COMP61511 (Fall 2017)
Software Engineering Concepts
In Practice
Week 2
Bijan Parsia & Christos Kotselidis

<bijan.parsia,
christos.kotselidis@manchester.ac.uk>
(bug reports welcome!)
Who Owns Your Code?

- You **wrote** some code!
  - All week!
  - Both systems and tests!
- A key question:
  - Who **owns** that code?
    - Or different bits of it?
  - What **kind** of ownership?
Intellectual Property (IP)

**Intellectual property** is any articulable, tangible production of a mind whose physical realisations are restricted by law (in production, distribution, etc.)

- We don't control what other people think!
- We can control what they **do** with certain thoughts.
- Intellectual Property rights give power to certain people to control what other people do
  - For example, whether they can distribute a book, song, or program
## Kinds Of Intellectual Property

<table>
<thead>
<tr>
<th>Name</th>
<th>Establishment</th>
<th>Enforcement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copyright</td>
<td>Automatic, immediate</td>
<td>Civil and Criminal</td>
</tr>
<tr>
<td>Patent</td>
<td>Application; exposure before application destroys it</td>
<td>Mostly civil</td>
</tr>
<tr>
<td>Trademark</td>
<td>Application and vigorous defense</td>
<td>Mostly civil</td>
</tr>
<tr>
<td>Trade Secret</td>
<td>Automatic (by not telling people) and NDAs</td>
<td>Mostly civil</td>
</tr>
</tbody>
</table>
Copyright

*Copyright* is a licensable monopoly of *tangible* expression of an idea with respect to reproduction, derivation, display, distribution, and the like.

- Protects the *expression* not the *idea*
  - Though these *blur* at the limit
    - Some plagiarism is a copyright violation; some is not
- Typically *automatically assigned at creation time*
  - No "notice" or "registration" needed
    - Though these might help with lawsuits
A **patent** is a licensable monopoly of the use or sale of a "non-obvious" **invention** (of a process, machine, design (sometimes), mechanism, procedure, etc.

- A patent is an **incentive** to **disclose**
  - Many patentable inventions **could** be exploited "secretly"
  - Goal is to **add** to our **common** knowledge
- **Prior art** destroys a patent
  - Including **your own**
- **Defensive** patenting "common"
- **Independent** invention no defense
Trade Secret

A trade secret is an invention which is not disclosed

- Persists forever
  - Unless leaked
  - Or reinvented
- Typically protected by secrecy
  - Or specific contracts
    - "Non-Disclosure Agreements" (NDAs)
Who Owns Your Code?

- Copyright **starts** with the **creator**
  - I.e., you!
  - Cheap! (Even to register)
  - Unless you create it as **work-for-hire**
    - Or **otherwise** transfer it
- Patents belong to the **patenter**
  - Expensive(ish) to secure
- Trade secrets belong to the **inventor**
Are You Working For Hire?

- Not **quite**!

### 3.6. Student IP Licence to the University

3.6.1. Each Student grants to the University a licence to use the:

3.6.1.1. IP created by him or her in the course of his/her studies at the University and which they own; and

3.6.1.2. IP in any thesis or dissertation submitted to the University for the award of a degree.

3.6.2. In each case, the licence will take effect upon the creation of the relevant IP.

3.6.3. The licence:

3.6.3.1. is a continuing, non-exclusive, worldwide, irrevocable, royalty-free licence to use the IP in any format (whether existing or future);

3.6.3.2. will last for as long as the relevant IP remains in existence; and

3.6.3.3. is granted so that the University can (i) use such IP for its and its subsidiaries' administrative, promotional, educational and teaching purposes; and (ii) do all such things in relation to such IP which would otherwise be an infringement of such IP.

3.6.4. As part of the licence, the University is also permitted to sub-license to others the rights granted to it by Students.

3.6.5. Any thesis or dissertation submitted to the University for the award of a degree may be placed by the University in its institutional repository in electronic or other format.
What To Keep In Mind (Now)

- Software engineers typically **produce** IP
  - Even if not protected, our **output** is "intellectual"
  - Various forms of IP drive
    - **product** value
    - **employee/entrepreneur** value
- Software engineers typically **use** IP
  - All sorts and in all ways
  - IP considerations a constraint on the design space
Comprehending Product Qualities
Comprehension?

- We can distinguish two forms:
  - Know-that
    - You believe a **true** claim about the software
    - ...with appropriate evidence
  - Know-how
    - You have a **competency** with respect to the software
    - E.g., you know-how to recompile it for a different platform
- They are interrelated
- Both require significant effort!
Quality Levels

• We talked about different kinds of quality
  ▪ But for each kind there can be degrees or levels thereof
  ▪ "Easy" example: High vs. Low performance

• Most qualities in principle are quantifiable
  ▪ Most things are quantifiable in some sense

• But reasonable quantification isn't always possible
  ▪ Or worth it
  ▪ Being clear about your vagueness is essential!
Clarity

Our discussion will be adequate if it has as much clearness as the subject-matter admits of, for precision is not to be sought for alike in all discussions, any more than in all the products of the crafts... for it is the mark of an educated [person] to look for precision in each class of things just so far as the nature of the subject admits...

— Aristotle, Nicomachaen Ethics, Book 1, 3
Clarity (2)

*We demand *rigidly defined* areas of *doubt* and *uncertainty*!*
Defects As Quality Lacks

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

- Quality levels need to be elicited and negotiated
  - All parties must agree on
    - what they are,
    - their operational definition
    - their significance

What counts as a defect is often determined late in the game!
Question

If your program crashes then it

1. definitely has a bug.
2. is highly likely to have a bug.
3. may or may not have a bug.
Question

If your program crashes, and the cause is in your code, then it

1. definitely has a bug.
2. is highly likely to have a bug.
3. may or may not have a bug.
Bug Or Feature?

(Does QA hate you? — scroll for the cartoons as well as the wisdom.)

- Even a crashing code path can be a feature!
- Contention arises when the stakes are high
  - and sometime the stakes can seem high to some people!
  - defect rectification costs the same
    - whether the defect is detected...
    - ...or a feature is redefined
- Defects (even redefined features) aren't personal
Problem Definition

This is a logical, not temporal, order.
The penalty for failing to define the problem is that you can waste a lot of time solving the wrong problem. This is a double-barreled penalty because you also don't solve the right problem.

—McConnell, 3.3
Quality Assurance

- Defect **Avoidance** or **Prevention**
  - "Prerequisite" work can help
    - Requirement negotiation
    - Design
    - Tech choice
  - Methodology
- Defect **Detection** & Rectification
  - If a defect exists,
    - Find it
    - Fix it
The Points Of Quality

1. Defect **prevention**
   - Design care, code reviews, etc.
2. Defect **appraisal**
   - Detection, triaging, etc.
3. **Internal** rectification
   - We fix/mitigate before shipping
4. **External** rectification
   - We cope after shipping
## Defect Detection Techniques

<table>
<thead>
<tr>
<th>Removal Step</th>
<th>Lowest Rate</th>
<th>Modal Rate</th>
<th>Highest Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Informal design reviews</td>
<td>25%</td>
<td>35%</td>
<td>40%</td>
</tr>
<tr>
<td>Formal design inspections</td>
<td>45%</td>
<td>55%</td>
<td>65%</td>
</tr>
<tr>
<td>Informal code reviews</td>
<td>20%</td>
<td>25%</td>
<td>35%</td>
</tr>
<tr>
<td>Formal code inspections</td>
<td>45%</td>
<td>60%</td>
<td>70%</td>
</tr>
<tr>
<td>Modeling or prototyping</td>
<td>35%</td>
<td>65%</td>
<td>80%</td>
</tr>
<tr>
<td>Personal desk-checking of code</td>
<td>20%</td>
<td>40%</td>
<td>60%</td>
</tr>
<tr>
<td>Unit test</td>
<td>15%</td>
<td>30%</td>
<td>50%</td>
</tr>
<tr>
<td>New function (component) test</td>
<td>20%</td>
<td>30%</td>
<td>35%</td>
</tr>
<tr>
<td>Integration test</td>
<td>25%</td>
<td>35%</td>
<td>40%</td>
</tr>
<tr>
<td>Regression test</td>
<td>15%</td>
<td>25%</td>
<td>30%</td>
</tr>
<tr>
<td>System test</td>
<td>25%</td>
<td>40%</td>
<td>55%</td>
</tr>
<tr>
<td>Low-volume beta test (&lt;10 sites)</td>
<td>25%</td>
<td>35%</td>
<td>40%</td>
</tr>
<tr>
<td>High-volume beta test (&gt;1,000 sites)</td>
<td>60%</td>
<td>75%</td>
<td>85%</td>
</tr>
</tbody>
</table>

Source: Adapted from *Programming Productivity* (Jones 1988a), "Software Defect-Removal Efficiency" (Jones 1996), and "What We Have Learned About Fighting Defects" (Shull et al. 2002).
Defect Detection Techniques

- Regression test
- Informal code reviews
- Unit test
- New function (component) test
- Integration test
- Low-volume beta test (< 10 users)
- Informal design reviews
- Personal desk checking of code
- System test
- Formal design inspections
- Formal code inspections
- Modeling or prototyping
- High-volume beta test (> 1000 users)
Experiencing Software

- It's one to know that there are bugs
  - All software has bugs!
- It's another to be able to trigger a bug
  - Not just a specific bug!
  - If you understand the software
    - You know how to break it.
- Similarly, for making changes
  - tweaks, extensions, adaptions, etc.
- The more command, the more modalities of mastery
Forms Of Knowledge (Manifestations)

- **Human interpretable**
  - Comments, design docs, user stories, javadoc
  - **Source code**
    - Both, a written description and a "live" object
    - Also things like demo code, examples, test suites, etc.

- **Diagrams**
  - "Mere" pictures to semi-formal to formal diagrams: ER docs, UML, etc.

- **Formal specifications**
- **Competencies**
  - I can make it crash
Sources Of Knowledge (Modalities)

- **Analytical** knowledge
  - Derived from *inspection* and *reasoning*
  - Can be *automated* using formal methods
- **Experimental** knowledge
  - Derived from the conduct of experiments
  - Typically tests
- **Experiential** knowledge
  - Derived from *personal interaction* with the software
  - Strong "know-how" component
Coursework Recap

KEEP CALM AND CARRY ON
Coursework Activities

- Reading
- Q1
  - Mostly related to reading
  - Mostly "Recall"...with some interpretation
    - They will go higher on the Bloom taxonomy!
- SE1
  - Reading and analysing
- CW1
  - Reverse engineered a specification
  - Reengineered miniwc from the spec
    - Program construction
A Note On Marks

- UK marks run from 0-100%
  - <=49 = Failing (<40% serious failure)
  - 50-59 = Pass
  - 60-69 = Merit
  - over 70 = Distinction
  - NOTE THE WIDE BAND AT THE TOP AND BOTTOM
- A 65% is a good mark
- An 85% is exceedingly rare
- Over 70% is fairly rare
Q1

- Mean of 3.71 (74%)
  - This is unusual for MCQs
- We will do some "in exam conditions"
- Let's delve
Simplified Problem

- This was a **small** problem
  - With clear boundaries
- Even here:
  - We ended up with **support programs**
    - And corners cut
- Software engineering is (complex) system engineering
  - On both the **product** and **project** sides
  - We use a complex infrastructure!
Challenges

What were the challenges you encountered?

What challenges were inherent to the problem?

What challenges were environmental?
FizzBuzz And Golf!

• Will have some more data soon!
• But we have a winner!

<table>
<thead>
<tr>
<th>ID</th>
<th>Lines</th>
<th>Words</th>
<th>Char</th>
</tr>
</thead>
<tbody>
<tr>
<td>j02654hg_Lab1</td>
<td>1</td>
<td>8</td>
<td>83</td>
</tr>
<tr>
<td>mbax9jc7_Lab1</td>
<td>1</td>
<td>7</td>
<td>94</td>
</tr>
<tr>
<td>mbaxtmb3_Lab1</td>
<td>1</td>
<td>13</td>
<td>104</td>
</tr>
<tr>
<td>mbaxtod2_Lab1</td>
<td>9</td>
<td>15</td>
<td>134</td>
</tr>
<tr>
<td>mbaxtsd2_Lab1ERROR</td>
<td>13</td>
<td>23</td>
<td>155</td>
</tr>
</tbody>
</table>
Testing Correctness

*Beware of bugs in the above code; I have only proved it correct, not tried it.*
— Don Knuth, *Figure 6: page 5 of classroom note*
Really Beware Of Bugs!

<table>
<thead>
<tr>
<th>Time</th>
<th>Action</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:00</td>
<td>Test started</td>
<td></td>
</tr>
<tr>
<td>0:10</td>
<td>Components started</td>
<td></td>
</tr>
<tr>
<td>0:20</td>
<td>Components stopped</td>
<td></td>
</tr>
<tr>
<td>0:30</td>
<td>Microprocess started</td>
<td></td>
</tr>
<tr>
<td>1:00</td>
<td>Cosine Tape started</td>
<td></td>
</tr>
<tr>
<td>1:10</td>
<td>Muly Adder Test</td>
<td></td>
</tr>
<tr>
<td>1:30</td>
<td>Relay 29 Panel F</td>
<td>First actual case of bug being found</td>
</tr>
<tr>
<td>2:00</td>
<td>Cleaned up</td>
<td></td>
</tr>
</tbody>
</table>

—Grace Hopper's Bug Report
Developer Testing

- We can distinguish between
  - Testing done by **non-specialists**
    (McConnell: "Developer testing")
    - For many projects, the only sort!
  - Testing done by (test) **specialists**
- If you **compile** and **run** your code
  - Then you've done a test! (or maybe two!)
    - If only a "**smoke**" test

*Testing is inescapable; good testing takes work*
Question Time!

- If you **compile** your code
  - you have tested it for syntactic correctness.
  - you have tested it for semantic correctness.
  - you have tested it for both.
  - you haven't tested it at all.
What Is A Test?

A test case is a repeatable execution situation of a software system that produces recordable outcomes. A test is a particular attempt of a test case.

- The outcomes may be expected (i.e., specified in advance)
  - E.g., we expect passing 1+1 to a calculator to return 2
  - Generally boolean outcomes (pass or fail)
    - We might have an error that prevents completion
- The outcomes may be measurement results
  - E.g., we want to find the time it takes to compute 1+1
What Is A Test? (2)

A test case is a repeatable execution situation of a software system that produces recordable outcomes. A test is a particular attempt of a test case.

- The outcomes should testify to some software quality
  - E.g., correctness, but also efficiency, usability, etc.
- A (single) test specifies a very particular quality
  - E.g., correct for a given input
  - E.g., uses X amount of memory for this scenario

The fundamental challenge of testing is generalisability
Generalisability Problem (1)

*Testing shows the presence, not the absence of bugs.*  
Terminology Note

- **Test** and **test case** are often used interchangeably
  - And in other loose ways
  - Most of the time it doesn't matter because easy to distinguish
- We often talk about a **test suite** or **test set**
  - But this also might be subordinated to a **test**
  - For example,

  "We used the following **test suite** to **stress test** our application".
Anatomy Of A Test (1)
Generalisability Threat

- A test case (A):
  - **Goal:** Correctness to the specification
    - **Input:** a pair of integers, $X$ and $Y$
    - **Output:** the integer that is their sum
  - **Test Input:**
    - $X=1$ and $Y=1$
  - **Expected output:**
    - 2
- Test result of System S is **pass**
  - What can we conclude?
Question Time!

- From the test result **Pass test case A**, we can conclude that:
  1. System S **correctly implements** the specification.
  2. System S **correctly implements** the specification **for this input**
  3. Both 1 and 2
  4. Neither 1 nor 2
Question Time!

- From the test result **Fail test case A**, we can conclude that:
  1. System S **does not correctly implement** the specification.
  2. System S **does not correctly implement** the specification **for this input**
  3. Both 1 and 2
  4. Neither 1 nor 2
Anatomy Of A Test (2)
Environment Matters

The next most significant subset of [Modification Requests (MRs)] were those that concern testing (the testing environment and testing categories)—24.8% of the MRs. ...it is not surprising that a significant number of problems are encountered in testing a large and complex real-time system...First, the testing environment itself is a large and complex system that must be tested. Second, as the real-time system evolves, so must the laboratory test environment evolve.

A Good Test

- A good test case is
  - **part** of a suite of test cases
  - **understandable**
    - i.e., you can relate it
    - to the spec
    - to the system behavior
  - **fits in** with the test environment
  - is (given the suite) **informative**
## Environment Matters

**Table 25-1 Summary of faults (modified) Making Software pg 459**

<table>
<thead>
<tr>
<th>MR Category</th>
<th>Proportion</th>
<th>D+C</th>
<th>D+C+R</th>
</tr>
</thead>
<tbody>
<tr>
<td>Previous</td>
<td>4.0%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Requirements</td>
<td>4.9%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Design</td>
<td>10.6%</td>
<td>D+C</td>
<td>D+C+R</td>
</tr>
<tr>
<td>Coding</td>
<td>18.2%</td>
<td>28.8%</td>
<td>33.7%</td>
</tr>
<tr>
<td>Testing Environment</td>
<td>19.1%</td>
<td>TE+ T</td>
<td></td>
</tr>
<tr>
<td>Testing</td>
<td>5.7%</td>
<td>24.8%</td>
<td></td>
</tr>
<tr>
<td>Duplicates</td>
<td>13.9%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>No problem</td>
<td>15.9%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Other</td>
<td>7.8%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
A Classification Of Tests
A Classification Of Tests

- Based on a 5W+H approach by **Ray Sinnema** (archived)
  - **Who** (Programmer vs. customer vs. manager vs...)
  - **What** (Correctness vs. Performance vs. Useability vs...)
  - **When** (Before writing code or after)
    - Or even before architeciting!
  - **Where** (Unit vs. Component vs. Integration vs. System)
    - Or lab vs. field
  - **Why** (Verification vs. specification vs. design)
  - **How** (Manual vs. automated)
    - On demand vs. continuous
Who?

- Sinnema: Tests give confidence in the system
  - I.e., they are evidence of a quality
  - Who is getting the evidence?
    - Users? Tests focus on external qualities
      - Can I accept this software?
    - Programmers? Tests focus on internal qualities
      - Can I check in this code?
    - Managers? Both?
      - Are we ready to release
  - But also, who is writing the test?
    - A bug report is a (typically partial) test case!
What?

- Which **qualities** am I trying to show?
  - Internal vs. external
  - Functional vs. non-functional?
  - Most developer testing is functional (i.e., correctness)
    - And at the unit level
    - Does this class **behave as designed**
When?

- **When** is the test written?
  - **Before** the code is written?
  - **After** the code is written?

- Perhaps a better distinction
  - Tests written with *existing code/design in mind*
  - Test written **without regard** for existing code/design
  - This is related to white vs. black box testing
    - Main difference is whether you **respect the existing API**
Where?

- **Unit**
  - Smallest "chunk" of coherent code
  - Method, routine, sometimes a class
  - *McConnell*: "the execution of a **complete class, routine, or small program** that has been written by a **single programmer or team of programmers**, which is tested **in isolation** from the more complete system"

- **Component** (McConnell specific, I think)
  - "work of **multiple programmers or programming teams**" and **in isolation**
Where? (Ctnd)

- **Integration**
  - Testing the *interaction* of two or more units/components
- **System**
  - Testing the system as a whole
  - In the lab
    - I.e., in a controlled setting
  - In the field
    - I.e., in "natural", uncontrolled settings
Where? (Cntd Encore)

- **Regression**
  - A bit of a funny one
  - **Backward looking and change oriented**
    - Ensure a change *hasn't broken anything*
    - Esp previous fixes.
Why?

- Three big reasons
  1. **Verification** (or validation)
     - Does the system possess a quality to a certain degree?
  2. **Design**
     - Impose constraints on the design space
       - Both structure and function
  3. **Comprehension**
     - How does the system work?
       - Reverse engineering
     - How do I work with the system?
How?

- **Manual**
  - Typically interactive
    - Human intervention for more than initiation
  - Expectations *flexible*

- **Automated**
  - The test executes and evaluates on initiation
  - Automatically run (i.e., continuously)
Program Equivalence
Many Equivalences

- Source code equivalent
  - Character equivalent
  - AST equivalent
  - Non-comment/names AST equivalent
- Translation equivalent
  - E.g., after compilation
- All-behavior equivalent
- Bisimilar
- Functionally equivalent
Functionally Equivalent

Two programs are *functionally equivalent* just in case they implement exactly the same functionality.

- Functionality is typically characterised by "Input-Output" behaviour
  - Internal structure doesn't matter
    - Programming language, algorithm, etc.
    - FEPs can differ "solely" in execution paths
  - There can be *behaviour* differences (e.g., performance!)
- Strong but not maximally strong!
What Behaviour Is "Functionality"?

The *functionality* of a software system is the *required* behaviour of the system.

Not ideal, as non-functional behaviour may be required.

The *functionality* of a software system is the behaviour of the system that performs some user task.

In either cause, the functionality is a *subset* of all behaviour.
Functionality Equivalent (Reprise)

Two programs are **functionally equivalent** just in case they implement exactly the **same functionality**

The **functionality** of a program are those behaviours which performs a user task

Functionality may be changing, unknown, or misunderstood

The set of functionally equivalent programs depends on the **functionality parameter**
FizzBuzz Example (8)

Compare a "normal" FizzBuzz solution with a golf version:
Functionally Equivalent

Given a set of required functionalities $F$, and two systems, $S_1$ & $S_2$, $S_1$ is functionality equivalent (with respect to $F$) to $S_2$ if $S_1$ and $S_2$ enact $F$.

So, if two programs are behaviourly equivalent then they are functionally equivalent.

What happens if $S_1$ and $S_2$ don't quite enact the same $F$?
"Sufficiently" Functionally Equivalent

Given a set of **required functionalities** $F$, and **two systems**, $S_1$ & $S_2$, which enact functionality sets $F_1$ & $F_2$ (respectively), where, $F_1 \neq F_2 \neq F$, $S_1$ is **sufficiently functionally equivalent** to $S_2$ wrt $F$ if $F_1$ and $F_2$ share "enough" of an overlap with $F$.

Obviously, "enough" is a **parameter**!
**wc** Example!

- GNU **wc** has more functionality (and user-notable behaviour) than **miniwc.py**
  - Or other **wcs**!
  - Different flag options, find longest line, etc.
- Some behaviour is user visible but not "functional" (or interesting)
  - **wc --help --version** vs. **wc --version --help**
    - Non-equivalent in GNU **wc**
    - Do we care to preserve this?!
- How about spacing?!
**wc** Example!

- GNU **wc** has more functionality (and user-notable behaviour) than **miniwc.py**
  - Or other **wcs**!
  - Different flag options, find longest line, etc.
- Some behaviour is user visible but not "functional" (or interesting)
  - **wc --help --version** vs. **wc --version --help**
    - Non-equivalent in GNU **wc**
    - Do we care to preserve this?!
- How about spacing?!