Reflecting On Personal Qualities
This is a good time to reflect.

- What are the next steps?
- Where are you going?
- And where are you?
- 3/5s done after today?
- We're at Week 3!

3 Of 5
Reflection is the process of examining our own thoughts, beliefs, experiences, concepts, etc. in order to gain self-knowledge and insight. It doesn’t need to be judgmental. You aren’t looking for flaws. You are trying to understand yourself. Maybe they can get better. This includes good things!

• Reflection
Reflection Example

For CW1 some people handed in a rar archive (e.g., mbassbp2_cw1.rar) instead of mbassbp2_cw1.zip. Some people didn't "twig" until I called it out again in class. This is in spite of being described in the assignment. Some people didn't get it because of being mentioned in lecture. What should be the takeaway?

- Some people didn't "twig" until I called it out again in class.
- There being a preparation script.
- It being described in the assignment.
- My mentioning it in lecture.
- This is in spite of being described with a name like Bijan_Parsia_cw1.zip.
- A tar archive (e.g., mbassbp2_cw1.tar) for CW1 some people handed in.
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!
A cognitive bias is a systematic departure from rationality. We all have them, and lots of them: cognitive biases.
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
- paralyzes you
- is related to Dunning-Kruger overconfidence
- leads to underachievement

Too little self-efficacy
- is related to Dunning-Kruger underperformance
- leads to underachievement

Perceived self-efficacy is defined as people's
Two Key Biases

"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." - The Bias-Blind Spot

"The tendency for unskilled individuals to overestimate their own ability and the tendency for experts to underestimate their own ability." - The Dunning-Kruger Effect

Be very careful here!
Goldilocks Self-Efficacy

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

Aim for the sweet spot!
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, develop 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 artifacts, including the final software product.
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,
- be able to integrate 1,000,000 LOC third-party components exceeding 1,000,000 lines of code,
- be able to modify a few thousand lines of code from scratch, and
- integrate (1000s LOC) third-party components.

Development and modification include:

- Should yield high-quality artefacts, analyses, design, and verification, and
- Including the final software product.
Where does `wc.py` get us?

For a proper clone
- With extensions
- Not counting infrastructure
- Maybe thousands
- Tests, etc.
- Does 100s predict 1000s?

Good question!

Where does `wc` get us?

`wc`
Look Around!

Modest size software systems?
What do they look like?
What do they do?
Collect some examples!

Rewrite from scratch!
Port from a different language!
Remember reverse engineering!

Create something new!
What do they look like?
What do they do?
Test Coverage(s)
This typically requires coverage at a reasonable level of confidence. Determines some property.

We want a set of tests that determine some property.

E.g., for fine-grained tests, generality is a problem.

Coverage


Bill Sempf
@sempf
Coverage and Requirements

What happens without requirements coverage?

Consider test driven development

- Where tests drive design
- Requirements
- It needs to provide information about all the critical
- For a test suite to support acceptance
- Consider acceptance testing

Consider acceptance testing
Code Coverage

A test case (or suite) covers a line of code

Aim for minimal test suite with full code coverage

Tricky bit typically involves branches

The more branches, the harder to achieve code coverage

See McConnell on "basis" testing

See coverage.py

Code coverage is a minimal sort of completeness

If the running of the test executes the LOC

Code coverage
Input Coverage

- Input spaces are (typically) too large to cover directly
  - E.g., common inputs
- Pure sample probably inadequate
  - E.g., where the bugs are
- We want a biased sample
  - Space too large and uninteresting
- So we need a sample

Hence, attention to boundary cases
  - E.g., where the bugs are

That is, what's likely to be seen

Input Coverage
Situation/Scenario Coverage

• System tests answer to this!

  • Hence alpha plus narrow and wide beta testing
  • Field testing helps
  • Interaction patterns
  • History of use
  • Machine configuration
  • Inputs aren’t everything

Inputs aren’t everything
Testing always has limits

- Tests are incomplete
- Tests are buggy
- Tests are wrong

"Self" testing subject to cognitive biases

- Confirmation bias: We interpret wrongly
- Observer-expectancy effect/experimenter bias: We influence others to interpret incorrectly
- Congruence bias: We look in the wrong place

Self Testing subject to cognitive biases
Developing Test Strategies

Developing Test Strategies

The McConnell basic strategy (22.2) is a good default.

- Situation
  - Individual or team psychology

You may need adjustments based on:

- Review it regularly
- Ad hoc testing rarely works out well
- Have one! However, preliminary

Ad hoc testing rarely works out well.
Developer Test Strategies

McConnell: 22.2 Recommended Approach to Develop

Test the test cases along with the product.

"Use a checklist of the kinds of errors you’ve made on the project to date or have made on previous projects."

"Design the test cases to make sure that the requirements have been implemented."

"Test for each relevant requirement to make sure that the requirements have been implemented."

"Test for each relevant design concern to make sure that the design has been implemented... as early as possible."

"Use basis testing... At a minimum, you should test every line of code."

"Use a checklist of the kinds of errors you’ve made on the project to date or have made on previous projects."

Design the test cases along with the product.
Some Internal Qualities

### Software Quality Landscape

<table>
<thead>
<tr>
<th>Internal Qualities</th>
<th>Functional</th>
<th>Non-Functional</th>
</tr>
</thead>
<tbody>
<tr>
<td>Testability</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reusability</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Portability</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Flexibility</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maintainability</td>
<td></td>
<td></td>
</tr>
<tr>
<td>For Comprehension</td>
<td></td>
<td></td>
</tr>
<tr>
<td>For Modification</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Robustness</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Integrity</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reliability</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Efficiency</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Usability</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Internal Qualities**

**External Qualities**
Thus Far We Looked At...

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

Thus Far We Looked At...
Maintainability
- ease to extract parts for use in other systems
- ease to modify for new situations (internal version of adaptability)
- ease to modify for new environments
- flexibility
- reusability
- portability

Internal: For Modification
- correct defects
- ease to change or add capabilities, improve performance, or
- maintainability
Readability is part of understandability. But you can have readable methods or functions and an impossible to grasp architecture.

- Readability
- Understandability
  - to the myopic ("worm’s eye") view
  - from the synoptic ("bird’s eye") view

Understandability
- Level
- Ease of comprehending the software system as a whole
- Ease of comprehending the source code, especially at the statement level

Readability

Internal: For Comprehension
Test-based evidence is essential

Low testability blocks knowing qualities

Partially speaking

Having great hooks for tests pointless without tests

Partly determined by test infrastructure

Low testable for efficiency

Highly testable for correctness

A system could be

Relative to a target quality

A critical property!

Internal: Testability
Problem Indicators

- "A surface indication that usually corresponds to a deeper problem" (Kent Beck via Martin Fowler)

- "Quick to spot (if you have experience)"

- "Hard to use"

- "Often revisited"

- "Somewhat subjective"

Problem Indicators

- "The WTF test"

- "Doesn’t always correspond to a problem"

- "A part of the system that recurrently causes problems"

- "Somewhat subjective"

Code Smell

Pain Points

- "Problem (Kent Beck via Martin Fowler)"

- "A surface indication that usually corresponds to a deeper problem"
Thanks to the brave student who volunteered their code!

```python
return max(max_val, max_item)
max_val, max_item = append(rec, get_max_value())
for rec in title_list:
    max_val, max_item = [], []
def get_max_value():
    ...
    args_list = sys.argv[1:]  # Get list of arguments from the command line, minus "wc.py"
def get_title_list():
    ...
```

Testability Smell
Thanks to the brave student who volunteered their code!

```
>> get_file_list(["wc.py", ",", ",", "filename.txt"])
>> import wc
```

So we can test by:

```
...

    # Get list of arguments from the command line, minus "wc.py"

def get_file_list(*args,**kwargs):
```

Testability Smell FIXED
Thanks to the brave student who volunteered their code!

What about:

Testability Smell 2

```python
def test_module_code():
    # Test the module's code
```
We want to import the module without running anything!

Testability Smell 2 FIXED

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

Thanks to the brave student who volunteered their code!
We refactored the code
- reusability
- maybe readability and maintainability
- testability
- But we improved
- None of these moves changed functionality

Notice

Refactoring
Refactoring
What Is Refactoring

Refactoring is a transformation of code into “sufficiently functionally equivalent” code that has “better” internal properties. “Sufficiently functionally equivalent” means that user observable/desirable behaviour is preserved up to some point. “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, 242.
Examples

- For example, a monolithic script has low testability (only system tests).
- Replace it with a set of functions.
- For example, a monolithic script is easier to tweak or eventually make a parameter.
- Refactor by making them configurable.

For example, hard coded values are great for getting going (tech debt!).

result: more flexibility

result: easy to test script

result: easier to tweak or eventually make a parameter
Code Smells

- Code is duplicated.
- A routine is too long or too deeply nested.
- A loop is too long or too deeply nested.
- A class doesn’t do very much.
- Inheritance hierarchies have to be modified in parallel.
- A program contains code that seems like it might be needed someday.
- Comments are used to explain difficult code.
- A routine has a poor name.
- A class doesn’t do very much.

Problem signs (select sample, McConnell 24.2)
Known Debt

- Code smells indicate (potentially) unknown debt
- But there's explicit known debt

Overengineered code
- Code for discarded features
- Hacky workarounds
- Incomplete transitions from earlier designs
- Learning code
- Hacks done for time pressure

Known Debt
What Refactoring Is Not

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

Precursor activity
- Design changes or rearchitecting
Refactoring Preconditions

Tests, tests, tests

Even when applying "automatic" refactorings

Remember, no change in behavior

Even when applying "automatic" refactorings

Tests, tests, tests

For complex refactorings

- Rename a routine
- Use a tool!

For simple refactorings

- Up to a point at least

and test!

have a plan!
Technical Debt Revisited
Technical debt (Recall) is "the obligations incurred by a project as a result of choosing an expedient design or construction approach that increases complexity and is more costly in the long term." It may just buy project effects.

- Lower (internal) quality level
- More efficiency
- More functionality (correctness)
- It may have negative external effects

Technical debt can be costly in the long term, especially if not addressed.

E.g., developer effort
Debt Taxonomy

- Non-Debt
- Technical Debt

2.8 Debt Long-Term

2.4.2 Identifiable

2.4.1 Unidentifiable

4.1 Similar to a Car

4.2 Identifiable

(Incurred for Strategic Reasons)

Situations where the impact of these assets is
identified through economic benefits,
financial collapses, etc.

(Incurred for Technical Reasons)

Situations where the impact of these assets is
identified through economic benefits,
financial collapses, etc.

Debt

Debt

Type 1

Type 2

Developed

Debt

Debt

Debt of Poor
If you don’t know the scope, it’s probably not (fully) intentional

- With a scope
  - Needs an identifiable rationale
  - Intentional debt == deliberated, knowingly incurred

- Results of poor practice
  - We might not know the interest
  - We might not know we incurred it

Before we discussed unintentional debt

Intentional Debt
Why Go Into Debt?

**Strategic Reasons**
- 2. B Long-Term Debt
  - Tiny Shortcuts
- 2.A.2 "Little" (individual) Debt
  - Significant Shortcuts
- 2.A.1 "Big" Debt

**Tactical Reasons**
- 2.A Short-Term Debt
Do you always have to pay down your debt?

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

Debt can become unmanageable
Good Debt Vs. Bad Debt

**Good Debt**
- Has a clear benefit
- Is worth the cost
- Is manageable
- Has a clear benefit

**Bad Debt**
- Skewed cost/benefit ratio
- Less or un-manageable
- Debt can "spoil"
- Too much good debt can become bad

Good Debt Vs. Bad Debt
Technical Debt Case Study

Slides
Project Effects On Product Qualities
A Key Point (1)

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 raise 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!
A Key Point Summarised

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

But...pick two:
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

• Does the Good-Fast-Cheap/Pick-2 triangle + the general principle imply that
## Cost of Detection

McConnell, 3.1

<table>
<thead>
<tr>
<th>Time Introduced</th>
<th>Post-Release</th>
<th>Construction</th>
<th>Architectural</th>
<th>Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-25</td>
<td>10</td>
<td>1</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>25-100</td>
<td>15</td>
<td>1</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>100-1000</td>
<td>10</td>
<td>10</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 3.1: Average Cost of Fixing Defects Based on When They’re Introduced and Detected
McConnell, 3.1

Cost of Detection

Figure 3.1. The cost to fix a defect rises dramatically as the time from when it is introduced to when it is detected increases. This remains true in which a defect is detected.
Project Qualities

Per Se

We've only talked about product qualities.

Projects have qualities too!

Let's study them as well.

There is an interaction.

- Using a certain methodology (correctly (or no))
- Being popular
- Being well resourced
- Being well run
- Being on (or off) budget and schedule

- "F.G."