Introduction to Testing
Lab Update

- Will be receiving purchasing card later today
- What things need to be purchased for the lab?
  - Tape, paper towels, windex, whiteboard cleaner, whiteboard erasers, post-its, whiteboard markers (tons), rulers (12-18inch, for burndown charts), graph paper
  - Portable webcams for meetings, Speakers, desktop microphones, dvi cable, wired xbox controllers 3-4
  - Javascript books, networking in Javascript, web server book, etc.
  - MX vs ATV reflex, Skyrim, Arkam City, Skyward Sword, El Sheddai, Wii Motion Plus, Child of Eden, Gunstringer, Kinect Games, Resistance 3, Dark Souls
  - iPhone for testing, Vacuum
  - Photoshop on Alienware machines
  - More whiteboards in big lab (Yoshi, right of TV)
Upcoming Events

- **February 16:** Visit by Si Shen, CEO Papaya
  - Discussion of mobile social games

- **February 15:** Emily Short
  - Building a Comedy of Simulated Manners
  - E2 180, 11am

- **February 22:** Graeme Devine
  - E2 180, 11am

- **February 28:** Visit by Microsoft Studios
  - Game crits by staff from Microsoft Studios
  - Discussion of career opportunities

- **Prom Week released today!**
  - Please consider voting for it in the IGF Audience Choice awards: http://igf.com/audience.php
Upcoming deadlines

- **Friday (February 17)**
  - Team status reporting due

- **Thursday (February 23)**
  - Sprint 2 ends
  - Game playtesting plan
  - Web site framework
Scrum Issues from Sprint 1

- Break tasks down into smaller pieces
  - Tasks should ideally be around 2-4 ideal work hours
  - Warning sign: task over 8 ideal work hours
- Declaring a task to be done
  - Checked in to the repository
  - Bug-free
  - Pass unit tests
- Testable user stories
- Pull/update from repository often
  - Reduces merge problems
  - Pull/update 2-3 times a day when coding – keeps conflicts small
  - Commit small pieces (i.e., smaller tasks), so impacts on others are smaller
- Underestimating time for tasks
- Keeping daily scrums focused
Team Exercise

- Gather as a team

- Have each team member determine how many hours they actually worked on the project each week during Sprint 1
  - Try to focus just on time spent on task, and not on total time in lab (since that includes eating, game playing, etc.)

- Across the entire team, determine:
  - Average amount of time the total team spent working per day (assume 21 days in the sprint)
    - Sum each person’s hours/week to get total hours/sprint for each person
    - Sum all of the individual hours/sprint to get total team hours/sprint
    - If a person is missing, add in an estimate of their total hours/sprint
    - Divide team total hours/sprint by # of days (21)
    - This is team hours/day
  - Average amount of time each person spent working per week
    - Start with total team hours/sprint (from above)
    - Divide by (3 weeks * # of team members)
## Sprint 1 Statistics

<table>
<thead>
<tr>
<th>Team</th>
<th>IWH Complete (entire team)</th>
<th>IWH/Day (entire team)</th>
<th>IWH/person-week</th>
</tr>
</thead>
<tbody>
<tr>
<td>Devils Bargain</td>
<td>59.5</td>
<td>2.83</td>
<td>2.20</td>
</tr>
<tr>
<td>Hello World</td>
<td>150</td>
<td>7.14</td>
<td>6.25</td>
</tr>
<tr>
<td>Puzzle Defense</td>
<td>203</td>
<td>9.67</td>
<td>8.46</td>
</tr>
<tr>
<td>Firewall</td>
<td>151</td>
<td>7.19</td>
<td>4.58</td>
</tr>
<tr>
<td>Sonar</td>
<td>91.5</td>
<td>4.36</td>
<td>3.39</td>
</tr>
<tr>
<td>Chroma</td>
<td>171</td>
<td>8.14</td>
<td>6.33</td>
</tr>
<tr>
<td>MicroVentures</td>
<td>101</td>
<td>4.81</td>
<td>3.37</td>
</tr>
</tbody>
</table>
Software Engineering: Model-based collaboration

- Over a software project, software engineers collaborate to create a series of models that describe the system’s behavior
  - capture system goals and behavior in models
  - creation of shared meaning around these models
- Constant task: elimination of error and ambiguity within the models

---

**Stakeholders/ Users/Customers**

**Software Engineers**

**Models**

**Collaboration Goals**

- Develop shared mental model of system
- Negotiate scope and capabilities
- Elicit requirements from stakeholders

- Drive convergence to single architecture
- Negotiate modular decomposition
- Reduce dependencies among org. units

- Resolve inconsistencies
- Negotiate interfaces and use protocols
- Record design rationale

- Negotiate dependencies among classes and methods
- Identify and eliminate bugs
- Resolve edit conflicts

- Identify and confirm bugs
- Verify and apply code patches
- Identify new feature requests

**Project management**

**Requirements Engineer**

- Use cases, scenarios, requirements statements

**System Architect**

- Feature model, arch. diagrams, tradeoff analyses, arch. document

**Developer**

- UML, design documents

**Code Unit Test Inspection**

- Source code, test code, build scripts, test harness

**Bug reporting, tracking, & fixing**

- Bug reports, code patches
Ways of removing software errors

- **Inspection**
  - Engineers examine a model, looking for errors
  - Use their expertise to evaluate model for correctness
  - Example: software inspections, requirements inspections, heuristic walkthrough of user interfaces

- **Refinement**
  - Engineers take a given model and produce another model from it
  - Process of making new models exposes missing information and assumptions in base models
  - Examples: formal specifications, pseudocode, UML

- **Simulation**
  - Make a model executable, then verify its behavior meets expectations
  - Example: executable Statecharts

- **Rewriting**
  - Take an existing model and rewrite it at the same level of abstraction, hopefully simplifying the model and thereby removing error
  - Example: code refactoring

- **Testing**
  - Provide specific inputs to an executable model, and then verify the outputs are correct (with respect to a specification of correct behavior, or oracle)
  - Example: software testing

- **Analysis**
  - A piece of analysis software examines an executable model of a system (source code) for errors
  - Example: static source code analysis (Findbugs), verifying XML against a DTD
Software Testing

- Software testing has two primary goals:
  - **Descriptive**
    - Provide an evaluation of the quality or acceptability of a software system
    - “Is the program acceptable to the customer?”
    - “Is the program ready to be released?”
  - **Perfective**
    - Reveal problems in a software system, so they can be fixed
Testing Scope

- Software can be tested at varying sizes
  - Unit test
    - Testing a small number of methods/functions (often 1)
    - Testing new code before check-in
  - Object test
    - Testing all of the methods of an object’s implementation
    - Usually focuses on one object, but may include other objects (e.g., an object whose role depends on working with other object, such as the strategy pattern).
  - Integration test
    - Testing how a collection of objects or collection of methods/functions work together.
    - Focus is on ensuring that bugs are not introduced by interactions among the behavior of constituent parts
  - System test
    - Testing an entire software system as a whole
Functional vs Structural Testing

- There are two fundamental approaches to testing software
  - Functional (Black Box)
    - No detailed knowledge of internal implementation
    - Treats system as function mapping inputs to outputs
  - Structural (White Box or Clear Box)
    - Can use detailed knowledge of internal implementation detail
    - Can “see” inside the implementation
Distinguishing Functional vs Structural

Functional (Black Box)

Test Cases (Functional Method I)

Test Cases (Functional Method II)

Structural (White Box)

Test Cases (Structural Method I)

Test Cases (Structural Method II)
Triangle problem

A program reads three integer values from the console. The three values are interpreted as representing the lengths of the sides of a triangle. The program prints a message that states whether the triangle is scalene, isosceles, or equilateral.

Write a set of test cases (inputs to the method) that would adequately test this method.

Recall:

- Equilateral triangle:
  - Three equal sides, three equal angles (all 60 degrees)
- Isosceles triangle:
  - Two equal sides, two equal angles
- Scalene triangle:
  - No equal sides, no equal angles
Test cases (from the class)

- A, 1, 1 – input validity
- -1, -1, -1 – input validity
- 1, 5, 7 – invalid triangle (1 + 5 < 7)
- Maxint+1, maxint+1, maxint+1 – input validity
- Sqrt(2), 1, 0.5 – input validity
- 4, -4, 0 – input validity
- 4, 4, 0 – check for early exit (isosceles)
- 1, 1, 1 – check minint, equilateral
- 1, 2, 1 – not valid triangle
- 5, 5, 4 – isosceles (and permutations)
- 3, 4, 5 – scalene
- 1, 1, 1, 1 – too many inputs
- 1 – too few inputs
- \t – monkey on keyboard, invalid inputs
- Nothing – too few inputs
- Try to kill task during processing ^c or equivalent
Did you miss these?

- Some values specify an invalid triangle
  - Examples; 1, 2, 3 or 1, 2, 4 are not valid triangles
    - Even if the angle between the first two sides is 180 degrees, the third side is still too long!

- Did you give as test cases:
  - Non-integers?
  - Non-numeric values?
  - Negative numbers?

- Note how some of the values are given by your knowledge of the domain (triangles), and others based on knowledge of the range of possible inputs
Types of Functional Tests

- **Boundary Value Testing**
  - Test values just before, on, and after some input boundary
  - E.g., -1, 0, 1

- **Equivalence Class Testing**
  - Provide representative samples within a large space of similar ("equivalent") tests

- **Decision-Table Testing**
  - Create a table that lists all possible outcomes for a set of inputs
  - Use this table to drive test case selection
Boundary Value Testing

- Core idea: provide inputs just before, on, and just after some inflection in the input space
- Intuition: errors occur at extremes of input space
- Example:
  - Determining if a point is in a range

```
min  min+  nom   max-  max
```

Provide test inputs at: min, min+, nominal, max-, and max
Variations on Boundary Value Testing

- Robustness testing
  - Add min- and max+ to values tested

- Worst-case testing
  - When multiple variables might be experiencing extreme at the same time. Example: robust worst-case testing with two variables
Triangle Problem (Revisited)

- Let’s revisit the triangle problem, and determine the set of boundary value test cases
  - Set arbitrary upper bound on side length of 200
  - Lower bound of all ranges is 1 (are integers)
  - Focus only on valid inputs