CS189A
Software Engineering: Concepts and Practices

Jianwen Su
Department of Computer Science
UC Santa Barbara

https://capstone.cs.ucsb.edu/cs189a/cs189a_sched.html
Today’s Lecture

- Progress checking and upcoming deadlines

- Updates from teams
  - Retrospective for Sprint 2 (1 good, 1 bad, and changes)
  - Plans for Spring 3 & PRDv2

- Design technique (last three weeks)

- Today: Software Testing
Overall Plan @Week 7

- Four 2-week sprints (dates adjusted)
  - Oct 11-25 (PRD v1 – tools, technologies, design, terminology)
  - Oct 25-Nov 8 (use cases/user studies, prototyping, PRD v1, PRD v2)
  - Nov 8-22 (design, prototyping, testing, PRD v2)
  - Nov 22-Dec 3 (prototype demo/pres prep, prototyping and testing)

- Fall presentations and demos: Dec 3, 3:30-5:30 (Girvetz 1004)

- Specify what the product will do
  - Vision statement
  - Product Requirements Document (PRD) (due Oct 29 and Nov 26*/29)
  - Design tools, brainstorming, coding (tests and implementation)

- Build and test an initial prototype
  - Typically teams iterate on these activities until they converge to a working prototype!
This Week’s Plan

❖ Team activities
  – Scrum: Sprint 3, PRDv2
❖ Section: TA meetings

Upcoming deadlines:
❖ Nov 22: Sprint 3 ends (Sprint 4 starts)
  – Product Requirements Document v2
❖ Nov 29: PRDv2 due
Sprints 3 and 4

- Sprint 3 starts today
- Break down stories into tasks & components associated with design
  - Prioritize stories
  - Assign timings to stories/use cases AND store/use-case tasks
  - Specify acceptance/test that can be used to verify a story is finished
- Sprint: Prototype tasks (primary implementation before demo)
  - Prioritize tasks
  - Assign timings to tasks
  - Specify what test(s) are to be used as evidence of task completion/acceptance (use case post condition OR user story acc test)
  - Each member/developer chooses task, implements, and tests task
  - Another member does code review/test and accepts the pull request
    - Test is the one specified above (Acceptance)
  - When store/case is complete, some member performs story test/acceptance
PRDv2: Your Living Requirements Document: A Shared Google Doc

- Authors, Team, Project Title
- Intro: problem, innovation, science, core technical advance (3+ pages)
  - Define project specifics, team goals/objectives, background, and assumptions
- System architecture overview
  - High level diagram (1 page)
  - User interaction and design (1+ pages) – ie detailed design
- Requirements (functional and non-functional)
  - User stories or use cases (links) → 20+ for PRDv2 prioritized w/acc. tests
  - Prototyping code, tests, metrics (10+ user stories): github commits/issues
- System models (1+ pages)
  - Contexts, interactions, structural, behavioral (UML)
  - Use cases, sequencing, event response, system state, classes/objects
- Appendices - Technologies employed
Your Project Design: PRDv2

- **Architecture (hardware/software)**
  - Evolve your overview picture from PRDv1 to provide significantly more detail and any updates or changes

- **Detailed design**
  - UML diagrams of **primary data structures** that comprise the system architecture connected via their associations (if any)
    - Ensure that each "class" is balanced in terms of cohesion & coupling
    - Annotate with pre/post conditions when appropriate
  - **Sequence diagrams**
    - Synchronous and asynchronous for key interactions between classes
      - At least 3 different interactions
    - User interactions with the system
      - At least 3 different interactions
      - Can be a human user or a machine user (API) interaction
        - Event response, updated application state
      - If you have a user interface: **Provide mockups for primary UIs**
PRDv2 User Stories / Use Cases

- Revise spec to add detail to the functional specification to match your design
- Add user stories and break up the stories you have into finer grained stories
  - Provide UML, sequence diagrams, dataflow diagrams
  - Goal: a CS senior should be able to take your doc and implement the project
- For each fine-grained story, provide a description and acceptance test
  - Provide time estimates (1 person-hours) for each story implementation
    - Ensure you can finish the implementation in the time you have (this/next quarter)
  - Prioritize tasks to have a complete prototype by the end of this quarter
    - Focus on the externally facing interfaces, mock out what you cannot get to
  - Write unit tests to implement tasks for mandatory tasks
    - Document these tasks (autogen the documentation/usage)
  - Add trello/pivotal task links (titles must match) to PRDv2 for each story
- Prototype designed mandatory tasks; add github commit ID/link to PRDv2
  - Github must have unit tests, documentation (for anything without unit tests), and prototyping implementations for each story in Sprint
- If you have a user interface
  - Provide mockups that are tied to the functionality described in 1+ components
Completing the Fall Quarter

- Today: (Testing)  
  - Meetings with Sea++, Pina Colada, Null Pointers  

- Nov 15: Team meetings  
  - Meetings with 5-7 teams (to be invited)  

- Nov 22: Potpurri  
  Presentation/demo details  
  looking ahead (break & Winter)  
  - Meetings with 3-5 teams (to be invited)  

- Nov 29: short class meeting  

- Dec 3, 3:30-5:30, Project presentations with demo  
  - Will be recorded: Check with your mentor if concerns  
  - All mentors are invited:  
    Team leads—please invite your mentors
Today’s Lecture

- Progress checking and upcoming deadlines
- Updates from teams
  - Retrospective for Sprint 2 (1 good, 1 bad, and changes)
  - Plans for Spring 3 & PRDv2

- Design technique
  - 3 weeks ago: Requirements engineering → PRDv1
  - 2 weeks ago: Techniques and tools → PRDv2
  - Last week: drill down—the decorator pattern

- Today: Software testing
Software Testing

- What and why
- How
- Planning

http://barbie.uta.edu/~mehra/Book1_The%20Art%20of%20Software%20Testing.pdf
What is Software Testing

- Testing is the process of demonstrating that errors are not present

- The purpose of testing is to show that a program performs its intended functions correctly

- Testing is the process of establishing confidence that a program does what it is supposed to do

- Testing is the process of executing a program with the intent of finding errors
Most Costly Software Errors

- NASA’s Mars Climate Orbiter (1998, $125m)
- Ariane 5 Flight 501 (1996, $500m)
- EDS Child Support System (2004, $1b)
- Soviet Gas Pipeline Explosion (1982)
- Bitcoin Hack, Mt. Gox (2011, 850,000 bitcoins, $500m)
- Heathrow Terminal 5 Opening (2008, 10 days, 42000 bags, 500 flights)
- The Mariner 1 Spacecraft (1962, $18m)
- The Morris Worm (1988, $100m)
- Patriot Missile Error (1991, 28 deaths)
- Pentium FDIV bug (1994, $475m)
- Knight’s $440 Million Error (2012)
10 Principles of Testing

1. A necessary part of a test case is a definition of the expected output or result.
2. A programmer should avoid attempting to test his or her own program.
3. A programming organization should not test its own programs.
4. Thoroughly inspect the results of each test.
5. Test cases must be written for input conditions that are invalid and unexpected, as well as for those that are valid and expected.
6. Examining a program to see if it does not do what it is supposed to do is only half the battle; the other half is seeing whether the program does what it is not supposed to do.
7. Avoid throwaway test cases unless the program is truly a throwaway program.
8. Do not plan a testing effort under the tacit assumption that no errors will be found.
9. The probability of the existence of more errors in a section of a program is proportional to the number of errors already found in that section.
10. Testing is an extremely creative and intellectually challenging task.
already have seen that it is impossible to test a program sufficiently to guarantee the absence of all errors. Methodologies discussed later in this book let you develop a reasonable set of test cases for a program, but these methodologies still require a significant amount of creativity.

Summary

As you proceed through this book, keep in mind these three important principles of testing:

• Testing is the process of executing a program with the intent of finding errors.
• A good test case is one that has a high probability of detecting an as yet undiscovered error.
• A successful test case is one that detects an as yet undiscovered error.
Black-box Testing vs White-box Testing

- **Black-box strategy:** data-driven, input/output-driven
  - Unconcerned about the internal logic
  - Test data derived from specification (user stories/use cases)
  - A criterion: exhaustive input testing

- **White-box strategy:** logic-driven
  - Test data derived form internal structure of software
  - Exhaustive statement coverage, path testing

- **Management of complexity**
Inspection: 3-5 people, checking:
- Data reference (dangling references, subscripts, etc.)
- Computation (value overflow, non-arithmetic variables, etc.)
- Data declaration (declared, Initialized, etc.)
- Comparison (mixed types, etc.)
- Control flow (loop termination, program termination, iteration error, etc.)
- Input/output (file attributes, OPEN statements, format, buffer size, etc.)
- Interfaces (parameter number & types, etc.)
- Others (missing functions, etc.)

Walkthrough 3-5 people, similar but manually “execute” the program

Peer reviews
Test Cases

- Each test case should include:
  - Title
  - Description
  - Assumptions and preconditions
  - Expected results
  - Clear test steps and concise
  - Reuseable

- Example:

TC02 - Login Page - Authenticate Successfully on gmail.com
Last updated on: 29th Nov 2015, Last saved by: Jake Bartlett, View stats
A registered user should be able to successfully login at gmail.com.

Precondition: the user must already be registered with an email address and password.
Assumption: a supported browser is being used.

Test steps

1. Navigate to gmail.com
2. In the 'email' field, enter the email of the registered user.
3. Click the 'Next' button.
4. Enter the password of the registered user
5. Click 'Sign In'

Expected result

A page displaying the gmail user's inbox should load, showing any new message at the top of the page.
## Test Case Design Methodologies

<table>
<thead>
<tr>
<th>Black Box</th>
<th>White Box</th>
</tr>
</thead>
<tbody>
<tr>
<td>Equivalence partitioning</td>
<td>Statement coverage</td>
</tr>
<tr>
<td>Boundary-value analysis</td>
<td>Decision coverage</td>
</tr>
<tr>
<td>Cause-effect graphing</td>
<td>Condition coverage</td>
</tr>
<tr>
<td>Error guessing</td>
<td>Decision-condition coverage</td>
</tr>
<tr>
<td></td>
<td>Multiple-condition coverage</td>
</tr>
</tbody>
</table>

- **White box: logic-coverage**
Equivalence Partitioning

- Basic rational: program behaves similarly on similar values (cases, inputs)
- Identifying the equivalence classes
- Identifying the test cases
Boundary-Value Analysis

- Each equivalent class, select one or more elements on edges of the class for test
- Test cases are also derived by considering the result space (output equivalence classes)
Cause-Effect & Error Guessing

- Techniques to reduce complexity
- Identifying “high potential” (for finding bugs!) test cases
Module Testing

- Test management technique
  - Bottom up approach
  - Top down approach

- Individual pieces are tested first, and then

- Non-incremental testing:
  - integration test

- Incremental testing
  - E, C, F, then
  - B, D, and finally
  - A

- Use “stubs”
System Testing

- Not only test the system, but test the product!
- Demonstrate how the program does not meet the objectives
  - Need clear objectives!
- 15 Categories of system testing:
  - Facility (functions), volume (e.g., data size), stress (heavy load), usability, security, performance/efficiency, storage, configuration, compatibility, installability, reliability, recovery, serviceability, documentation (accuracy), procedure (workflow)
Inductive Debugging

The induction process is illustrated in Figure 7.1. The steps are as follows:

1. Locate pertinent data.
   A major mistake debuggers make is failing to take account of all available data or symptoms about the problem. The first step is the enumeration of all you know about what the program did correctly and what it did incorrectly—the symptoms that led you to believe there was an error. Additional valuable clues are provided by similar, but different, test cases that do not cause the symptoms to appear.

2. Organize the data.
   Remember that induction implies that you’re processing from the particulars to the general, so the second step is to structure the pertinent data to let you observe the patterns. Of particular importance is the search for contradictions, events such as that the error occurs only when the customer has no outstanding balance in his or her margin account. You can use a form such as the one shown...

Other techniques:
- Deductive
- Backtracking
- Testing
Yes and no…

– When you write code in professional settings with teammates, definitely!
  o In such settings, failing to test your code just means you are inflicting errors you could have caught on teammates!
  o At Google, people get fired for this sort of thing!
– So… in industry… test or perish!

But what if code is just “for yourself”?

– Testing can still help you debug, and if you go to the trouble of doing the test, JUnit helps you “keep it” for re-use later
– But obviously no need to go crazy in this case

https://www.tutorialspoint.com/junit/index.htm
A Bug Can Reveal a Missing Test

… but can also reveal that the specification was faulty in the first place, or incomplete

– Code “evolves” and some changing conditions can trigger buggy behavior
  
  o Perhaps specification needs to evolve

– This isn’t your fault or the client’s fault but finger pointing is common

Great testing dramatically reduces bug rates

– And can make fixing bugs way easier

– But can’t solve everything: Paradise isn’t attainable in the software industry
The Q/A Cycle

- Real companies have quality assurance teams
- They take the code and refuse to listen to all the long-winded explanations of why it works
- Then they do their own, independent, testing
- And then they send back the broken code with a long list of known bugs!
- Separating development from Q/A really helps
Why is Q/A a cycle?

- Each new revision may fix bugs but could also break things that were previously working.

- Moreover, during the lifetime of a complex application, new features will often be added and those can also require Q/A.

- Thus companies think of software as having a very long “life cycle”. Developing the first version is only the beginning of a long road!
Q/A Can Help But…

- The best code written by professionals will still have bugs!
  - They reflect design oversights, or bugs that Q/A didn’t catch
  - Evolutionary change in requirements
  - Incompatibilities between modules developed by different people, or enhancements made by people who didn’t fully understand the original logic

- So never believe that software will be flawless
- Our goal in CAPSTONE is to do as well as possible