.5

Counterintuitive principle!
1. Poor processes raise the risk of poor products
2. Improving quality reduces development costs

But...pick two:
HOW TO WRITE GOOD CODE:

1. Start project.
2. Do things right or do them fast?
   - Right
   - Code well
   - Are you done yet?
     - No
     - No, and the requirements have changed.
     - Throw it all out and start over.

3. Code fast
   - Does it work yet?
     - No
     - Almost, but it's become a mass of kludges and spaghetti code.

4. Good code
Question Time!!

* Does the Good-Fast-Cheap/Pick-2 triangle + the general principle imply that
  
  1. quality software **must** take a long time
  2. quality software is **impossible**
  3. the **triangle** is false
  4. the **general principle** is false
Cost Of Detection

McConnell, 3.1

Table 3-1. Average Cost of Fixing Defects Based on When They're Introduced and Detected

<table>
<thead>
<tr>
<th>Time Introduced</th>
<th>Requirements</th>
<th>Architecture</th>
<th>Construction</th>
<th>System Test</th>
<th>Post-Release</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirements</td>
<td>1</td>
<td>3</td>
<td>5–10</td>
<td>10</td>
<td>10–100</td>
</tr>
<tr>
<td>Architecture</td>
<td>—</td>
<td>1</td>
<td>10</td>
<td>15</td>
<td>25–100</td>
</tr>
<tr>
<td>Construction</td>
<td>—</td>
<td>—</td>
<td>1</td>
<td>10</td>
<td>10–25</td>
</tr>
</tbody>
</table>

Cost Of Detection

McConnell, 3.1

Figure 3-1. The cost to fix a defect rises dramatically as the time from when it’s introduced to when it’s detected increases. This remains true whether the project is highly sequential (doing 100 percent of requirements and design up front) or highly iterative (doing 5 percent of requirements and design up front).
Project Qualities Per Se

- We've only talked about **product**
  - **Projects** have qualities too!
  - E.g.,
    - Being on (or off) budget and schedule
    - Being well run
    - Being well "resourced"
    - Being popular
    - Using a certain methodology (correctly (or no))
- Since project qualities influence product qualities
  - We have to study them as well!
  - There is an interaction
Complexity
Complexity Challenge

But when projects do fail for reasons that are primarily technical, the reason is often uncontrolled complexity... When a project reaches the point at which no one completely understands the impact that code changes in one area will have on other areas, progress grinds to a halt.
Complexity Challenge

- McConnell, 5.2 "Software's Primary Technical Imperative has to be managing complexity."
- Architecture is key to managing Complexity
  - Provides a guide
  - Good architecture controls interaction
  - Allows independent consideration of subsystems
Dealing With Complexity

- We **can not** understand the entire complex system
- We hide information via:
  - **Modularisation**
  - **Abstraction**
- ...to be able to effectively deal with complexity
Modularity And Abstraction

- We get intellectual leverage to understand and reason about subsystems
- Apply these concepts at different levels
- Understanding enables us to:
  - Comprehend, Maintain, Extend our systems
Levels Of Design

- **Modularity**
  - Confines the details
  - Facilitates Abstraction
- As we **move up** levels
  - We loose details
  - Expand our scope of **understanding**
  - Good design/construction allows us to safely ignore details
Design Levels

1. Software system
2. Division into subsystems/packages
3. Division into classes within packages
4. Division into data and routines within classes
5. Internal routine design
Components Example

McConnell, 5.2: Figure 5-3. An example of a system with six subsystems
Complexity "Unconstrained"

McConnell, 5.2: Figure 5-4. An example of what happens with no restrictions on inter-subsystem communications.
McConnell, 5.2: Figure 5-5. With a few communication rules, you can simplify subsystem interactions significantly.
Levels Of Modularity

- Modularity, Encapsulation and Interfaces at different levels:
  - Subsystem
  - Package
  - Class
  - Routine
Design As An Activity

- Can be found in many fields
  - e.g., Architecture, Civil Engineering, Computer architecture
- Characteristics of software design:
  - Knowledge of three domains (maybe more):
    - Applications, Technical domain, Design domain
  - Motivated choices and tradeoffs
  - What to consider and what to ignore
  - Multi-faceted and multi-level
"Horst Rittel and Melvin Webber defined a wicked problem as one that could be clearly defined only by solving it, or by solving part of it (1973)." McConnell, 5.1
Change Is A Reality

- **Requirements** and problem definitions change
  - **Exogenously**: the external world changes
    - e.g. a regulation is passed during development
  - **Endogenously**: triggered by the evolving system
    - e.g. people understand better the system
Software Development Must Cope

- **Methodologically**, e.g. agile methods tailored for changes in requirements
- **Architecturally**, e.g. modularity let us replace modules
- **Constructionally**, e.g. robust test suites support change
Direction Of Design

- **Top down**
  - Start with the general problem
  - Break it into manageable parts
  - Each part becomes a new problem
  - Decompose further
  - Level out with concrete code

- **Bottom up**
  - Start with a specific capability
  - Implement it
  - Repeat until able to think about higher level pieces
Opportunistic Focus

• Top down and bottom up are not exclusive
  ■ Thinking from the top
    ○ Focuses our attention on the whole system
  ■ Thinking from the bottom
    ○ Focuses our attention on concrete issues
• Choosing where to focus our attention opportunistically is useful
  ■ Reason about top level by realising code at lower levels
Exploring The Design Space

- **Wickedness** suggests
  - we need to **do stuff early**
  - build experimental solutions
- Three common forms
  - Spikes
  - Prototypes
  - Walking skeletons
Spikes

- Very small program to explore an issue
  - Scope of the problem is small
- Often intended to determine specific risk
  - Is this technology workable?
- No expectation of keeping
Prototypes

• May have some small or large scope
• Intended to demonstrate something
  ■ rather than ‘just’ find out about technology (a spike)
• Mock ups through working code
• Can be “on paper”!
• Prototypes get thrown away
  ■ ...or are intended to!
Walking Skeletons

- **Small** version of “complete” system
  - “tiny implementation of the system that performs a small end-to-end function. It need not use the final architecture, but it should link together the main architectural components. The architecture and the functionality can then evolve in parallel.” - Alistair Cockburn

- Walking skeletons are meant to *evolve* into the software system
Beyond Lines Of Code: Do We Need More Complexity Metrics?
"Software's **Primary Technical Imperative** has to be managing complexity." (McConnell, 5.2)

- What *is* complexity?
- How do we **know** if we're managing it?
- Can we **tell** if a change
  - *increases* or *decreases* complexity

- Complexity/Complication might not be **obvious**
  - Some things might *seem* more than they are
Contrast

```python
print(0)
print(1)
print(2)
print(3)
```

```python
for i in range(4):
    print(i)
```

or

```python
print(0)
print(2)
```

```python
for i in range(4):
    if i % 2 == 0:
        print(i)
```
Metrics

• We need **metrics**
  ▪ I.e., a **measure** of complexity
• Consider 2
  ▪ (Source) Lines of Code: **(S)LOC**
    ◦ I.e., as measured by *wc* (modified)
Cyclomatic Complexity

- Count the **linearly independent paths**
- **Average** vs. **Max CYCLOmatic Complexity**

*FIGURE 8-1. Two sample graphs. Graph (a) has a CYCLO of 1, and graph (b) has a CYCLO of 3.*
Which Measure Is Better? (Pg 133)

- Analyse ArchLinux packages (2010)
  - 4,015 packages, containing **1,272,748 source code files**
  - 576,511 were **written in C**
  - 338,831 are **unique**
  - 212,167 **nonheader**; 126,664 **header**
- **Run** each of a number of metrics on each file
  - Compare!
Results For Nonheader Files

**TABLE 8-4. Correlation coefficients for nonheader files**

<table>
<thead>
<tr>
<th></th>
<th>SLOC</th>
<th>MCYCLO</th>
<th>ACYCLO</th>
<th>HLEVE</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLOC</td>
<td>1.00</td>
<td>0.83</td>
<td>0.60</td>
<td>0.91</td>
</tr>
<tr>
<td>MCYCLO</td>
<td>1.00</td>
<td></td>
<td>0.86</td>
<td>0.82</td>
</tr>
<tr>
<td>ACYCLO</td>
<td></td>
<td>1.00</td>
<td></td>
<td>0.65</td>
</tr>
<tr>
<td>HLEVE</td>
<td></td>
<td></td>
<td>1.00</td>
<td></td>
</tr>
</tbody>
</table>

**HLEVE is Yet Another Metric**
Question!

- The high correlation between complexity measures means:
  1. they are all equally good.
  2. they are all equally bad.
  3. they give the same information.
  4. we can't tell!
Some (Tentative) Conclusions

- With respect to **amount**
  - more LOC == more complexity
  - doesn't tell use **why** or **how**
  - (and this is C non-header files)
- Other metrics might tell us **other things**
  - Cyclomatic complexity tells us **minimum number of tests for line coverage**
Reflect!

- Even the **measurement** of complexity
  - Is **complex**!
  - And **contestable**
    - Always "on another hand"
- Complexity on **many** levels
  - "First order": **this code** is a mess
  - "Second order": **this complexity metric** is a mess
  - "Third order": **complexity measurement** is a mess!
- It's messes **all the way up**!
  - Part of your job is to develop coping strategies.
Creation

Mindset
What Is (Code) Creation?

Code creation (or coding) is the addition of new functionality by the generation of new code and units of code.

- **Key** activity!
  - Often directly measured
    - Productivity as LOC/day
    - (Though, deleting code might be better!)
  - Does not have to be *ex nihilo*
    - Cut-paste-modify reuse counts
    - Reuse counts!
Prerequisites

• Remember the prerequisites!
  ■ What's your **overall problem definition**
    ◦ What **part** are you tackling
  ■ What are the **pertinent requirements**
  ■ Understand the **architecture**
    ◦ And how your **current code** fits in
  ■ Know the **local standards**
    ◦ E.g., code formatting style
Architecture

• A good architecture should:
  1. help you determine **where** your code should go
  2. constrain how functionality is **divvyed up**
  3. determine your **communication** channels
  4. give you a sense of things **fitting together**
     ■ that is **shared**

• Code-Architecture **conflicts** indicate
  ■ A **problem** with one or the other
  ■ A **limit**
Awarenesses

- Situational Awareness
  - Your perception of the current pertinent factors for decision making
  - **Good** situational awareness
    - Tracks *all pertinent* factors
    - to the *right* degree
    - in a manner to *drive appropriate reactions*
    - at *low cost*
  - Drives *tactics* and thus *action*

- Understanding
  - Your *systematic grasp* of all factors related to decision making
  - Results from *sensemaking*
  - More *cognitive* (*indirectly* drives action)
Getting In The Zone

- Given a problem, our solving can be
  - **focused**
    - we have **tight** situational awareness
    - the "situation" is the problem and solution space
    - we **react** rather than **act**
  - **unfocused**
    - our awareness is **scattered**
      - distracted/multitasking
      - disengaged
      - confused

_The "zone" is a much higher productivity state_
Admin

- Record-keeping is extremely **helpful**
  - And sometimes required, e.g., billable hours
- Tracking helps! (a lot can be automated)
  - Time
  - Effort (and sense of effort)
  - What was done (and why, by whome)
  - Mood
  - Discussions and decisions

*Some is better than none; enough is better still; there is too much*
Let’s Take A Look!
Debugging

—Grace Hopper's Bug Report
Defects Again

Recall:

A defect in a software system is a quality level (for some quality) that is not acceptable.

- We focus on functional defects
  - Correctness primarily
  - Though robustness is also key
    - More stability, i.e., doesn't crash
What Is Debugging?

Debugging is the modification of code to remove (or mitigate) correctness defects.

- We don't count **missing** functionality defects
- Debugging starts **after** a purported **detection**
  - Input: a result of testing or a bug report
- We allow **mitigation**
  - Not **properly** fixing the bug
  - But enough so it's **less damaging**
  - Must still involve **code modification**
    - Other workarounds don't count!
Functional Landscape (Enhanced)
Debug Cycle

• Input: An indication of a defect
  ■ **Stabilise** — Make reliably repeatable
  ■ **Isolate** (or localise) — To the **smallest unit**
  ■ Explain — **What's wrong** with the code
  ■ Repair — **Replace** the broken code
  ■ Test — **Verify** the fix

• Check for
  ■ Regressions
  ■ **Masked** bugs
  ■ **Nearby** bugs
Indication

An **indication** of a defect is a **tangible record** of a behaviour contrary to the (explicit or implicit) functional specification in a **designated situation**.

- Key parts:
  - Situation
    - Preferably, sufficiently described for replication
  - Expected Behaviour
  - Witnessed Behaviour
    - Typically with some explanation why it's wrong
- Often **very vague**
Indication?

- Often *very vague*
  - Program crashed *sometime* during this test
  - Open Office on Ubuntu won't print
    - Actually, only on Tuesdays!

*From John Regehr, "Classic Bug Reports"*
Basic Debug Cycle

1. **Stabilize** → **Repeatability** → **Isolate** → **Bug Area** → **Explain**
2. **A Bug Theory** → **Repair** → **Good Code?**
3. **Not Good** → **Test** → **Good!** → **Post Fix Checks**
Stabilise

• Bugs are often very **situation dependent**
  - Precise input + state
    - OS, hardware
    - Sequence of actions
    - Length of operating
  
• A stabilised bug
  - is **reliably** repeatable
  - preferably with **minimal** sufficient conditions
Isolate (Localise)

- Bugs are often very **local**
  - **Single** LOC
  - **Single** routine
  - **Particular** class
- They don't **have** to be!
  - Communication points are **vulnerable**
- A defect is **isolated** if
  - you have identified the **minimum subsystem necessary** to exhibit the defect
  - for an **trigger input and situation**
Explain & Repair

• **Explaining** the bug
  - You can *articulate the mechanism* of the bug
    - Your *bug theory*
  - You can *manipulate* the bug
    - *Trigger* or *avoid* it
    - Produce *variants*
    - *Predict* its behaviour
    - *Fix it*

• **Repairing** the bug
  - Modifying the code so the defect is eliminated
  - May not be possible!
Test

- Post fix
  - You need to **verify**
    - Your **theory**
    - Your *execution* of the fix
  - You need to **guard against**
    - Unintended consequences!
- "New" bugs **arise**
  - Bugs in the **fix**
    - The fix is **incomplete**
    - The fix triggers a **regression**
  - Masked **bugs**
Post Successful Fix!

- Post Fix Checks
  - Regressions
    - Broke Other Stuff
      - DEBUG! or Revert
  - Masked Bugs
    - Fixed Bug Hid Bugs
      - DEBUG!
  - Nearby Bugs
    - Buggy Pattern Generated Bugs
      - DEBUG!
Check Nearby

- Bugs come in **families**
  - **Similar** mistakes
    - You did it once, you might have done it twice
    - Persistent misunderstanding with multiple manifestations
  - **Clustered** mistakes
    - Some bugs **hidden**
      - A crash conceals much
    - Some routines are **broken**
      - Lots of debt!
- A bug is a predictor of more bugs!
Bug Reports To WONTFIX

• Sometimes, a fix isn't going to happen
  ■ The bug is too small
    ◦ Or insignificant
    ◦ Or ambiguous
  ■ The bug is too big
    ◦ It would change too much behavior
      ◦ Which some people rely on
    ◦ Other debt increases the risk
  ■ The but is too hard
def get_console_output(script, file_path):
    try:
        output = subprocess.check_output(['python', script, file_path], stderr=subprocess.STDOUT, timeout=200).decode('ascii')
    except subprocess.CalledProcessError:
        return "-1 " * 4
    except OSError:
        print("No such file or directory.")

- Ascii was a reasonable for shell output.
- “We don’t handle that situation yet!”
  - Introduces unicode “by accident”.
  - Copy and Paste (the right thing) breaks this with an exception.
- Repetitive statement throughout test suite!
Is It A Bug?

```python
file_content = file.read()
lines = file_content.count('
')
```

VS

```python
def getLines(filename):
    file = open(filename, 'rb')
    num_lines = 0
    for line in file:
        num_lines += 1
    return num_lines
```
Optimising
Resources

- Size
  - Running space
    - At all levels
  - Persistence and transmission
  - Code

- Time
  - Response vs. throughput
    - Instant vs. Overall
  - Wall/CPU Time/Instructions
What Is Optimisation?

Optimisation is a transformation of code into sufficiently functionally equivalent code that has "better" resource utilisation.

- "Sufficiently functionally equivalent"
  - **User observable/desirable** behaviour is preserved
  - Up to some point
  - It may be **specialised** to a certain particular scenario
- Resource utilisation
  - **Type** and **Pattern** must be specified
Where?
Tuning Trade-Offs

- **Time** for **Space** (and the reverse)
- **Performance** for **Readability** (and the reverse)
  - And other comprehension qualities
  - Not **always** a trade off for algorithmic improvements
    - Or fat removal
- Performance for **Correctness**
- Performance for **Cost**
Tuning Alternatives

- Buy More and Faster **Hardware**
- Use the **Optimiser**
- **Better** compilers/frameworks/libraries
- **Input** manipulation
  - "It's slow when I do this" "Don't do that!"
Tuning Safety

- Tuning is **risky**
  - Even optimisation can be risky!
- It's easy to make code **fast**
  - By making it **incorrect**
- It's easy to **modify the code** a lot
  - And **not improve** performance much
  - Or **make worse**
Tuning As (Performance) Debugging

• Input: An **indication** of a performance defect
  - **Stabilise** — Make **reliably repeatable**
  - **Isolate** (or localise) — To the **smallest unit**
    - USE A PROFILER! TEST CASES ARE CRITICAL
    - Explain — **What's wrong** with the code
  - Repair — **Replace** the "slow" code
  - Test — **Verify** the improvements

• Check for
  - **Sufficiency** (Was that enough?)
  - **Trade-offs** (e.g., space consumption)
  - (Correctness) **Bugs**