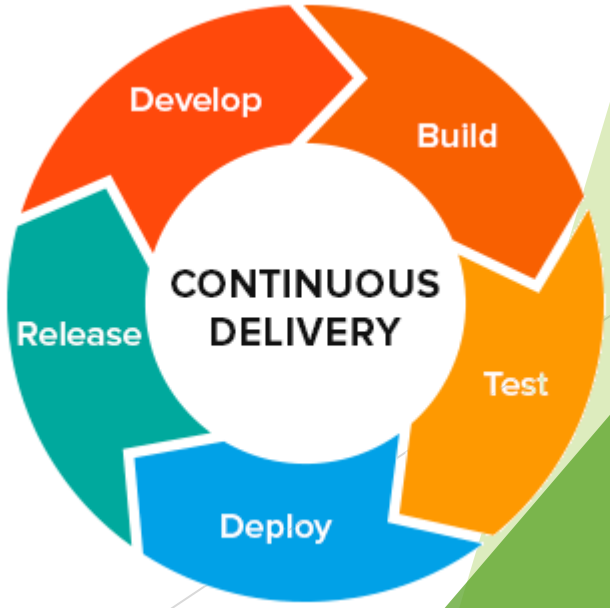
Lecture 10 Supplement: Behavior-Driven Development (BDD)

Wonsun Ahn

Agile Software Development: Quick adaptation to changing requirements

- Agile: A development process amenable to frequent changes in requirements
  - Stresses adapting to user needs quickly vs. negotiating a contract
  - Stresses efficient communication vs. comprehensive specification
  - Stresses iterative design vs. rigid plan
- Some Agile practices:
  - Continuous Delivery (CD):
    Frequent delivery of software for user feedback
  - Test Driven Development (TDD):
    Allows continuous delivery through continuous testing
  - Behavior Driven Development (BDD):
    A type of TDD better suited for adapting to user needs



TDD Strength:
Coding is driven by requirements

- Requirements drives → Testing drives → Development
  - Test cases are written based on the requirements
  - Code is written to fulfill the test cases

- End result:
  1. Ensures all code adheres to requirements at all times ☺
  2. Ensures all coding effort is focused on fulfilling requirements ☺

- What's not to like?

TDD Weak Link 1:
Requirements can become a burden

- A typical Software Requirements Specification (SRS):

The rent-a-cat system shall list all cats when command “1” is given such that an empty string is returned when no cats are available, and a full listing of cats is returned when there are cats available.

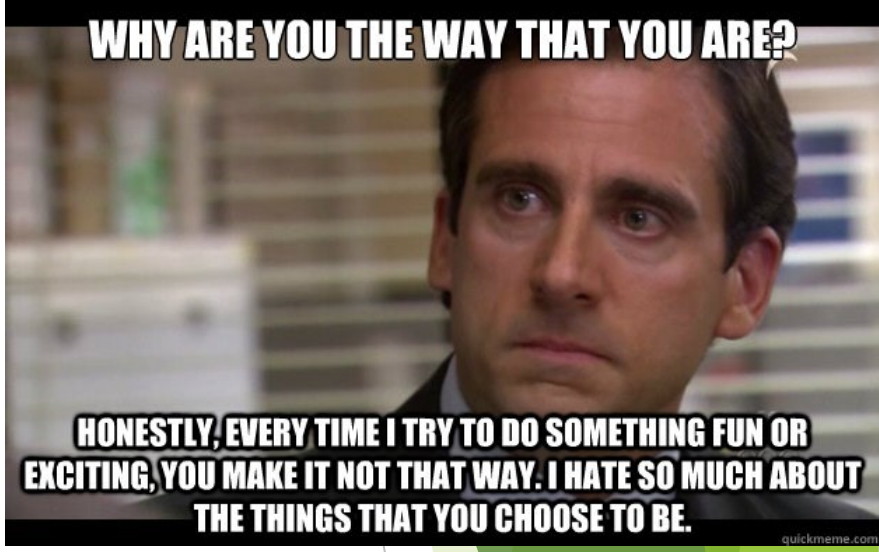
Each line in a listing of cats shall consist of the string “ID” followed by one space (ASCII code 32) followed by the numerical ID of the cat followed by the string “.” followed by the name of the cat followed by a UNIX-style newline ‘\n’ (ASCII code 10).

A cat name shall consist of alpha-numeric characters and spaces (ASCII code 32).

→ Why is it so long? If I’m an end-user, I’m not going to read this. ☹

→ Even if I reads this, I am not going to understand it. ☹

SRS, Why are you like this?



- SRS is not meant to be read by the end-user:
  1. SRS is a contract (why it sounds like legalese) → meant for lawyers
  2. SRS is a specification (why it is so technical) → meant for developers
- Painstakingly compiled by requirement analysts after interviewing users

- But what if requirements must change often (e.g. during prototyping)?
- Now SRS becomes a burden:
  1. End-user must pore through SRS in order to give feedback
  2. Hundreds of pages of SRS documents must be maintained throughout
- SRS should be a tool for communication with user, not documentation!

TDD Weak Link 2:
Testing can become a burden

- If requirements change often, since Requirements → Tests → Code,
  - Both testing infrastructure and code implementation must change

- There is no way to avoid changing code but…
- TDD means a lot of time is also spent updating testing infrastructure
  - Time that would not be spent had we not done TDD
  - That's why maintaining testing infrastructure can feel like extra baggage
  - Testing should improve productivity, not decrease it!

Behavior-Driven Development (BDD): Behavior = Requirement / Test

- Introduced by Dan North in 2006 issue of “Better Software magazine”
  - https://dannorth.net/introducing-bdd/

- Paradigms laid down by Dan North in above article:
  - Software is described in terms of behaviors not code-centric specifications
  - Behaviors are “executable” --- behaviors are directly testable on the code (Behaviors are the requirements *and* the tests at the same time)
  - Behaviors are written in a “ubiquitous language” --- in other words plain English

A Code-Centric Specification: Something that only coders understand

The rent-a-cat system shall list all cats when command “1” is given such that an empty string is returned when no cats are available, and a full listing of cats is returned when there are cats available.

Each line in a listing of cats shall consist of the string “ID” followed by one space (ASCII code 32) followed by the numerical ID of the cat followed by the string “:” followed by the name of the cat followed by a UNIX-style newline ‘\n’ (ASCII code 10).

A cat name shall consist of alpha-numeric characters and spaces (ASCII code 32).

A Behavior Driven Specification: Something that users can understand

```
Rule: When there are cats, the listing is one line per each cat.


Scenario: List available cats with 1 cat

Given a cat with ID 1 and name "Jennyanydots"
When I list the cats
Then the listing is: "ID 1. Jennyanydots\n"

Scenario: List available cats with 2 cats

Given a cat with ID 1 and name "Jennyanydots“
And a cat with ID 2 and name "Old Deuteronomy“
When I list the cats
Then the listing is: "ID 1. Jennyanydots\nID 2. Old Deuteronomy\n"
```

Behavior-Driven Development: Solves existing problems with TDD

1. Requirements can become a burden
   → Behaviors are easily shared and updated among stakeholders
2. Testing can become a burden
   → Since behaviors *are* the tests, tests are updated with requirements

- Now stakeholders become active participants in shaping requirements
  - Agile: software can adapt quickly to changes in user need

- In other aspects, BDD still works like TDD: red-green-refactor loop

Dialect for Describing Behaviors: Gherkin or JBehave

- Dialect for describing behaviors must satisfy two criteria
  1. Must be like plain English so end-users can understand it
  2. Must have some structure so testing behaviors can be automated

- Two popular Domain Specific Languages (DSLs):
  1. Gherkin : Used with the Cucumber testing framework
  2. JBehave : Used with the JBehave testing framework
- We will learn Gherkin with Cucumber but JBehave is almost identical
  - In fact, JBehave framework also has a Gherkin language parser

Gherkin : Domain Specific Language for describing Behaviors

Gherkin hierarchy

- Top Level: Features
  - Feature: a discrete functionality or subsystem of the program
  - Often a description of a feature is called a *story*
- Middle Level: Rules
  - Rules: one or more business rules that the feature must follow
- Bottom Level: Scenarios
  - Scenarios: one or more use cases that demonstrate a rule
- Basement level: Steps
  - Steps: one or more execution steps and pre/postconditions for a scenario

Feature Syntax

- **Feature:** <text>
  - <text> can be any multi-line text that extends until the next keyword

- <text> is usually a one line description of the feature followed by:

  As a <role>

  I want <function>

  So that <reason / benefit>
  - <role> : user, administrator, customer service, data analyst, …
  - <function> : what functionality the feature provides
  - <benefit> : what business goals the function serves for the role

Feature Example

Feature: Rent-A-Cat listing

As a user

I want to see a listing of available cats in the rent-a-cat facility

So that I can see what cats are available for rent.

Rule Syntax

- **Rule**: <text>
  - <text> can be any multi-line text that extends until the next keyword
  - Multiple rules can follow the Feature keyword

- Examples:

```
Feature: Rent-A-Cat listing
…
Rule: When there are not cats, the listing is an empty string.
Rule: When there are cats, the listing is one line per each cat.
```

Scenario Syntax

- **Scenario:** <text>
  - <text> can be any multi-line text that extends until the next keyword
  - Multiple scenarios can follow the Rule keyword

- Examples:

```
Rule: When there are cats, the listing is one line per each cat.
Scenario: List available cats with 1 cat
…
Scenario: List available cats with 2 cats
…
```

Step Syntax

- One or more steps describe a scenario
  - **Given** <text>: Describes a precondition
  - **When** <text>: Describes an execution step
  - **Then** <text>: Describes a postcondition

- There can be multiple **Given**, **When**, **Then** steps for a scenario
  - When there are multiple steps of same type can use **And** keyword
  - # First precondition
Given <text1>
Equivalent to **Given** <text2>
And <text2>

Step Example

```
Scenario: List available cats with 1 cat
Given a rent-a-cat facility
Given a cat with ID 1 and name "Jennyanydots"
When I list the cats
Then the listing is: "ID 1. Jennyanydots\n"
```

Background Syntax

- Each feature can have an optional Background
  - Same purpose as @Before in JUnit (common preconditions for all scenarios)
- **Background:** <multiple **Given** steps>
  - Comes immediately after the **Feature** keyword
- Example:

```
Background:
Given a rent-a-cat facility
And a cat with ID 1 and name "Jennyanydots"
And a cat with ID 2 and name "Old Deuteronomy"
And a cat with ID 3 and name "Mistoffelees"
```

How are the steps executed on code?

- No, the plain English scenarios won't run automatically on code
  - At least we are not there yet ☺
- You must implement the steps (in Java)
  - Cucumber framework will use regular expression matching
    --- to match plain English steps to your Java steps
  - Same steps are used repeatedly for many scenarios, so not much coding!
  - If different English words are used for same step, match with regex

Note Behaviors can be Ambiguous

- It's basically specification by example
  - A feature is described using a story consisting of scenarios
- Leaves room for ambiguity
  - Both in terms of specification and testing
- But the big pay off is in much better user collaboration
- Depending on domain/field, may not be a good choice
  - Great for user-facing functionality
  - Worse for back-end or safety-critical development

Practice

- Now let’s try what we learned on our rent-a-cat application!