Programming Practices

Overview

- Introduction to Programming Practices
- The Importance of Documentation
- Tech Debt and How to Reduce it

Learning Objectives

- Discover the features of clean, well written code
- Identify why documentation is important
- Explain what refactoring is
- Define what tech debt is and learn how to reduce it
- Discover the importance of code reviews

Writing code like a pro

> Programs must be written for people to read, and only incidentally for machines to execute. -- **Harold Abelson** You know you are working on clean code when each routine you read turns out to be pretty much what you expected. -- **Ward Cunningham**

Harold "Hal" Abelson - MIT Professor and involved in the free software movement

Ward Cunningham - Developed the first Wiki (The software which is the basis of Wikipedia) and co-authored the manifesto for agile software development

Read the quotes and ask the learners what they think about them. Do they know of clean practices?

Clean code is easier to...

- Test
- Refactor
- Extend
- Understand
- Reuse

We will see how all of these apply to clean code throughout the module.

Key Ideas

Functionality: does it work?

Testability: are you sure it works?

Readability: can everyone understand how it works?

Ask the learners, are any more important than the other? Try to generate a discussion.

Bad code tries to do too much; it is often muddled and ambiguous.

Clean code is focused. Each function, class, and module does a single, clear, obvious thing.

The Acronyms of Clean Code

- KISS
- DRY
- YAGNI
- RTFM

Ask the learners if they know any.

What do they mean?

Keep It Simple Stupid. (Obvious code is better than obfuscated code which is abstracted and seems clever, but is hard to follow)

Don't Repeat Yourself. (Don't write code to perform an identical task multiple times, prefer re-use,)

You Ain't Gonna Need It. (Don't add functionality until required, e.g. don't add capability for integrating your app with multiple types of database just in case you need to do that - likelyhood is you probably won't)

Read The F'ng Manual. (Probably best not to swear!) - You can discover a lot about how to best use the tools in your toolbox by checking their docs and looking at examples, rather than guessing at their functionalities

---

Programming Practices

Much like the humble piano player, it takes time to refine our abilities to practice clean code practices.

Code Layout

- Indentation is essential
- Whitespace is free

- Keep your lines short
- Keep related elements together
- Remove dead code **aggressively**

Is this good code?

Go to example code snippets provided in `bad-code.py`.

Ask students how they would improve it.

Distribute good-code.py when done.

Variables

- Avoid global variables
- Use intention-revealing names
- Longer names are not bad if they're meaningful (but don't overdo it)
- If you feel the need to have a really long variable name, your variable is probably trying to do too much.
- Immutable variables make for applications which are easier to follow and test

Immutable meaning variables that don't change value once assigned

- Variables cannot be reassigned in functions when passed into that function as a parameter (which encourages those functions to not side-effect)
- Note that while variables cannot be reassigned, the object that a variable points to can be modified in place for many types (e.g. lists, dictionaries, custom objects) - This may also be best avoided.

Functions

- Keep them small
- Functions should only do **one** thing
- Functions should have no side effects
- Function names should be verbs
- Should not affect the state of the application directly

The less it does, the easier it is to test!

'One thing' means one conceptual task - if a function's name can't succinctly describe what it does then it might be too big

Smaller functions make for more testable chunks of code.

Function Side-Effects

A function has "side effects" if it makes an observable change to the application state other than just providing its return value.

Side effects tend to muddy the waters when it comes to ease of testing and ease of understanding, so they're usually avoided as far as possible.

Necessary Side-Effects

Sometimes, side effects are necessary.

Can we think of any?

- Accepting user input
- Rendering to the screen (or terminal)
- Reading and writing Files
- Database access
- API Calls

A program with zero side effects would be technically useless, as it could neither take input, nor produce output.

Side-effects can be necessary, but by limiting where they happen we can make our application cleaner and more testable.

Documentation

- Include what adds value for understanding
- Code comments are good, but also lean towards self-documenting code
- Include a README. Tell people how to install, set up, use, and (where applicable) contribute code to your application

---

Self-Documenting Code

Is this self-documenting?

```
g = 9.81
t = 5
d = 0.5*g*t**2
```

** is the mathematical power operator

Self-Documenting Code

Better:

```
gravitational_acceleration = 9.81
time = 5
distance = 0.5 * gravitational_acceleration * (time**2)
```

Self-Documenting Code

Even better:

```
def calculateFreeFallDistance(time):
    gravitational_acceleration = 9.81
    distance = 0.5 * gravitational_acceleration * (time**2)
    return distance
```

Quiz Time! 🤓

Which of these statements about variables is false?

1. Global variables should be avoided.
2. Variables should have intention-revealing names.
3. Longer variable names are fine if they're meaningful.
4. Mutable variables make for applications which are easier to follow and test.

Answer: 4

(1) Global variables encourage side-effecting functions.

(4) True for immutable variables, as opposed to mutable.

Which of these statements about functions is false?

1. Functions should be small
2. Functions should only do one thing
3. Functions should have side effects
4. Function names should be verbs

Answer: 3

Functions should NOT have side effects.

Refactoring

When we start working on a new feature and identify issues in the code we're going to be working on, it's a good idea to refactor the code first and develop the feature after.

Good test coverage is essential to refactoring confidently and effectively. Otherwise, welcome to regression hell...

We **refactor** code when we improve the way our code is written without affecting the end result.

Refactoring is a gradual process. Do it as required.

A Short List Of Dangerous Things to Do In Life:

1. Cliff jumping
2. Shark diving
3. Refactoring without good test coverage

Perhaps in programming terms we can think of the first two as: - "Cliff jumping" is diving into a program without a spec - "Shark Diving" is sending your code to senior devs and hoping for the best

---

Tech Debt

Deadlines and priorities will sometimes force us to take shortcuts and implement quick-and-dirty solutions for our problems.

Tech debt reflects the implied cost of additional rework caused by choosing an easy (limited) solution now instead of using a better approach that would take longer.

Every time we take one of these shortcuts, we increase our tech debt.

Why Is Tech Debt Bad?

- Introduces uncertainty
- Balloons time and effort estimates
- Slows down development
- Increases likelihood of regressions

Much like regular debt, tech debt accrues "interest" in the form of slowing down other development.

Tech debt should be recorded as and when it's noticed - unknown or untracked tech debt is even worse for causing unexpected delays.

Causes of Tech Debt

- Lack of documentation
- Delayed refactoring
- Business pressures causing rush
- Lack of technical understanding
- Lack of collaboration - knowledge isn't being shared around
- Many many more! https://en.wikipedia.org/wiki/Technical_debt#Causes

---

Ensuring Code Quality

Coding Standards

How do you enforce coding standards?

- Team consensus
- Document them (or follow a public style guide)
- Code reviews
- Linters
- Consistency, consistency, CONSISTENCY!

https://pylint.pycqa.org/en/latest/user_guide/options.html

Linters can check for variable and function naming standards (snake_case, camelCase etc) correct formating, indentation and whitespacing, enforce documentation comments on functions and more

---

Code Reviews

- Reviewing some code written by a peer to ensure it meets the team's quality standards
- Concentrate on the important stuff, let the tools pick up on the minutiae
- Does the code appear to do what it's supposed to do? Code structure? Is the code readable? Are the tests good? Is the test coverage good?

Quiz Time!

What is refactoring?

1. Improving the way our code is written which affects the end result.
2. Improving the way our code is written without affecting the end result.
3. Improving our ability to ensure tech debt is kept to a minimum.
4. The implied cost of additional rework.

Answer: 2

Which comment is best?

```
# comment to go here
date_config = get_date_config(event)
for target in targets:
    target['date'] = date_config
```

1. Update targets.
2. Attach the date configuration to each target.
3. Here we are getting the date config for a particular event. As we iterate through each target, we assign a key-value pair, with `date` being the key, and the date config being the value.
4. None. It's self-documenting.

Answer: `2`

3 is way too verbose and descriptive. We should be able to understand what is happening in comment 2, which is more concise.

Self-documenting code is better than code littered with comments, and that adding a comment should only be done when you're needing to warn other developers about something out of the ordinary, or contextualise an unusual decision.

Learning Objectives Revisited

- Discover the features of clean, well written code
- Identify why documentation is important
- Explain what refactoring is
- Define what tech debt is and learn how to reduce it
- Discover the importance of code reviews

Terms and Definitions Recap

Documentation: Any communicable material that is used to describe, explain or instruct.

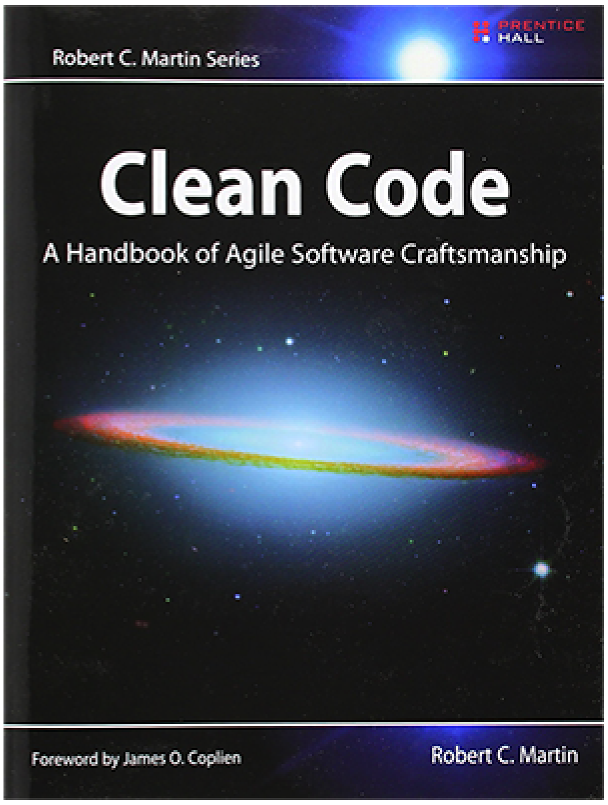
Tech Debt: Reflects the implied cost of additional rework caused by choosing an easy (limited) solution now instead of using a better approach that would take longer.

Terms and Definitions Recap

Static Code Analysis: The analysis of computer software that is performed without actually executing programs.

Code Review: Reviewing some code written by a peer to ensure it meets the team's quality standards.

Further Reading


Robert C. Martin Series
PRENTICE HALL
Clean Code
A Handbook of Agile Software Craftsmanship
Foreword by James O. Coplien
Robert C. Martin