Module 9 - Legacy Code, Refactoring

Topics covered:

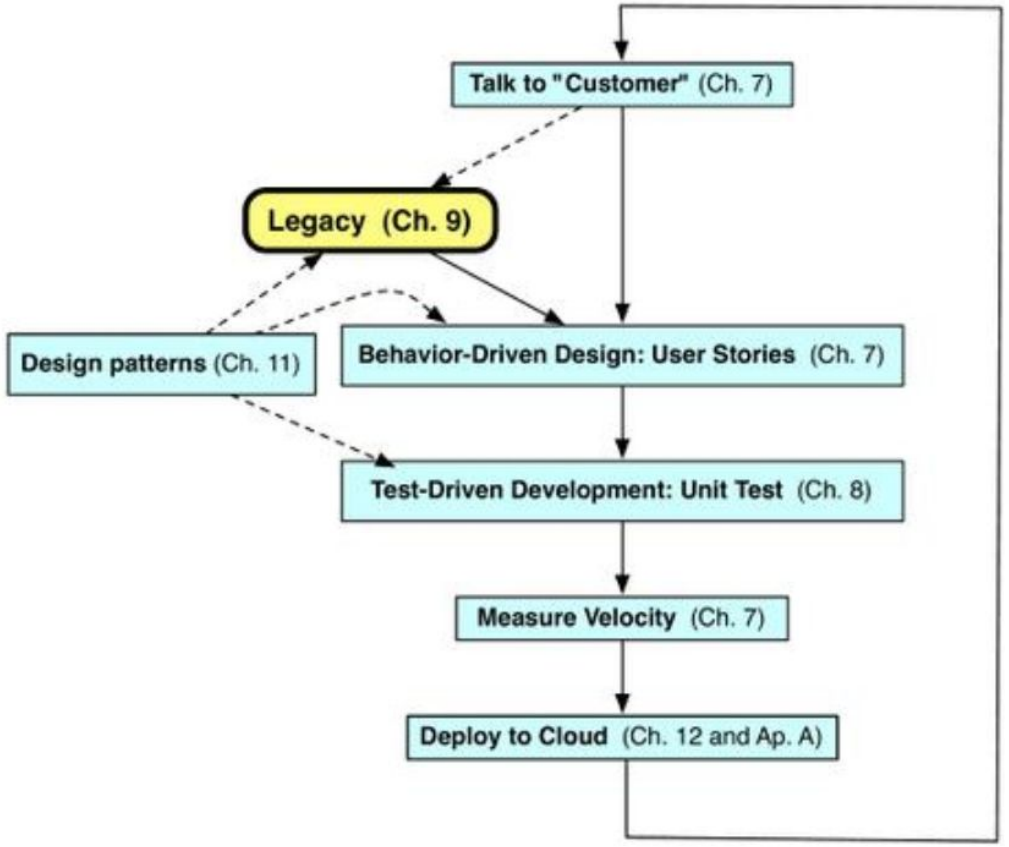
- Introduction to Module 9: Legacy Code, Refactoring
- Practice questions for module 9


WE DON'T KNOW WHAT
THIS DOES BUT THE APPLICATION
CRASHES IF YOU REMOVE IT
imgflip.com


WE JUST NEED YOU TO ADD THIS
LITTLE FEATURE TO LEGACY
made on imgur

Legacy Code & Agile

- **Legacy code** is hardly understood code whose original designers are gone and many **patches** are added that aren't explained by **design documents.**
- **Legacy code** stays in the project because it still meets customer need even if its design & implementation may be outdated or poorly understood.
- **Agile** techniques can be used to **enhance and modify legacy code**.
- Legacy's position in the Agile lifecycle ⇒
- One of the biggest difficulties of working with **legacy code** is **maintainability** challenges.



Legacy Code >> Maintainability >> Refactoring

- **Maintainability** is the ease with which a product can be improved, adapted to operating environment, repaired or improved to facilitate future maintenance.
  - **Corrective maintenance:** repairing defects and bugs,
  - **Perfective maintenance:** expand functionality for new customer requirements,
  - **Adaptive maintenance:** coping with a changing operational environment,
  - **Preventative maintenance:** improving software structure to increase future maintainability
- Key component of maintenance activities is **refactoring.**
- **Refactoring** is a process that improves the structure of code without touching code's functionality. Continuous **refactoring** improves functionality

Legacy Code >> Formal & Informal Design Documents

- **Formal design documents** are written by the creators of the code
- **Informal design documents** include:
  - Unit, functional and integration tests
  - Lo-fi mockups and cucumber style user stories
  - Photos of sketches about application architecture, class relationships,
  - Git commit log messages, comments, rdoc style documentation embedded int he code
  - Archived email, blog notes like campfire or basecamp.
- **Agile** suggests placing more value on documentation that’s closer to the working code.

Enhancing Legacy Code

- There are two ways to make changes to existing software:
  - **Edit and Pray:** familiarize yourself with some small part of the code where you need to make your changes, edit the code, manually look around to see if you broke anything, deploy and hope for the best.(Like name suggest, not so reliable)
  - **Cover and Modify:** create tests if they don't already exist that cover the code you want to modify and use them to detect unintended behavioral changes caused by your modifications.
- **Legacy Code** often lacks sufficient tests to modify with confidence, regardless of who wrote it and when.

Agile Techniques for Legacy Code

- Embracing change over long timescales
- IF well-structures software with good testing: BDD and TDD to drive attention to functionality using small steps.
- IF not, we need to bootstrap into desired situation using change points, characterization tests, and refactoring.
  - **Change points:** where you can possibly make changes in the code. Identifying them is first step (Done during **Code Exploration**).
  - **Characterization tests:** capture how code works to establish baseline behavior before any changes.
  - **Refactoring:** makes existing code more testable or accommodating to new changes
  - **Testing:** Regression test your implementation and then possibly add new test specific to your implementation

Exploring Legacy Code

- Legacy code **exploration** aims to understand how the code works from different points of view including customers, designers and developers.
  - **Customers**: request changes
  - **Designers & Developers**: created the original code
- Exploration can be aided by reading tests, design documents, inspecting the code, drawing of generating UML diagrams to identify relationships among important entities.
- Once successfully seen the app demonstrated in production, next steps are to get the app running in development by either cloning of ficturing the database to get the test suite running in development.
- This is where you identify the change points, you usually don’t write any code at this stage

Characterization Tests

- Tests **written after the fact** that **capture and describe the actual current behavior** of a piece of software, even if that behavior has defects and bugs.
- We add characterization tests when we adapt the **Cover and Modify** method but we lack tests to capture how the code works.
- **Easier to start with integration level characterization test** such as cucumber scenarios which puts less focus on how the app works and focuses on user experience, and externally visible app behavior. It is fine to start with imperative scenarios since the goal is to increase coverage and provide ground truth.
- To **create unit and functional level characterization tests** for code we don't fully understand, we can write a spec that asserts an incorrect result, fix the assertion based on the error message and repeat until we have sufficient coverage.

In line Comments

- Not only does **legacy code** lacks tests and good documentation, but it also **often has missing or inconsistent comments**.
- Comment are **best written at the same times as the code** instead of as an afterthought.
- Comments **shouldn't repeat what is obvious from the code** such as explaining the implementation or why a certain implementation is chosen.
- Comments **should raise the level of abstraction from the code**.

```
#This code sums up two integers and returns the result
def sum(a, b):
    a + b
end
```

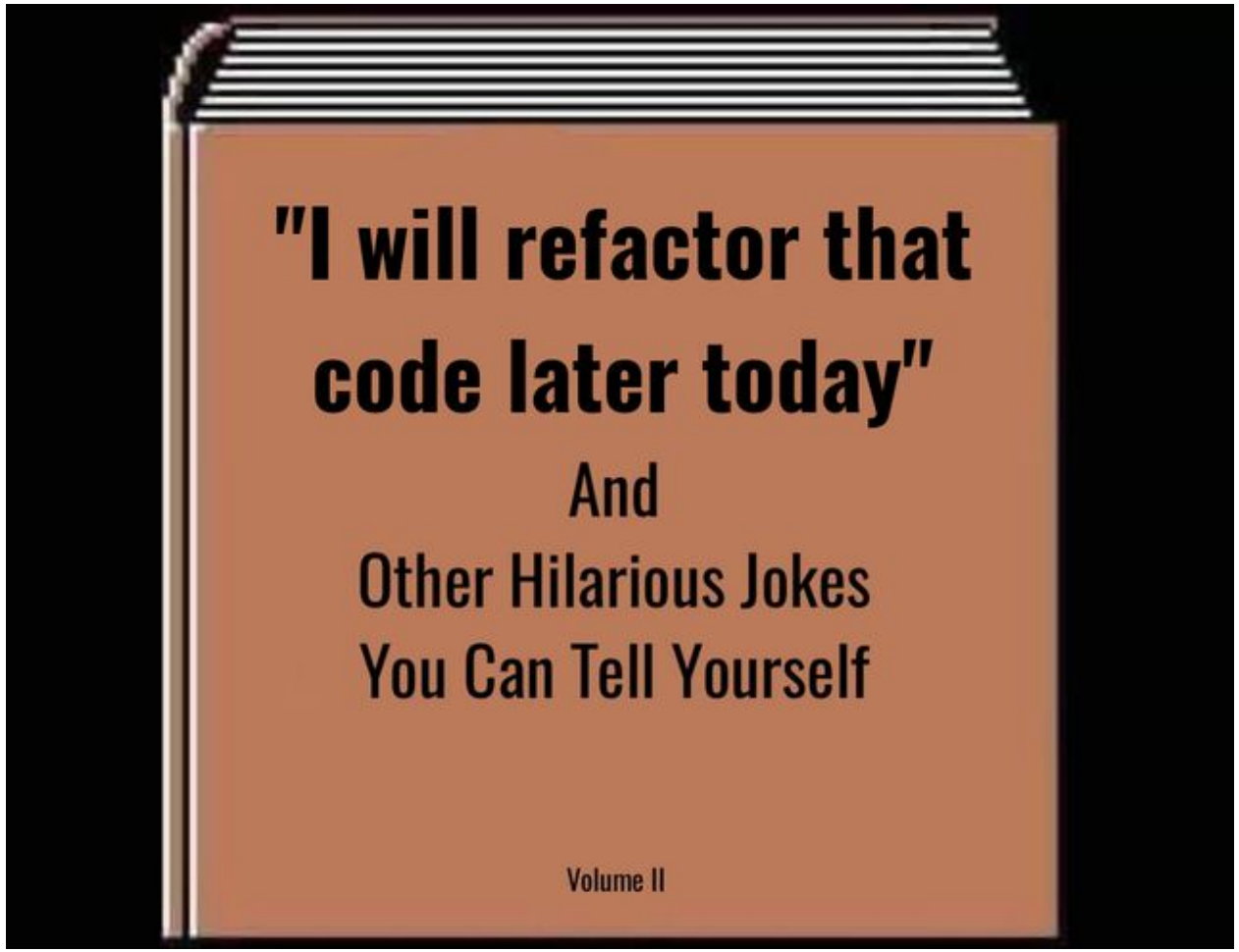
Is this a good comment?

Metrics, Code Smells and SOFA

- Software engineering is about creating ***beautiful*** working code.
- **Software metrics** provide **quantitative measure of code quality.** No consensus on which metrics are most important.
  - **Cyclomatic complexity** and **ABC score** can be used to guide towards code that is in particular need of attention
  - **C0 coverage:** low coverage identifies undertested code
- **Code smells** provide qualitative but specific descriptions of problems that make code hard to read. There are over 60 specific code smells that have been identified.
- **SOFA:** short, do one thing, few arguments, maintain single level of abstraction
- SOFA acronym is for desirable properties of a method.

Refactoring

- A **refactoring** is a **particular transformation of a piece of code**, including a name, description of when to use the refactoring and what it does, and detailed sequence of mechanical steps to only apply the refactoring.
- Effective refactorings **improve software metrics, and/or eliminate code smells.**
- Most refactorings inevitably **cause some existing tests to fail** (or the code is undertested)
- A key goal is to **minimize the amount of time until those tests are modified again to make the tests pass.**
- Refactoring may result in recursively having to apply simpler refactorings first.



Agile as a Maintenance Process

- **Change is the norm**, new iterations of products are routinely deployed as new releases
- You are in **continuous contact with the customer**
- **Regression testing** and **refactoring** are standard in Agile process.
- **Plan and Document lifecycle:**
  - **Maintenance managers:** play as project managers, interface with the customer, make cost and schedule estimates, documents the maintenance plan, and manage the maintenance engineers
  - **Change Control Committee:** truages the change requests from customers and stakeholders
  - Regression Testing: ensures new features do not interfere with the old ones, and has a big role in maintenance
  - Refactoring: there's less refactoring in Plan-and-Document process during product development than in Agile development.
  - Reengineering: is an alternative to starting over when the code becomes increasingly difficult to maintain.
- Two thirds of the cost of product are in the maintenance phase in Agile development, so why not use the same maintenance compatible software development process for the whole lifecycle

Recap:

- **Agile** development is a good approach to both maintain software and to enhance legacy code
- **Refactoring** is needed for development to keep code maintainable. Refactoring while transforming the code improves software metrics and eliminate code smells.
- While writing code **software metrics** and **code smells** can identify hard-to-read code.
- Methods should be **SOFA:** short, do one thing, have few arguments, and maintain single level of abstraction
- Without good test coverage, we lack confidence that refactoring or enhancing the code will preserve existing behavior.
- Therefore we define **legacy code** as **“code without tests” and create characterization tests where necessary to beef up test coverage before refactoring and enhancing legacy code.**

PRACTICE QUESTIONS

We will use the remainder of the discussion to go over some practice questions!

Question 1

Which of the following is true regarding legacy code bases? [Select all that apply]

A. It contains deprecated and un-functional code
B. It reeks of code smells
C. It lacks in test coverage and documentation
D. It will have a relatively high maintenance cost
E. Proper refactoring will enhance its maintainability as well as readability

Question 1

Which of the following is true regarding legacy code bases? [Select all that apply]

A. It contains deprecated and un-functional code
B. It reeks of code smells
C. It lacks in test coverage and documentation
D. It will have a relatively high maintenance cost
E. Proper refactoring will enhance its maintainability as well as readability

Question 2

Which of the following are properties of good comments? (Select all that apply)

A. They must be grammatically accurate
B. They don’t repeat what is obvious from the code
C. They are expressed at a higher level of abstraction than the code
D. They express relevant input/output data types, variable names and implementation details
E. They are written by the original developers

Question 2

Which of the following are properties of good comments? (Select all that apply)

A. They must be grammatically accurate
B. They don’t repeat what is obvious from the code
C. They are expressed at a higher level of abstraction than the code
D. They express relevant input/output data types, variable names and implementation details
E. They are written by the original developers

Question 3

The cost of software maintenance usually exceeds the cost of software development

A. True
B. False

Question 3

The cost of software maintenance usually exceeds the cost of software development

A. True
B. False

Question 4: What is wrong here?

https://gist.github.com/b44d2059ebcccd4a1d45111884ba350b

```
1  # WARNING! This code has a bug! See text!
2  class TimeSetter
3    def self.convert(d)
4      y = 1980
5      while (d > 365) do
6        if (y % 400 == 0 ||
7            (y % 4 == 0 && y % 100 != 0))
8          if (d > 366)
9            d -= 366
10           y += 1
11         end
12       else
13         d -= 365
14         y += 1
15       end
16     end
17     return y
18   end
19 end
```

Attendance

https://tinyurl.com/discussion-week-10

Passcode: **legacy**