Brian Chen

CSCC01 – Introduction to Software Engineering
University of Toronto Scarborough, Fall 2016

Requirements Personas and User Stories

Week 1: Requirements and Personas

Requirements are entirely dependent on the **Software Development Process** we use
Personas are the archetypal users of a system:

- We usually need a few personas to cover typical users
- We have personas to design software for specific people
- Understand requirements
- Separate market segments
- Communicate with team members

Some traits of a persona may include: Age, Gender, Attitude towards Tech, Skills, Personality, Profession, etc.

Week 2: Requirements and User Stories

A **User Story** is a high level definition of a requirement
We structure it: **as a [Persona] I want to be able to [do thing] because of [benefit]**
Example: As Alice, a student, I want to be able to browse courses required for my program so I can prioritize my course choices.
When user stories conflict, you must consult the person who is responsible for the project (owner) and get their advice

Software Development Processes

Week 3: Software Development Processes

- Roles and Workflows
- Work products
- Milestones
- Design Guidelines

For any software process, we usually have four activities: **SDVE**
Specification

- Feasibility Study
- Requirement Elicitation and Analysis
- Requirements Specification
- Requirements Validation

Design and Implementation

- Architectural Design
- Algorithm Design
- Interface Design
- Data Structure Design
- Abstract Specification

Verification and Validation

- Verification: Are we building the product right?
- Validation: Are we building the right product?
- Component or Unit Testing: Individual components work appropriately
- System Testing or Integration Testing: System as a whole, emergent properties
- Acceptance Testing: Testing with customer data to check system meets needs

Evolution

- Software requirements must change through circumstances so we need to keep it updated

Week 4: The Agile Development Process

Advantages of the Agile Development Process:

- Can adapt to changes that will arise
- Improved risk management and planning
- Is the modern way of industry programming

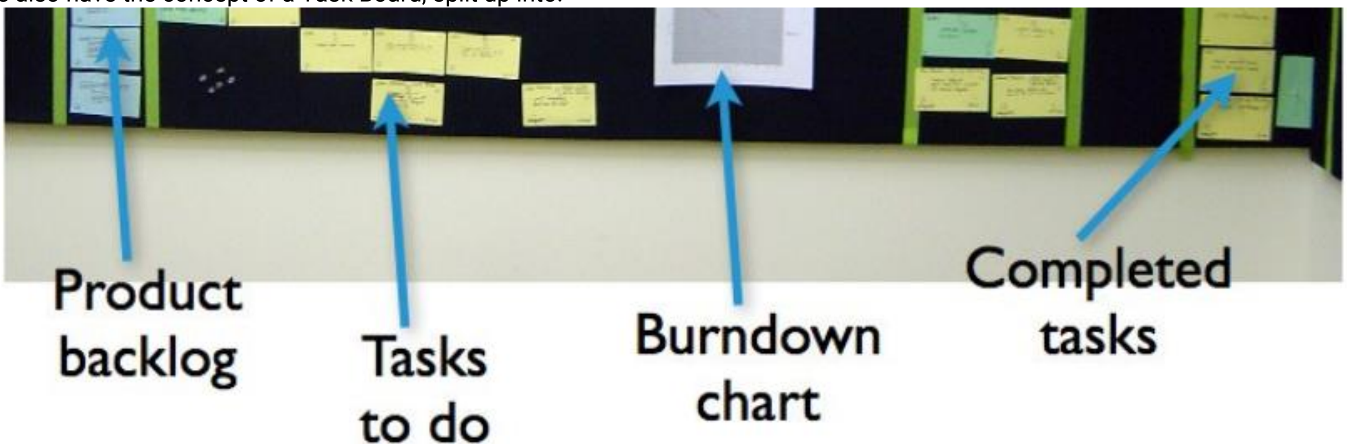
There is a **Scrum** ideology that is used:
Team: Implementers of the product
Users: Customers or consumers of product
Product Owner: Speaks for the customer
Scrum Master: Behaves like a team lead, project manager, resolves issues
Stakeholders: Project sponsors also have say like users do
Managers: Keep the team organization running smoothly
The Scrum process is split up into:
Sprints: Usually 30 days, anywhere from 2-4 weeks. This is an iteration of development

- Starts with a planning meeting
- Takes objects from Prioritized Project Backlog from **Product Owner** and builds a **Sprint Backlog**
- **DEVELOPMENT TIME**
- Ends with a Sprint Review Meeting to inspect product, reevaluate, and reassess

Daily Scrum: 15 minute meeting every day to debrief and check

- Daily, stand up meetings
- Only members who talk are Scrum Master and Product Owner
- "What has been completed" "What are the roadblocks" "What is to be done from today to next scrum"

We also have the concept of a Task Board, split up into:



We have a **Burndown Chart,** which tracks progress through days vs developer/story points
Burndown charts have the story points on the y-axis and days on the x-axis. We track the completion of stories on the burndown chart in relation to planned speeds.
We can only mark down a story once it is done. (e.g. 4/5 hours does not move the story points down 4 points, only when complete)

Week 5: Project Planning

Release Planning:
Release a plan which lays out overall project, then specifies which user stories are to be implemented for each release
Release plan is then used to create iteration plans
Project velocity: Amount of work completed during iteration, adjusted as project goes on
Developers and business agree on a set of stories to be implemented for the next release
Iteration Planning:
Meet at the beginning of each iteration
Just-in-time planning to stay on top of changing user requirements
We can readjust the iteration if we can't meet all our tasks
Concentrate on the most important issues

Week 5.5: Project Management

A sprint backlog looks more like this:

User Story	Tasks	Day 1	Day 2	Day 3	Day 4	Day 5	...
As a member, I can read profiles of other members so that I can find someone to date.	Code the ...	8	4	8	0		
	Design the ...	16	12	10	4		
	Meet with Mary about ...	8	16	16	11		
	Design the UI	12	6	0	0		
	Automate tests ...	4	4	1	0		
	Code the other ...	8	8	8	8		

The project manager keeps track of the **Task Board** and the **Burndown Chart**
We can split up any user story into a **Task Board organization, for example, the above User story would become**

Story	To Do	In Process	To Verify	Done
As an user I…	Code the… Design the…	Automate Tests…	Test the…	Meet with Mary about…

We measure progress by **Story Points**, these can be developer hours, team-days, money spent, etc.

Software Configuration Management

Week 6: Configuration Management

Configuration Management refers to the idea of managing software system structures over its lifetime: **SCVCR**

- System Modelling: Module Interconnection Level
- Composition: Managing System building and integration
- Version Control: Managing and controlling source code evolution
- Change Control: Managing and Controlling changes to a system
- Release Management: Managing the software release process

Source Control is the most important tool for commercial software development

- Tracks the source (code/docs/etc.)
- Central repository
- History of changes
- Management can control what goes to the source
- Collaboration and Syncing
- Multiple maintenance streams (branch/merge/etc.)
- Reproducible system state (revert)

Version Control: Centralized (e.g. SVN - Subversion)
We have Updates and Commits (update is pull, commit is push)
There exists one central canonical reference copy (repo/depot/etc.)
Multiple local copies at user-side.
Version Control – Distributed (e.g. Git)
Peer to peer model
No canonical copy, only working copies
Can do most of the work offline
Configuration Item – Single entity for purposes of configuration management
Version – Identifies the state of any config item at a particular time
Baseline – Version of a config item that has been formally reviewed and can only be changed through request
Branching: We branch to develop a new feature. We: git branch, git commit, git sync, then git merge
git branch **(lists)**
git branch <branch> **(creates)**
git checkout **(go to branch)**
git commit **(commits to current HEAD)**
git merge <branch> **(merges <branch> to current HEAD)**

Verification and Validation

Week 7: V&V

Validation: Does problem accurately capture real problem? Do we account for all needs of stakeholders?
Verification: Does our design meet the specification? Implementation? Does system do what we said it would?
Verification:

- **Testing:** Experiment with the program
- **Reviews:** Inspecting the program
- **Static Verification:** Reasoning about the program

Code Inspections are for finding defects. They note each defect but do not fix it.

- Checklist of problems
- Walkthrough of problems
- Round robin of issues
- Speed review

There exists **defects (missing requirement, wrong algorithm, etc)** that lead to **failures (showing up elsewhere)**
We can have: Unit Testing → Integration Testing → Function Testing → Performance Testing → Acceptance Testing → Installation
Disadvantages of Beta Testing:

- Might have a particular perspective
- Bias of technological competence
- Much harder to handle a user reported bug

Black Box Testing
User has to test functional requirements

- No knowledge of inner workings, only know input → output
- Unbiased, no programming knowledge required
- Cant identify all possible test cases

White Box Testing
Knows internal workings and tests specific parts

- Usually at unit test level
- Testing the particularities of an implementation

Partitioning → Partition the set of possible system behaviors to make sure we test samples from each partition
Regression Testing → Tests code upon modification
Unit-under-test (UUT) → Automates the process of running a test set
We may need to use **Test Stubs →** Parts of a program called by the Unit-Under-Test
Each Unit Test goes through: **Set Up, Exercise, Verify, and Tear Down**
We can classify these as Inside-out and Outside-in tests
Inside-Out Testing: We test from the very basics up into the advanced parts
Outside-In Testing: We test from the very advanced (using Mock objects) into the basics
Unit Testing:
Each individual unit is tested separately to check if it meets specification

Integration Testing:
Units tested together to make sure they work together

Bottom Up Integration Testing: Low level dependencies are tested first

Top Down Integration Testing: Top level is tested first with dependencies mocked using stubs

Inheritance Coverage and Testing:
With encapsulation, how do we test certain methods?
If we have a well-tested parent, do we just test overridden methods? (Dynamic binding says no)
Test Driven Development: Create test cases first which define requirements before coding (eXtreme programming)

1. Add new test
2. Run tests and Verify new test
3. Write code to satisfy new test
4. Run tests and check validity
5. Refactor the code
6. Repeat

SOLID and other Programming Design Patterns

Week 9: SOLID Programming

Single Responsibility Principle

- Each class should only have a single responsibility, that should be entirely encapsulated by the class

Open Closed Principle

- Software entities (classes, functions, etc.) should be **open for extension** but **closed for modification**
- Should have new features added by extending the class, not modifying the original class

Liskov Substitution Principle

- If S is a subtype of T, then S may be substituted for T without altering any properties
- E.g. a Square IS NOT a Rectangle, therefore cannot be a child of a rectangle.

Interface Segregation Principle

- No client should be forced to depend on things it doesn't use (e.g. Cube depending on Shape, it should depend on 3D Shape)
- Better to have lots of small interfaces that are specific than a few large ones

Dependency Inversion Principle

- We want to introduce an abstraction layer so we can define from high-level to low-level classes
- High level and Low level should depend on abstractions
- Abstractions should not depend on details, details should depend on abstractions

Builder Design Pattern → Separate the construction of a complex object from its representation

Python Testing

```
from dog import Dog, Puppy
import unittest


class TestDog(unittest.TestCase):
    ''' Test all public methods of Dog.
    '''

    def setUp(self):
        self.dog = Dog('Spot')

    def testWalk(self):
        self.assertEqual(self.dog.walk(), 'one two three four')

    def testTalk(self):
        self.assertEqual(self.dog.talk(), 'Woof!')
```

```
    def testRespondMyName(self):
        self.assertEqual(self.dog.respond('Spot'), 'Running back')

    def testRespondDiffName(self):
        self.assertEqual(self.dog.respond('Rex'), 'Ignoring')

class TestPuppy3(TestDog):
    ''' Test all public methods of Puppy.
    Third attempt: fix setUp.
    '''

    def setUp(self):
        self.dog = Puppy('Spot')

    def testTalk(self):
        self.assertEqual(self.dog.talk(), 'Woof!Woof!Woof!Woof!Woof!')


if __name__ == '__main__':
    # run tests for Dog
    print('Testing Dog')
    suite_dog = unittest.TestLoader().loadTestsFromTestCase(TestDog)
    unittest.TextTestRunner(verbosity=2).run(suite_dog)

    # third try at tests for Puppy
    print('\n\nTesting Puppy --- 3')
    suite_puppy = unittest.TestLoader().loadTestsFromTestCase(TestPuppy3)
    unittest.TextTestRunner(verbosity=2).run(suite_puppy)
```

Java Testing

```
package tests;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertFalse;

import dogs.Puppy;

import org.junit.Before;
import org.junit.Test;

public class PuppyTest extends DogTest {

  Puppy puppy;

  /* (non-Javadoc)
   * @see tests.DogTest#setUp()
```

```
   */
  @Before
  public void setUp() {
    super.setUp();
    puppy = new Puppy("Spot");
  }

  /**
   * Test method for {@link dogs.Puppy#talk()}.
   */
  @Test
  public void testTalk() {
    assertEquals("Woof!Woof!Woof!Woof!Woof!",
        puppy.talk());
  }

  /**
   * Test method for {@link Puppy#isMyName(java.lang.String)}.
   */
  @Test
  public void testIsMyNameMyName() {
    assertFalse("Puppy.isMyName returned a wrong value.",
        puppy.isMyName("Spot"));
  }

  /**
   * Test method for {@link Puppy#isMyName(java.lang.String)}.
   */
  @Test
  public void testIsMyNameOtherName() {
    assertFalse("Puppy.isMyName returned a wrong value.",
        puppy.isMyName("Rex"));
  }
}
```

Observer Design Pattern

```
package observerexample;

import java.util.Observable;

/** A product in a store. **/

public class Product extends Observable {

  public void changePrice(double newPrice) {
    if (Math.abs(price - newPrice) < EPSILON) {
      return;
    }
    setChanged();
    price = newPrice;
    notifyObservers(new PriceChange(this));
  }
}
```

```
package observerexample;

import java.util.Observable;
import java.util.Observer;

public class Shopper implements Observer {
  public void update(Observable obs, Object arg) {
    PriceChange pc = (PriceChange) arg;
    Product product = pc.getProduct();
    String msg = String.format("%s was notified about a price change of %s at %s to %.2f on %s.",
        name, product.getName(), product.getStore(), product.getPrice(), pc.getDate());
    System.out.println(msg);
  }
}
```

```
    Product banana = new Product("banana", 0.59, "Loblaw");
    Product cereal = new Product("cereal", 7.49, "Target");

    Shopper anya = new Shopper("Anya Tafliovich");
    Shopper paco = new Shopper("Paco Estrada");

    banana.addObserver(paco);
    cereal.addObserver(paco);
```