Behavior-Driven Design and User Stories

Agile

- Good because ensures stakeholders are getting what they want because of constant feedback (validation vs verification)
  - Avoid miscommunication
- Maintain working prototype while deploying new features every iteration
- Uses BDD

Behavior-Driven Design (BDD)

- Asks questions about desired behavior of app before and during development to reduce miscommunication
- Captures answers as user stories
- Concentrates on behavior of app vs implementation of app
  - TDD tests implementation

User Stories

- 1–3 sentences in everyday language
- Fits on an index card
- written by/with customer
- 'Connextra format':
  - Feature name
  - As a {kind of stakeholder}, so that I can {goal}, I want to {task}
  - Captures the 'business value' of a feature
- Plain language, so it is nonthreatening to nontechnical customers
  - All stakeholders help brainstorms
- Easy to re-arrange / prioritize
- Short and specific to one behavior so easier to change during development

SMART User Stories

- Specific and Measurable
  - Can write down a step-by-step procedure that someone with no knowledge of the app could do to verify the specific behavior
  - Anti-example: 'UI should be more user-friendly'
  - Example: Given/When/Then: Given {precondition} when {action} then {result}
  - Good for writing automated tests
- Achievable (ideally, implementation in 1 iteration)
  - Divide into multiple stories that form an epic if cannot be completed in 1 iteration
  - Goal: always have working code at end of iteration
  - <1 story per iteration $\implies$ need to improve point estimation
- Relevant ('the 5 why's)
  - Ask why 5 times as follow-up questions and if you can get to 5, it is probably good
- Timeboxed (knowing when to give up)

Class-Responsibility-Collaborator (CRC) Cards

- Goal: Make design concrete:
- Answers:
  - What does this class know?
  - What does it do?
  - What other classes does it collaborate with and how?

Lo-Fi UI Sketches and Storyboards

- Avoids WISBNWIW (what I said but not what I wanted)
- Just a basic sketch
- Storyboards animate the prototype manually
  - Have sketches for each view and link buttons on each sketch to the next view
- Less intimidating to nontechnical stakeholders
- More likely to suggest changes to UI if no code behind it
- More likely to focus on interaction (behavior) rather than colors, fonts, ...
  - CSS can be set up later to change the exact look

Stories to Acceptance Tests

- Acceptance: ensure satisfied customer
- Integration: test full-stack functionality
- Cucumber connects 'almost plain language' test description to code
  - Can discuss with customer
  - You define your own 'vocabulary' (a domain language) and terms that make sense for your app

Cucumber

- `Given` $\triangleq$ state of world before event: set up preconditions
- `When` $\triangleq$ user does something
  - eg: simulate user pushing a button
- `Then` $\triangleq$ check expected postconditions
  - `And` and `But` can be used as extension
- Feature is the name of the story
- The scenario is ≥ 1 happy / sad paths
- 3–8 steps per scenario
  - Begin with keyword
- Have to use explicit examples; cannot be ambiguous
- Regex used to match scenarios to step definitions
  - eg: `When /I follow "(.*)"/ do |match| {...}`
  - Many defined in `cucumbet-rails-training-wheels`
- Generally relies on Capybara

```
Feature: display studenst in alpha order
    As an instructor
    So that I can quickly find a student in the list
    I want the students to be in alphabetical order by last name

Scenario: list students in alpha order
    Given student 'John Anderson' exists
    And student 'Jane Bennett' exists
    When I visit the list of all students
    Then 'Jane Bennett' should appear before 'Jon Anderson'
```

- Usually stored in `features` at the root; specs end in `.feature`
- Step definitions stored in `features/step_definitions` where you can have as many `.rb` files with step defs
- Common to randomize values when necessary
- Every scenario is stateless
- Basically always use regex

Capybara

- Library that simulates user by automating browser
- HTTP request to any route in app
- Submit forms, interact with UI elements
- Interact with app, receive pages
- Execute JavaScript
- Parse HTML, allow programmatic inspection
- Some methods:
  - `visit` goes to url
  - `click_link`
  - `fill_in` fills in a field in a form (can either use label or css selector)
    - eg: `fill_in 'form#form1 nameField', :with => '{value}'`
  - `save_page` will save a page to `tmp/capybara/`
- Makes requests through `rack-test` or `capybara-webkit` for JS support
- Can also run on `webdriver` which opens a browser and actually simulates it all
  - Limits Capybara to the UI