Lecture 10 Supplement: Behavior-Driven Development (BDD)

Wonsun Ahn

TDD and ATDD

- TDD (Test Driven Development): really unit test driven development
  - One drawback is it does not test system-level behavior
- ATDD (Acceptance Test Driven Development): systems-level tests
  - Typically automated systems-level acceptance tests are written first
  - Then systems-level tests are broken down to unit tests using mocking
- Still follow the red-green-refactor loop, but with two layers:
  - Acceptance tests
  - Unit tests
- TDD and ATDD together allows coding to be driven by requirements
  - Requirements drives → Testing drives → Development

TDD and ATDD Weak Link: Requirements

- Requirements drives → Testing drives → Development

- But are the requirements driven by the end-user?
  - Requirements are written by requirement analysts after interviewing users
  - Requirements are like legalese: goal is complete specification for the coder
  - End-users rarely read requirements (even if they did, it would be gibberish)
- What if requirements need to change due to end-user input?
  - Changes must percolate through Requirements → Tests → Code
  - While that's happening, many tests may break and become unusable

Behavior-Driven Development

- Dan North introduced BDD (Behavior-Driven Development) at 2006
  - Article in “Better Software magazine”: https://dannorth.net/introducing-bdd/

- Dan laid down a few paradigms of BDD
  - 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: No user would describe SW this way

The rent-a-cat system shall enable the listing of cats such that an empty string is returned when no cats are available, and a full listing of cats are returned when there are cats available. Each line in the listing of cats will 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). The name of the cat 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
  - Now requirements as behaviors can easily be shared among stakeholders
  - Since behaviors *are* the tests, requirements and tests never go out of sync
- Now stakeholders become active participants in shaping requirements
  - Outside-in: now requirements and tests come directly from stakeholders
  - Agile: software can adapt quickly to changes in user demand
- BDD still follows the red-green-refactor loop

Dialect for Describing Behaviors: Gherkin or JBehave

- Dialect for describing behaviors must satisfy two criteria
  1. Must be similar to plain English so end-users can understand it
  2. Must have minimal 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!