User Stories, Part 4

CSCI 5828: Foundations of Software Engineering
Lecture 10 — 09/25/2014

Goals

- Continue our introduction to the topic of user stories
  - Acceptance Testing User Stories
  - Guidelines for Good User Stories

Acceptance Testing User Stories

- The details on the “front” of a user story are deliberately kept to a minimum
  - It should describe functionality valuable to a user or customer
  - However, it is meant to serve as a reminder to hold a conversation with the user in order to learn details/expectations about that feature
- Once we have a conversation, there are two things we can do
  - we can add expectations about the feature on the “back” of the card
  - when this story gets added to an iteration, these expectations get translated into test cases
    - indeed, in agile settings, the developers write these test cases first and then develop the software that makes these test cases pass (TDD)

Test-Driven Development: A detour

- The idea is simple
  - No *production* code is written **except to make a failing test pass**
- Implication
  - You have to write test cases **before** you write code

- Note: use of the word “production”
  - which refers to code that is going to be deployed to and used by real users
- It does not say: “No code is written except…”

Test-Driven Design in One Slide or Less

- This means that when you first write a test case, you may be testing code that does not exist
    - And since that means the test case will not compile, obviously the test case “fails”
        - After you write the skeleton code for the objects referenced in the test case, it will now compile, but also may not pass
- So, then you write the simplest code that will make the test case pass

Example (I)

- Consider writing a program to score the game of bowling
- You might start with the following test

```
public class TestGame extends TestCase {
   public void testOneThrow() {
      Game g = new Game();
      g.addThrow(5);
      assertEquals(5, g.getScore());
   }
}
```

- When you compile this program, the test “fails” because the Game class does not yet exist. But:
  - You have defined two methods on the class that you want to use
  - You are designing this class from a client’s perspective

Example (II)

- You would now write the Game class

```
public class Game {
   public void addThrow(int pins) {
   }
   public int getScore() {
      return 0;
   }
}
```

- The code now compiles but the test will still fail: getScore() returns 0 not 5
  - In Test-Driven Design, Beck recommends taking small, simple steps
  - So, we get the test case to compile before we get it to pass

Example (III)

- Once we confirm that the test still fails, we would then write the simplest code to make the test case pass; that would be

```
public class Game {

   public void addThrow(int pins) {
   }

   public int getScore() {
      return 5;
   }
}
```

- The test case now passes!

Example (IV)

- But, this code is not very useful!
- Lets add a new test case to enable progress

```
public class TestGame extends TestCase {
     public void testOneThrow() {
        Game g = new Game();
        g.addThrow(5);
        assertEquals(5, g.getScore());
     }
     public void testTwoThrows() {
        Game g = new Game()
        g.addThrow(5)
        g.addThrow(4)
        assertEquals(9, g.getScore());
     }
}
```

- The first test passes, but the second case fails (since 9 ≠ 5)
  - This code is written using JUnit; it uses reflection to invoke tests automatically

Example (V)

- We have duplication of information between the first test and the Game class
  - In particular, the number 5 appears in both places
  - This duplication occurred because we were writing the simplest code to make the test pass
  - Now, in the presence of the second test case, this duplication does more harm than good
  - So, we must now refactor the code to remove this duplication

Example (VI)

```
public class Game {
   private int score = 0;
   public void addThrow(int pins) {
      score += pins;
   }
   public int getScore() {
      return score;
   }
}
```

Both tests now pass. Progress!

Example (VII)

- But now, to make additional progress, we add another test case to the TestGame class

  …

```
    public void testSimpleSpare() {
        Game g = new Game()
        g.addThrow(3); g.addThrow(7); g.addThrow(3);
        assertEquals(13, g.scoreForFrame(1));
        assertEquals(16, g.getScore());
    }
```

  …

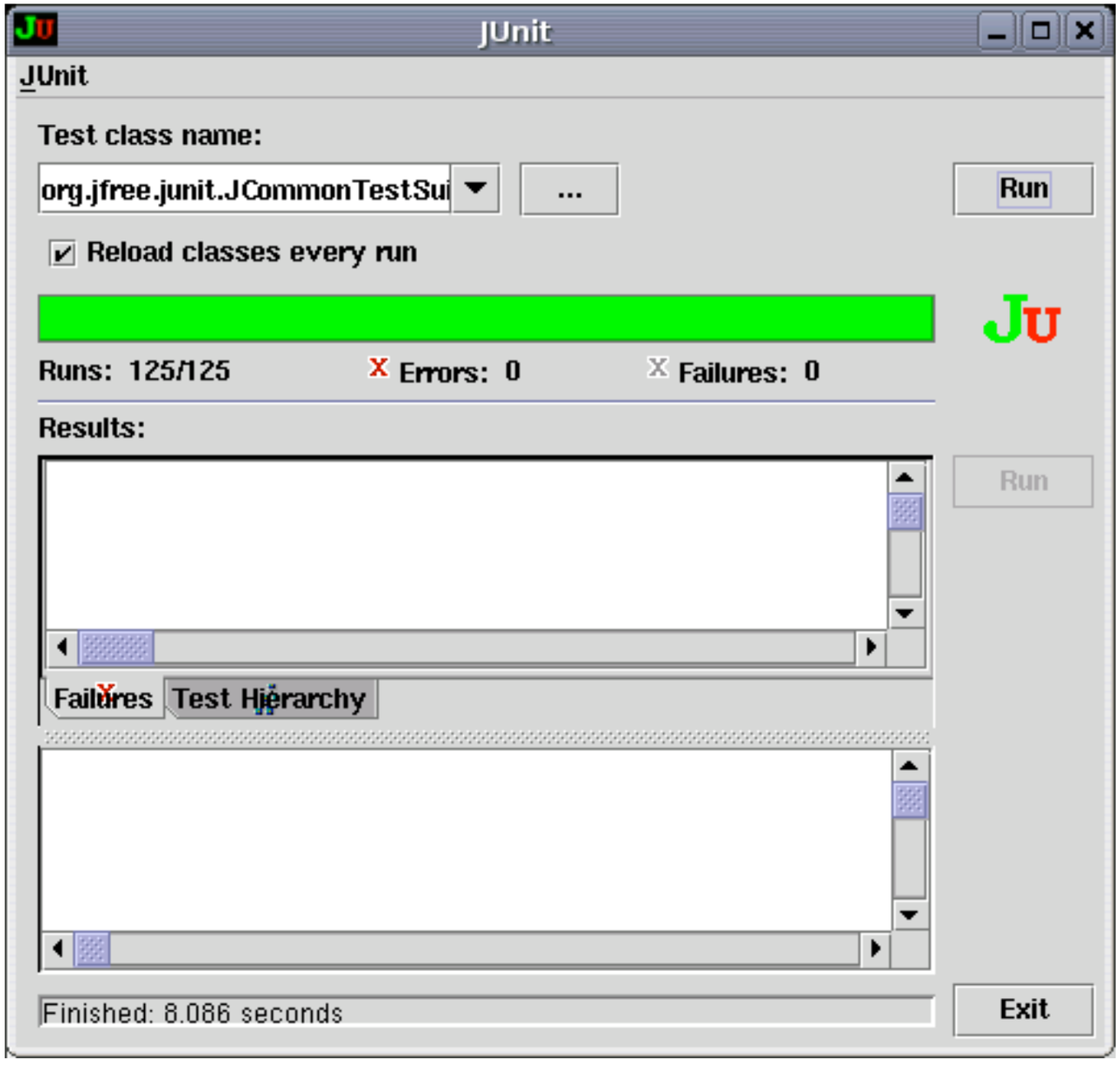
- We’re back to the code not compiling due to scoreForFrame()
  - We’ll need to add a method body for this method and give it the simplest implementation that will make all three of our tests cases pass

TDD Life Cycle

- The life cycle of test-driven development is
  - Quickly add a test
  - Run all tests and see the new one fail
  - Make a simple change
  - Run all tests and see them all pass
  - Refactor to remove duplication
- This cycle is followed until you have met your goal;
  - note that this cycle simply adds testing to the “add functionality; refactor” loop covered in the our lecture on refactoring

TDD Life Cycle, continued

- Kent Beck likes to perform TDD using a testing framework, such as JUnit.
- Within such frameworks
  - failing tests are indicated with a “red bar”
  - passing tests are shown with a “green bar”
- As such, the TDD life cycle is sometimes described as
  - “red bar/green bar/refactor”



JUnit: Red Bar...

- When a test fails:
  - You see a red bar
  - Failures/Errors are listed
  - Clicking on a failure displays more detailed information about what went wrong

Test-Driven Development: End of Detour

- Our textbook assumes that something like this is happening in the background as we add expectations to our user stories
  - When I say “translate the expectations/details” into test cases, I’m talking about exactly this
    - Pick your testing framework (there are a lot of them out there)
      - Use that framework to write code that
        - gets your system into a particular state
        - and then makes as many asserts() as needed to verify that the system performed as expected
- Run your test cases multiple times per day while developing the system.
  - Any changes that cause failures in previously working code will be detected right away

Example: User Story with Expectations

- For the user story
  - A company can pay for a job posting with a credit card
- The associated expectations might be
  - Verify that only Visa, MasterCard, and American Express are accepted
  - Test payments with good, bad, and missing card numbers
  - Test payments with an expired card
  - Test with different purchase amounts (including at least one payment over the card's limit)

Write Tests Before Coding (I)

- Acceptance tests provide a great deal of information that developers can use in advance of coding a user story
    - In order for that to occur, the tests have to be added to the card before it becomes “active” in one of the project’s iterations
- Tests are therefore typically written at the following times
    - whenever a conversation about the story occurs and details need to be recorded
    - as part of a dedicated effort at the start of an iteration before coding begins
    - whenever coding of a story reveals new questions that lead to new details that need to be translated into tests

Write Tests Before Coding (II)

- The start of an iteration is also a good time for a customer to review all stories that have become active and ask
  - what else do the developers need to know about this story?
  - what am I assuming about how this story will be implemented?
  - what can go wrong during this story?
  - are there variations on this story's behavior?
- The answers to these questions should be jotted down as expectations and eventually translated into tests
  - Or, in the case of the last question, translated to new user stories that are planned for later iterations
- The important point is that test writing is an integral part of our development process; something that happens on a regular basis at specified times

Users Write Tests

- A desired goal of agile software development is that tests come from users
  - We don't expect them to write their tests in JUnit (!) but the expectations that get generated come from the user/customer
  - The developers can always add on additional tests (as long as their tests do not contradict the expectations of the user)
    - but we want a strong user-centered perspective about what is being tested
- Why?
  - Our tests tell us when we are done!
  - As such, they need to specify what the user needs from the user story being tested

How do we get users to write tests?

- Given that most users will not be comfortable with low-level test automation frameworks, how do we get them to write tests during development
  - We can always restrict them to textual annotations on user stories
    - but that then requires developers to do a lot of translation work
- The book points at Ward Cunningham's Framework for Integrated Test
  - See <http://fit.c2.com/> for a still-running but "musty" website for FIT
  - See <http://fitnesse.org/> for a FIT-based tool under active development
- The idea here is that users can generate tables of data that can be automatically read by a testing framework and used to test an application
  - Feedback is provided by turning cells in the table green/red after the tests have been executed

How many user tests should be written?

- The user should continue to write tests as long as they add value an clarification to the story
  - The goal is not to generate a comprehensive set of tests
    - rather it is to provide sufficient detail for the story to get implemented
- The user is not responsible for low-level unit tests
  - Developers can write these using test-driven development
- The customer instead focuses on writing tests that
  - clarify the intent of the story
  - documents the behaviors that need to be handled to be considered “done”

Other types of tests

- Unit tests and acceptance tests focus primarily on functional testing
    - Does the system do what it needs to do to solve the user's problem
- There are many other types of tests that can be performed
    - UI testing: requires specialized testing frameworks
    - Usability testing: requires specialized HCC training
    - Performance testing and Stress testing
        - See CSCI 4753/5753 Computer Performance Modeling for details
- It will be up to the people responsible for software quality assurance (SQA) on your development team to perform these other test as appropriate

Guidelines for Good User Stories

- The book provides additional insight into the generation of useful stories with 13 user story guidelines
  - Start with Goal Stories
  - Slice the Cake
  - Write Closed Stories
  - Put Constraints on Cards
  - Size the Story to the Horizon
  - Keep the UI Out as Long as Possible
  - Some Things Are Not Stories
  - Include User Roles in Stories
  - Write for One User
  - Write in Active Voice
  - Customer Writes
  - Don’t Number Story Cards
  - Don’t Forget the Purpose

Start with Goal Stories

- A great way to kick start the process of generating user stories is to
  - Identify as many user roles for the system as possible
  - Identify the goals that user role has for interacting with the system
    - Each goal can be considered a user story (at least at the start)
- These user stories will then serve as inspiration for generating additional stories that are more detailed and more oriented to system functionality
  - Thus
    - “A Job Seeker wants to search for jobs continuously until they have found a job”
  - might lead to user stories related to the creation of persistent queries that are run on a regular basis (daily, weekly, etc.)

Slice the Cake

- User stories should touch all levels of a program's functionality
    - Think of user stories as defining a "vertical slice" of functionality, similar to slicing the cake (and revealing all the layers)
    - If you are building a web app, then for each story, you would develop
        - a little bit of the UI (i.e. a web page)
        - a controller that will respond to events from that page
            - either user clicks, form submits, or AJAX calls
        - the "database" that stores the data associated with that page
- By exercising every layer in the system, you reduce the risks of finding last minute problems; you also tackle hard issues up front

Write Closed Stories

- A “closed” story is one that finishes with the successful completion of a user/customer goal or objective
  - Thus, while it might be easy to write a requirement like this
    - A recruiter can manage the ads she has placed
  - it doesn’t really help us create functionality that helps recruiters achieve their goals. Instead it can be used to identify “closed stories” that do
    - A recruiter can review resumes from applications in response to an ad
    - A recruiter can change the expiration date of an ad
    - A recruiter can delete an application that is not a good match for a job
- Each of these stories accomplishes a clear task/goal, thereby providing value to that user

Put Constraints on Cards

- Every now and then the customer mandates an approach or implementation detail that must be obeyed rather than analyzed and designed
  - The new system must use our existing order database
  - The system must run on Windows 8 and Mac OS X 10.9.
  - The system must support peak usage of up to 50 concurrent users
- Our book recommends creating stories for these requirements and tagging them with the keyword “Constraint”
  - These stories do not get estimated/prioritized/scheduled
  - Instead they are “always on” and should be displayed such that team members are reminded of them
  - Best of all, tests should be written to monitor them (when that makes sense)

Size the Story to the Horizon

- Let iterations and releases guide you with respect to the level of detail associated with a story. You might start out with several “epic” stories
  - Job seekers can post resumes
  - Job seekers can search job openings
  - Recruiters can post job openings
  - Recruiters can search resumes
- And then be told that that posting resumes is the highest priority item
  - You would then start breaking that epic up into smaller, more detailed, more manageable stories and leave the others alone
- Basically, this guideline says to **focus your efforts on stories that are active now**; don’t spend too much effort adding details to stories that won’t be active for four or five iterations in the future

Keep the UI Out as Long as Possible

- You want to avoid specifying UI details in user stories
  - Think back to the use case that I presented at the start of Lecture 6
  - The “bad” version had direct references to the user interface
    - and that made the use case brittle (easy to break when things change)
  - The “better” version presented the desired functionality without reference to a particular UI or UI paradigm
    - This is way more flexible allowing us to design/implement the UI in a number of ways
    - For a particular story, its UI may change (or multiple UI’s may be supported), but the functionality should be the same
      - “Edit User” means the same thing regardless of how the edit is actually accomplished

Some Things Aren't Stories

- "If all you have is a hammer, everything looks like a nail."
- You're learning about user stories and how they can be used to structure and drive agile software development life cycles
  - Now that you've learned a useful tool, it might be tempting to use it for everything!
    - Functionality ⇒ User Stories
    - Constraints ⇒ User Stories
    - UI Guidelines ⇒ User Stories
    - Personas ⇒ User Stories
- You get the picture… don't use user stories to document information that would be better served in some other format

Include User Roles in the Stories

- If you go to the trouble of identifying user roles at the start of development
  - and you should (!)
- Then, use those roles in your user stories
- Do not do this
  - A user can access the last five days of server logs
- Do this
  - An administrator can access the last five days of server logs
- The author suggests a template for user stories that can encourage this
  - I as a `(role)` want `(function)` so that `(business value)`
  - I as an administrator want to access five days of server logs so I can monitor performance and identify problems

Write for One User

- This guideline recommends writing user stories from the stand point of a single user
    - This makes sense; even though a system can support multiple users at the same time, each user is interacting with the system individually
- This can also identify ambiguity
    - “A job seeker can remove resumes” might imply that a job seeker can remove the resumes of other job seekers
    - It would then become “A job seeker can remove her own resumes.”

Write in Active Voice

- Active voice lends energy to a statement
- Compare
  - “A job seeker can post a resume” or “A job seeker posts resumes”
- Vs.
  - A resume can be posted by a job seeker

- This is generically a good recommendation for writing style; passive voice can drag a document down, even when your “document” is one or two sentences long!

Customer Writes

- This guideline has been presented multiple times in the book
  - because it is such an important concept
- Your stories should be user-centered and speak with the user's voice
  - It should be written from the user's perspective and make use of terminology that makes sense to them
    - terminology that has to be learned by the developers
- It is REALLY EASY to let this drop and adopt a developer perspective on what the system needs to do
  - Agile guards against this by mandating frequent interaction with your users/customers and writing stories from their point of view

Don’t Number Story Cards

- Avoid the temptation to add numbers to user stories
    - It’s more important to talk about the “A list owner can create new topics” story than it is to talk about “Story 23” or “Story 42”
- This temptation will creep in when you want to associate stories with each other
    - Story 3 is too big?
    - Replace it with stories 3.1, 3.2, 3.3
- Doing this just adds “busy work” to the process, attempting to keep the numbers consistent rather than working on the stories themselves

Don’t Forget the Purpose

- Don’t forget user stories are meant to serve as “placeholders for conversations”
  - They don’t try to document everything
  - Instead our numerous conversations build up a shared understanding about the feature that is distributed across the team
    - and documented by executable test cases
- Let them serve their purpose as a scheduling and planning tool, allowing you to focus on getting details into code and test cases
  - We are likely more willing to keep code/test cases up-to-date than we are non-executable text on index cards anyway!

Summary

- Acceptance Testing User Stories
  - It is important that writing acceptance tests is integrated into the daily work practice of your software development project
  - The customer/user should write these tests (which are different from unit tests) and developers should then assist with translating them into executable tests
    - The goal is to clarify the intent of the story and identify when we are done
- Guidelines for Good User Stories
  - There are many things we can do to write useful user stories; the guidelines presented today can help
- We’re now ready to look at iteration/release planning in detail

Coming Up Next

- Lecture 11: Concurrency in Clojure, Part One
- Lecture 12: Concurrency in Clojure, Part Two