MANCHESTER 1824
The University of Manchester

COMP61511 (Fall 2018)

Software Engineering Concepts In Practice

Week 3

Bijan Parsia & Christos Kotselidis

<bijan.parsia, christos.kotselidis@manchester.ac.uk>
(bug reports welcome!)

Reflecting On Personal Qualities

3 Of 5

- We're at **Week 3**!
  - 3/5s done after today!
  - **Where** are you?
    - And where are you **going**?
    - What are the **next steps**?
- This is a good time to **reflect**

Reflection

Reflection *is the process of examining one own thoughts, beliefs, experiences, concepts, etc. in order to gain self-* ***knowledge*** *and* ***insight***

- Refection doesn't need to be **judgemental**
  - You aren't looking for **flaws**
  - You are trying to **understand yourself**
  - This includes **good** things!
    - Maybe they can get **better**

Reflection Example 1

- For CW1 some people handed in
  - a `rar` archive (e.g., `mbassbp2_cw1.rar`)
- This is in spite
  - my mentioning it in lecture
  - it being described in the assignment
  - Lab0 and Lab1 having the same requirment
    - And being graded and returned before CW1 was due! *What went wrong with their process? How to fix it?*

Reflection Example 2

- Last Thurs afternoon
  - several people asserted that their code 100% should pass some simple tests
  - and that their code review found "no bugs"
- For some, I found they **could not pass any test**
- For others, I looked at the source code and immediately spied 2-3 "obvious" bugs
- What went wrong?
  - And **how do we fix it**

Reflection Example 3

- Look on the **discussion board**!
  - A lot of my replies are along the lines:
    - "On the one hand, you have what `wc` does. On the other, you have a idea of how to do it differently. Which do you think should win?"
  - And that was **enough**!
    - How to get that **without** an **external prompt**?

Reflection Example 4

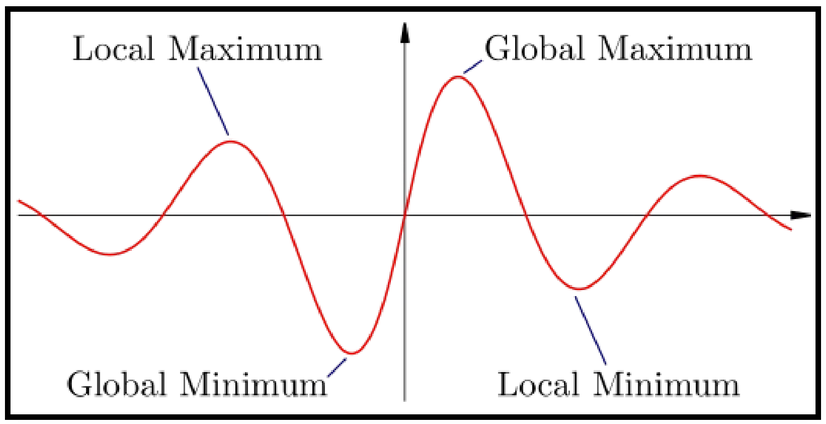
- People asked:
  - I fixed my miniwc.py **will you retest it** to check that it's right
  - I want to know **that my interpretation is right**
  - I just want to **confirm my understanding** so I **don't lose points**
- Reflect!
  - Do we **understand** why people asked these?
  - What's the **right response**?
  - What's would be a **problem** with answering these directly?

Are points the point of the coursework?

Reflection Example 4

- Look on the **discussion board**!
  - A lot of people **got stuck**
    - **How** and **why**?
    - Did **you** get stuck?
    - Did you get **unstuck**?
      - How?

Stuck?


Local Maximum
Global Maximum
Global Minimum
Local Minimum

Bijanic Reflection!

- How'd I do?
  - Not so bad, actually!
- Are my **goals** correct?
- Are my **methods** effective?
  - What would **prove me wrong**?
- Is it **worth the effort**?
  - I'm pretty tired!

Metacognition

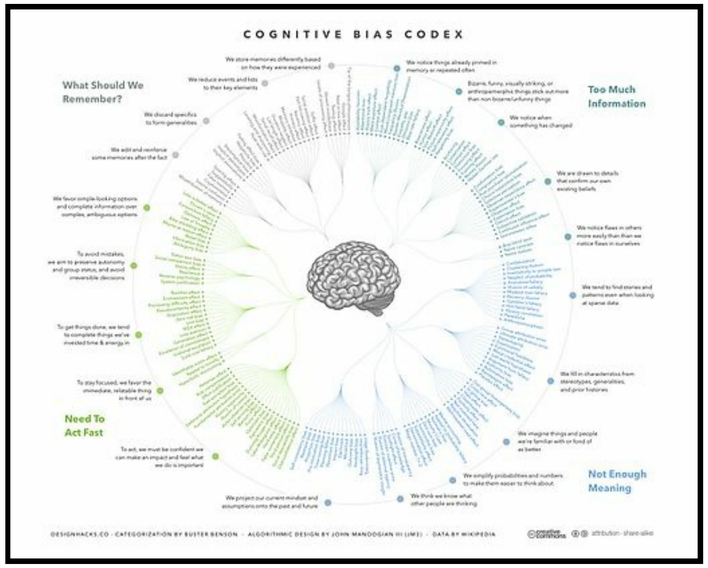
Metacognition *is thinking* ***about thinking***

- **Reflection** is one example
- In general, an important skill
  - For example, when you are **stuck** on a problem
    - it helps to check whether you are in a **rut**
    - that is, just **trying the same thing over and over**
  - Being aware that you got stuck can help you get unstuck!

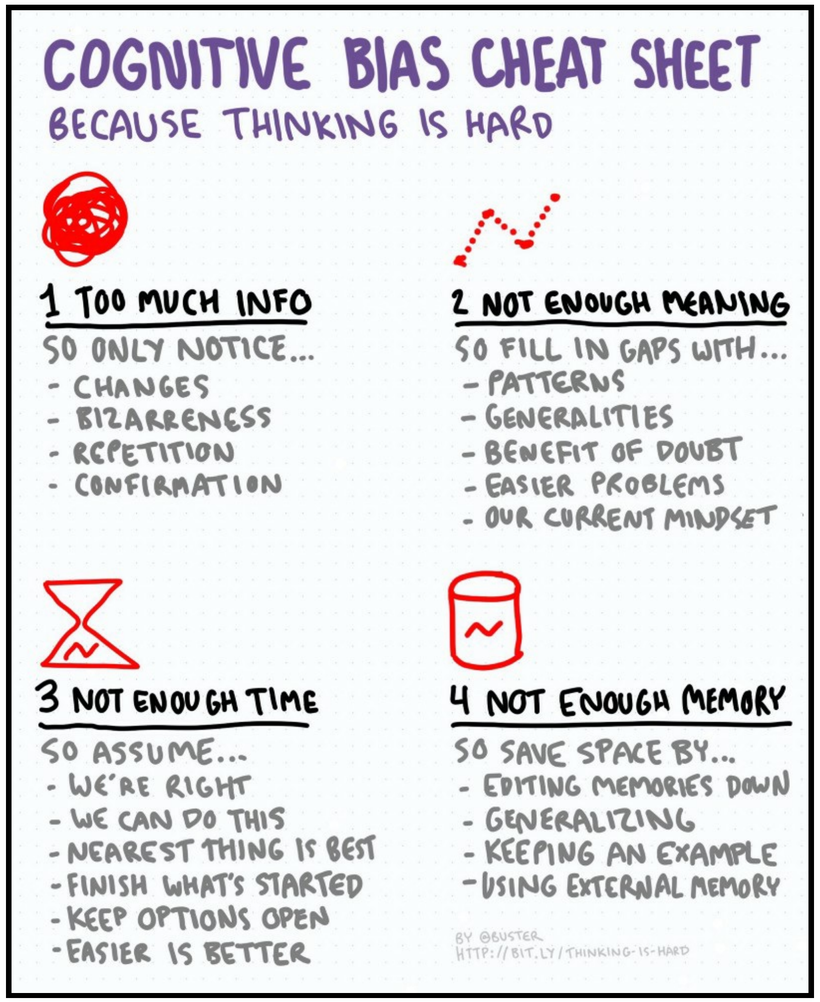
Cognitive Biases

*A **cognitive bias** is a **systematic** departure from rationality.*

- We all have them, and lots of them:



Cognitive Biases



Self-Efficacy

*Perceived **self-efficacy** is defined as people's beliefs about their capabilities to produce designated levels of performance that exercise influence over events that affect their lives.*

- Too **much** self-efficacy
  - is **overconfidence**
  - is related to Dunning-Krugar
- Too **little** self-efficacy
  - **paralyses** you
  - leads to **under**achievement

Two Key Biases

- Bias-Blind Spot
  - "The tendency to see oneself as less biased than other people, or to be able to identify more cognitive biases in others than in oneself."
- Dunning-Krugar Effect
  - "The tendency for unskilled individuals to overestimate their own ability and the tendency for experts to underestimate their own ability."

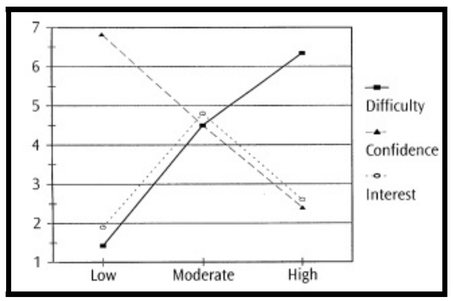
Be very careful here!

Goldilocks Self-Efficacy

- Both **too much** and **too little** are bad!
  - Too much == **bored**
  - Too little == **daunted** and uninterested

Aim for the sweet spot!

Goldilocks Science!



Awesomest paper title: "Self-efficacy and interest: Experimental studies of optimal incompetence"

Trajectory!

- Trajectory over current level
  - Current level is **static**
  - It **informs** trajectory
    - But doesn't **determine** it
- Reflection!
  - Are you learning **quickly** or **slowly**
  - Are you learning **how to learn**

A Goal

A student who has mastered the [Core Body of Knowledge (CBOK) will be able to develop ***a modest-sized software system*** *of a few thousand lines of code from scratch, be able to* ***modify a pre-existing large-scale software system*** *exceeding 1,000,000 lines of code, and be able to* ***integrate third-party components that are themselves thousands of lines of code.*** *Development and modification include analysis, design, and verification, and should yield high-quality artefacts, including the final software product.*

IEEE & ACM ACM 2009 Software Engineering Curriculum Recommendations.

A Student Will...

- **be able to develop**
  - a **modest-sized** software system
    - of a **few thousand lines of code from scratch,**
  - be able to **modify a...large-scale software system**
    - exceeding **1,000,000** lines of code,
    - and be able to **integrate (1000s LOC) third-party components**
  - Development and modification include
    - analysis, design, and verification, and
    - should **yield high-quality artefacts**,
    - **including** the final software product.

`wc`?

- Where does `wc.py` get us?
  - For a **proper clone**
    - ≈ hundreds of LOC
    - OTOH, maybe under 100! `wc` golf?
  - **With extensions**
    - maybe 1000s?
  - Not counting **infrastructure**
    - Tests, etc.
- Does 100s predict 1000s?
  - Good question!

Look Around!

- **Modest size** software systems?
  - What do they **look like**?
  - What do they **do**?
  - Collect some **examples**!
- Remember **reverse engineering**
  - **Port** from a different language!
  - **Rewrite** from scratch
- Create something new!

Some Internal Qualities

Software Quality Landscape

20.1. Characteristics of Software Quality, Code Complete

External Qualities	
Functional Correctness Accuracy Adaptability	**Non-Functional** Usability Efficiency Reliability Integrity Robustness
For Modification Maintainability Flexibility Portability Reusability	**For Comprehension** Readability Understandability
Testability	
Internal Qualities	

Thus Far We Looked At...

- External
  - Functional
    - Correctness (**the** functional quality)
  - Non-functional (a bit)
    - Efficiency (**the** non-functional quality)
- Now, some **internal**
  - Testability
  - For Modification
    - Maintainability

Internal: For Modification

- **Maintainability**
  - ease to "change or add capabilities, improve performance, or correct defects"
- **Flexibility**
  - ease to modify for new situations ("internal" version of **adaptability**)
- **Portability**
  - ease to modify for new environments
- **Reusability**
  - ease to extract parts for use in other systems

Internal: For Comprehension

- **Readability**
  - ease of comprehending the source code, esp at the statement level
- **Understandability**
  - ease of comprehending the software system as a whole
    - from the **synoptic** ("bird's eye") view
    - to the **myopic** ("worm's eye") view

Readability is part of understandability. But you can have readable methods or functions and an impossible to grasp architecture.

Internal: Testability

- A critical property!
  - Relative to a **target quality**
    - A system could be
      - highly testable for **correctenss**
      - lowly testable for **efficiency**
  - Partly determined by test infrastructure
    - Having **great hooks** for tests pointless without **tests**
- Practically speaking
  - Low testability blocks **knowing** qualities
  - Test-based evidence is essential

Problem Indicators

- **Code Smell**
  - "a **surface indication** that usually corresponds to a deeper problem" (Kent Beck via Martin Fowler)
  - **Quick** to spot (if you have experience)
  - Doesn't **always** correspond to a problem
  - Somewhat **subjective**
  - The "WTF test"
- **Pain Points**
  - A part of the system that recurrently causes problems
    - Hard to use
    - Revist often

Testability Smell

```
def get_file_list():
   # Get list of arguments from the command line, minus "wc.py"
   args_list = sys.argv[1:]
   ...
```

or

```
def get_max_width():
   max_val_list = []

   for rec in file_log:
       max_val_list.append(rec.get_max_value())

   return max(max_val_list)
```

Thanks to the brave student who volunteered their code!

Testability Smell FIXED

```
def get_file_list(**args**):
    # Get list of arguments from the command line, minus "wc.py"
    args_list = args[1:]
    ...
```

so we can test by:

```
>>> import wc
>>> wc.get_file_list(['wc.py', '-l', 'filename.txt'])
```

Thanks to the brave student who volunteered their code!

Testability Smell 2

What about:

```
#all the module's code
wc()
```

Thanks to the brave student who volunteered their code!

Testability Smell 2 FIXED

We want to import the module without running anything!

```
#all the module's code
if __name__ = "__main__":
    wc()
```

Now, `import wc` doesn't **run** `wc()`

Thanks to the brave student who volunteered their code!

Refactoring

- Notice
  - None of these moves **changed functionality**
    - Or pretty much any external quality
  - But we **improved**
    - testability
    - maybe readability and maintainability
    - reusability!
- We **refactored** the code

Why Did These Things Happen?

- Haste
- Lack of understanding
  - Of the problem
  - Of the tools
    - E.g., Python
- Someone Else's Code
- Just an accident!
  - Seemed like a good idea at the time!

A Consequence

- Remember the Testable Function's tests

```
>>> import wc
>>> wc.get_file_list(['wc.py', '-l', 'filename.txt'])
```

- What do tests look like for:

```
def get_file_list():
    # Get list of arguments from the command line, minus "
    args_list = sys.argv[1:]
    ...
```

Test Consequences

- Maybe

```
>>> import subprocess
>>> subprocess.check_output('python wc -l filename.
```

- or

```
>>> import sys, wc
>>> sys.argv = ['wc.py', '-l', 'filename.txt']
>>> wc.get_file_list()
```

Are these easy to maintain?

Technical Debt

- Our lack of testability is like a **debt**
  - We might have to **pay it off** in the future
    - By refactoring
  - If we **wait** it **incurs interest**
    - The **more tests** we write, the more tests we have to **change later**
    - Our current tests are
      - Hard to write
      - Hard to read
      - Maybe buggy!
- Debt breeds more debt!

A Bit On Git

Slides

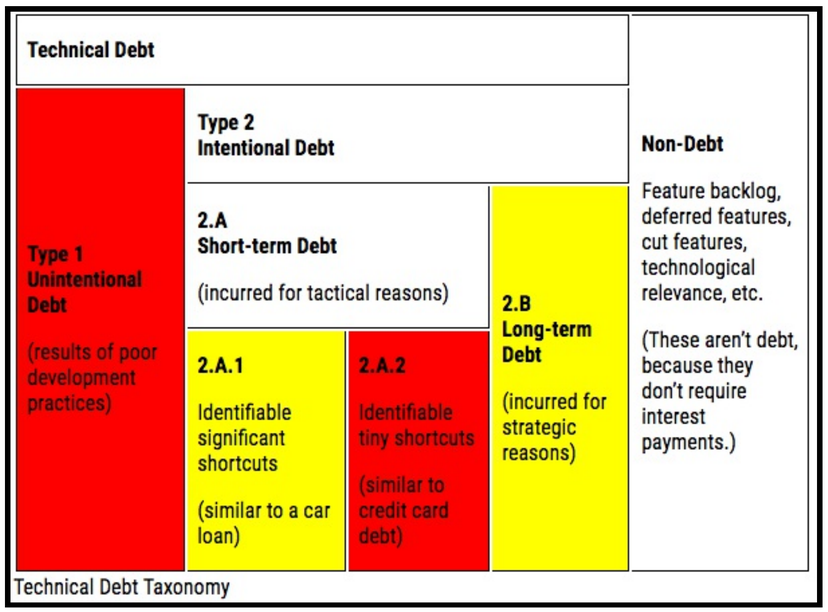
Technical Debt

Technical Debt

Technical debt *is "the obligations incurred by a software organization when it chooses an expedient design or construction approach that increases complexity and is more costly in the long term."*

- Typically, **lower (internal) quality level**
- It may **buy** an **external** quality **effect**
  - More **functionality** (correctness)
  - More **efficiency**
- It may have **negative** external effects
- It may just buy **project** effects
  - E.g., developer effort

Debt Taxonomy



Technical Debt Taxonomy

Intentional Vs Unintentional Debt

- **Unintentional** debt == accidental or incidental
  - We might not **know** we incurred it!
  - We might not know the **interest**!
  - Results of **poor practice**
    - See the testability issues from this morning!
- Intentional debt == deliberate, knowingly incurred
  - Needs an **identifiable rationale**
    - With a **scope**

If you don't know the scope, it's probably not (fully) intentional

Why Go Into Debt?

- 2.A **Short-Term** Debt
  - **Tactical** reasons
  - 2.A.1 "Big" Debt
    - **Significant** shortcuts
  - 2.A.2 "Little" (individual) Debt
    - **Tiny** shortcuts
- 2.B **Long-Term** Debt
  - **Strategic** reasons

Paying Down Debt

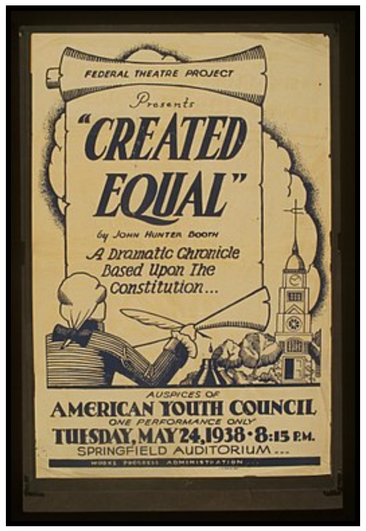
- Debt can become **unmanageable**
  - Even **manageable** debt can be **costly**
- Paying down debt **costs**
  - Debt **shifts** costs to the **future**
    - (But might **add** some costs now)
- **Refactoring** is the usual approach
  - But also things like **adding tests**

Do you always have to pay down your debt?

Good Debt Vs. Bad Debt

- **Good** debt
  - Has a clear **benefit**
  - Is **worth** the cost
  - Is **manageable**
- **Bad** debt
  - Skewed **cost/benefit ratio**
  - Less or un-manageable
- Debt can **"spoil"**
  - Too much **good** debt can become **bad**

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

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

```python
# Typical (8 lines, 196 chars)
for i in range(1, 101):
    if i % 3 == 0 and i % 5 == 0:
        print('FizzBuzz')
    elif i % 3 == 0:
        print('Fizz')
    elif i % 5 == 0:
        print('Buzz')
    else:
        print(i)

# Golf version (wrapped for easier viewing).
# (0 lines, 81 characters)
print ('\n'.join("Fizz"*(i%3==0)+"Buzz"*(i%5==0)
                 or str(i) for i in range(1,101)))
```

Functionally Equivalent

Given a set of ***required functionalities*** *F, and* ***two systems****, S1 & S2, S1 is* ***functionality equivalen****t (with respect to F) to S2 if S1 and S2 enact F.*

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

What happens if S1 and S2 don't **quite** enact the same F?

”Sufficiently” Functionally Equivalent

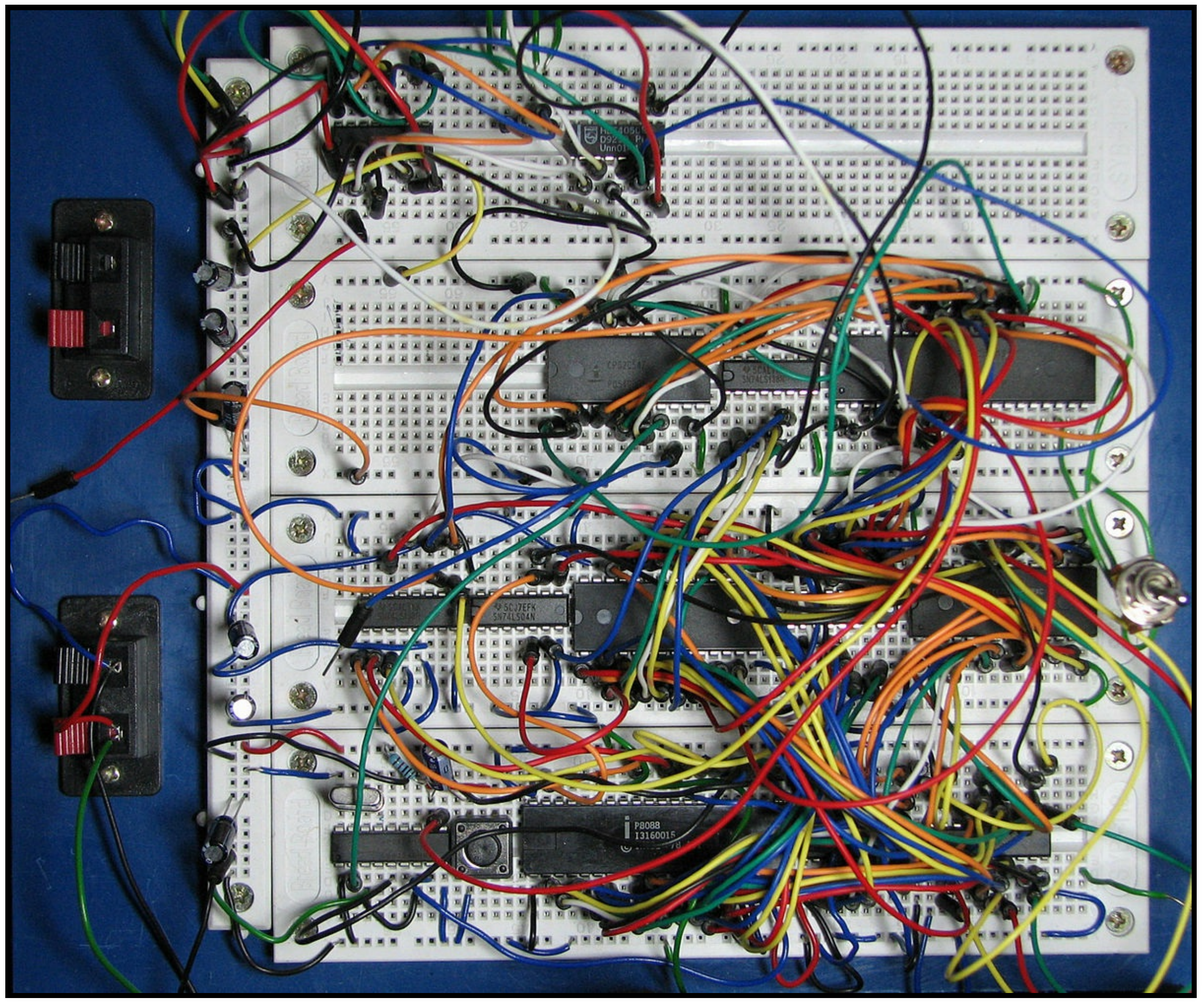
Given a set of ***required functionalities*** *F, and* ***two systems****, S1 & S2, which enact functionality sets F1 & F2 (respectively), where, F1 ≠ F2 ≠ F, S1 is* ***sufficiently functionally equivalent*** *to S2 wrt F if F1 and F2 share* ***"enough"*** *of an overlap with F.*

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

`wc` Example!

- GNU `wc` has more functionality (and user-notable behaviour) than `miniwc.py`
  - Or other `wc`s!
  - 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?!

Refactoring



What Is Refactoring

> ***Refactoring*** *is a* ***transformation*** *of code into* ***sufficiently functionally equivalent*** *code that has* ***"better"*** *internal properties.*

"Martin Fowler defines as "a change made to the internal structure of the software to make it easier to understand and cheaper to modify without changing its observable behavior" (Fowler 1999)" — McConnell, 24.2

- "Sufficiently functionally equivalent"
  - **User observable/desirable** behaviour is preserved
  - Up to some point

Examples

- For example, a **monolithic script**
  - has **low** testability (only system tests!)
  - replace it with a **set of functions**
    - e.g., for arg handling, counting, and printing results
  - result: **easy to test** script
- For example, **hard coded** values
  - great for getting going (tech debt!)
  - **refactor** by making them **configurable**
    - easier to tweak or eventually **make a parameter**
  - result: more **flexibility**!

Code Smells

- Problem **signs** (select sample, McConnell 24.2)
  - Code is **duplicated.**
  - A routine is **too long**.
  - A loop is **too long** or **too deeply nested**.
  - Inheritance hierarchies have to be **modified in parallel**.
  - A class **doesn't do very much**.
  - A routine has a **poor name**.
  - **Comments** are used to explain difficult code.
  - A program contains code that **seems like it might be needed someday**.

Known Debt

- Code smells indicate (potentially) unknown debt
- But there's explicit, known debt
  - Hacks done for **time pressure**
  - **Incomplete transitions** from earlier designs
  - **Learning** code
  - Technology **workarounds**
  - Code for **discarded** features
  - **Overengineered** code

What Refactoring Is Not

- Code **creation**
  - Refactoring might **enable** or facilitate new functionality
  - But you shouldn't add **while** refactoring
- Bug **fixing**!
  - Again, may facilitate
  - Refactoring may **reveal** or "fix" bugs
- Performance **tuning**
  - See above
  - Clean code may be faster...or not!
- Design changes or rearchitecting
  - **Prescursor** activity!

Refactoring Preconditions

- Tests, tests, tests
  - Even when applying "automatic" refactorings
  - Remember, **no change in behavior**
    - Up to a point at least!
- For **simple refactorings**
  - use a tool!
    - e.g., renaming a routine
- For **complex refactorings**
  - have a **plan**!
    - and test!

Technical Debt Case Study

Slides