Cucumber and Capybara

Refining Scope

- Can refine scope with `find` which takes in a css tag and returns the value of the element
  - Quacks like a `page`
- `page` is the entire page

Requirements

- Explicit: usually part of acceptance tests; expressed in user stories
- Implicit: logical consequence of explicit, typically integration testing
- Domain language: vocabulary that makes sense for the specific app
  - Describes what happens not how
- Imperative: Explains how something happens (sequence of steps)
- Declarative: What actually happens during a test in the context of the app

Steps

- Can use step definitions as subroutines
- Avoid using `web_steps.rb` over custom step defs
- Can define cleanup in `after` block at end of step defs.

Helpful Resources

- Tabular data can be handled with a cheatsheet
- Scenario outlines can help with lots of test data
- There are ways to carry state from one step to another using instance variables
- Time travel with `Timecop` can help with time dependencies
- Cucumber tests can be given tags to only run the subset of the tests that abide by a set of tags

Points and Velocity

- Generally vote with the team for 1-3 points per story
  - Anything above that should be an epic (when you don't know exactly how challenging a story is)
  - Only point stories if you have a good idea of what exactly is required
- The value of a point is dependent on the team
- Velocity $\triangleq$ moving average of the number of points delivered per iteration
  - Only within a team; not across teams

Categories of Stories

- Backlog $\triangleq$ stories prioritized but not yet started ('in the queue')
  - Prioritized with customer according to business value
- Icebox $\triangleq$ stories not yet pointed / prioritized
  - May or may not get done lated
- Spike $\triangleq$ short investigation of problem, technique, tool, library, etc.
  - Bound the time
  - When done, discard code, then do it properly
- Can have tasks
  - At a very fine granularity, define step-by-step implementation process

Centralizing a Team's Activities

- Features $\triangleq$ SMART business value and have points

- Bugs $\triangleq$ aren't worth points
- Chores $\triangleq$ necessary but don't have business value or points (eg: refactor)
- Design documents, GitHub integration...

Agile Cost Estimation

- Does not agree on delivery of features by a specific date
- Commits resources to work in most efficient way possible until date D
  - Customer works with team to define priorities continuously up to date D
- There is a customer 'scoping' meeting
  - Bring designers, designs/sketches, developers, notes, etc.
  - Bring experienced engineers who ask questions
- Contract is time and materials cost over the time of the contract
  - Understood that the schedule may slip / be pulled in

P&D Perspective on Planning and Monitoring the Schedule and Budget

- Qualitative: project manager's experience
  - Experienced programmer can assign a timeline for a ticket
- Quantitative:
  - Lines of code
  - COCOMO* (constructive cost model): $\text{effort} = \text{orgFactor} * \text{codeSize}^{\text{penalty}} * \text{prodFactor}$
- Estimate before and after contract
  - Add safety margin of 1.3-1.5x
  - Make 3 estimates: best, worst, expected
- PERT (program evaluation and review technique) chart
  - Milestones
  - Tasks get from one milestone from another
  - Effort $\triangleq$ weight of task
  - Dependency $\triangleq$ some milestone needed before another milestone
    - May or may not require resources (eg: code review)
  - Critical path $\triangleq$ most cost intensive path through chart

Commonalities

- Both require a 'requirements elicitation' section
- Requirements documentation may be IEEE standard or cucumber / lo-fi sketches for agile
- Change management through VCS
- Schedule building through PERT vs points and velocity
- Cost estimation strategies detailed above
- Risk management considered in P&D where risky tasks are designated vs in agile where it's implicitly managed with points

Fallacies and Notes

- Do not use velocity to compare teams
- Avoid assigning more than 3 points per story
- Keep 'finished', 'delivered', and 'accepted' different
- Avoid dividing work by layers vs by stories
  - Maintains a smaller communication overhead
  - Ensures that everyone knows what everything does
- Careless use of negative expectations ('then I should not see...')
  - Can easily cause false negative
  - Can also be bad for careless positive expectations
  - Refine scope when possible

- Don’t test only the happy path