Week 9 - Wednesday

COMP 3100

Last time

- What did we talk about last time?
- JUnit

Questions?

Project 3

More JUnit

JUnit practice

- Imagine you've got a class that stores time

```
public class Time {
        private int hour;
        private int minute;
        private boolean am;
        // Methods
        public Time(int hour, int minute, boolean am) {}
        public String toString() {} // Example: "3:06 pm"
        public int getHour() {}
        public int getMinute() {}
        public boolean isAm() {}
        public void addMinutes(int minutes) {}
        public void addHours(int hours) {}
}
```

- What are good tests for it?
- Let's write at least four JUnit tests for it

Debugging

Debugging jargon

- Debugging has some jargon:
  - A **failure** is a deviation between actual behavior and intended behavior
  - A **fault** is a defect that can give rise to a failure
  - A **trigger** is a condition that causes a fault to result in a failure
  - A **symptom** is a characteristic of a failure that can be observed
  - An **error** is something a person does (or doesn't do) that gives rise to a fault

Debugging example

- Specification: Write a program to read an **`int`**, divide 10 by that value (with integer division) and display the result (or an error message if the input is 0)

Bad Code 1

```
int value = in.nextInt();
int result = 10/value;
System.out.println(result);
```

Trigger: Entering 0
Symptom: Crash
Fault: Leaving out a check

Bad Code 2

```
int value = in.nextInt();
if (value != 0) {
    int result = 10/value;
    System.out.println(result);
}
```

Trigger: Entering 0
Symptom: No error message
Fault: No output

Debugging

- **Debugging** is using trigger conditions to identify and correct faults
- Steps of debugging
  1. **Stabilize:** Understand the symptom and trigger condition so that the failure can be reproduced
  2. **Localize:** Locate the fault
     - Examine sections of code that are likely to be influenced by the trigger
     - Hypothesize what the fault is
     - Instrument sections of code (with print statements or conditional breaks)
     - Execute the code, monitoring the instrumentation
     - Prove or disprove the hypothesis
  3. **Correct:** Fix the fault
  4. **Verify:** Test the fix and run regression tests
  5. **Globalize:** Look for similar defects in the rest of the system and fix them

Debug code

- **Debug code** is temporary output and input used to monitor what's going on in the code
- Instead of printing out just numbers, add context information so that the debug statements are clear
- Debug code is quick and dirty, useful when setting break points and tracing execution with a debugger might be too much work to catch a small issue
- There are logging tools that can print logging data at various levels
  - Normally, nothing prints out
  - Running the program in logging mode prints out important data
  - Running the program in verbose mode prints out everything it can
- Debug output can go to **`stdout`** or **`stderr`**
  - **`System.err`** (instead of **`System.out`**) prints to stderr in Java

Debuggers

- IntelliJ, Eclipse, Visual Studio, gdb and most fully-featured IDEs provide debugging tools
- Typical debugging features:
  - Setting **breakpoints** that will pause execution of the program when reached
    - Breakpoints can often be **conditional**, pausing only if certain conditions are met
  - Executing lines of code one by one, **stepping over** method calls or **stepping into** them and **stepping out** when you're done executing its code
  - Setting **watches** that display the current state of variables and members
- If you don't use your debugger, you're choosing to play the game with one hand tied behind your back

Performance optimization and tuning

- Students often rush to try to optimize before their code really works well
  - "Premature optimization is the root of all evil."
- Performance is important, however, especially for systems software
- Guidelines:
  - If a loop knows the answer before it's done iterating, use a condition to stop it iterating as soon as it knows
  - Instead of doing a computation repeatedly inside a loop, do the computation once outside (if possible)
- **Profiling** tools show which methods (or sometimes even which lines) take the most time to run
  - Often, our intuition about which lines are costly is wrong

Refactoring

Refactoring

- **Refactoring** means changing working code into working code
- It can be done to improve the structure, the presentation, or the performance
- You should refactor when:
  - There's duplication in your code
  - Your code is unclear
  - Your code smells:
    - Comments duplicate code
    - Classes only hold data (instead of operating on it)
    - Information isn't hidden
    - Classes are tightly coupled
    - Classes have low cohesion
    - Classes are too large
    - Classes are too small
    - Methods are too long
    - `switch` statements are used instead of good object-orientation

Common refactoring actions

- Renaming a variable or method
- Adding an explanatory variable
  - If an expression is too long, storing a partial computation into a named variable can help it be understood
- Inline temporary variable
  - If a temporary variable is useless, just use the full expression (the opposite of the previous)
- Break a method into two methods
- Combine two short methods into a single one
- Replace a conditional with polymorphism
  - Instead of an if or a switch, behavior changes because different objects have overridden methods with different behavior
- Move methods from child classes to parent classes

Keeping refactoring under control

- Changing working code always risks breaking it
- To avoid problems, follow these steps:
  1. Run tests and make sure they all pass before refactoring
  2. Identify a refactoring
  3. Make a small change that moves closer to the fully refactored version
  4. Run tests and fix what's broken
  5. If the refactoring isn't done, go back to step 3
  6. If the code still needs refactoring, go to step 2

Testing and coding

- Waterfall views system testing as a phase after coding
- Unit testing **must** happen during coding, using (some) clear box criteria
  - Small units of code must be exercised thoroughly
  - Only the developers themselves have the knowledge to do that
- Waiting to test after development has problems
  - Few tests are written because "the job is done" and coders don't want to find mistakes
  - Poor tests are written because coders are focused on whatever they were just writing

System Testing

System testing

- **System testing** is testing of the whole product
  - Both unit testing and integration testing of individual classes and larger components should have been done by now
  - Testing both functional and non-functional requirements
- System testing is necessary because:
  - There could still be faults in the components
  - Some things can't be fully tested without all the pieces together
- **Alpha testing** is the first stage of system testing
  - Developers test behavior similar to what real users would do
- **Beta testing** has real users testing the product

Differences between system and sub-system testing

- Unit tests (and some integration tests) are done by developers on their own code, but system tests can involve code written by different people and teams
  - Maybe there are misunderstandings between the teams of the format of input and output
- Sub-system testing can use clear-box and black-box testing, but system testing typically uses only black-box
  - Testers might only know the requirements, not the implementation
- One system test might test the system in several different states while sub-system tests tend to be more narrow
- System testing is focused on user interaction, so the nature of the product (e.g., web vs. desktop application) matters less
- Systems testing involves more people and is more likely to involve one team blaming another

Details of system testing

- Alpha testing and the two phases of beta testing are similar, but there are some details that are different, summarized in this table

	Alpha Testing	Beta Testing	
		Acceptance Testing	Installation Testing
Personnel	Testers	Users	Users
Environment	Controlled	Controlled	Uncontrolled
Purpose	Validation (Indirect) and Verification	Validation (Direct)	Verification
Recording	Extensive Logging	Limited Logging	Limited Logging

Alpha testing

- Alpha testing should validate that the product meets user needs and verify that it does so correctly
- Validation is usually indirect because it doesn't have real users
  - The software requirements specification or other documents are used to check user needs
- Teams independent from the developers are often used for alpha testing
  - Developers have a bias toward *not* finding tests at this point
- Alpha testing happens in a controlled environment that tries to simulate the real environment
  - Failures can be isolated and recorded
  - The product might be in a mode that does more logging than normal

Functional alpha testing

- Functional alpha testing is based on the requirements listed in the product specification
- To isolate failures, basic functionality is tested before more complex functionality
- **Operational profiles** give information about how often different use cases come up and the typical order of use cases
  - Using these profiles, testers can make tests that simulate typical usage

Non-functional alpha testing

- Some non-functional requirements are development requirements
  - Cost of the product
  - Time the product takes to be made
- Development requirements generally can't be tested, but there are many kinds of non-functional execution requirements that are testable
- Common non-functional execution tests:
  - **Timing tests** time the amount of time needed to perform a function, sometimes using **benchmarks**, standard timing tests
  - **Reliability tests** try to determine the probability that a product will fail within a time interval: mean time to failure
  - **Availability tests** try to determine that probability that a product will be available within a time interval: percent up time
  - **Stress tests** try to determine **robustness** (operating under a wide range of conditions) and **safety** (minimizing the damage from a failure)
  - **Configuration tests** check the product on different hardware and software platforms

User interface tests

- Some user interface tests straddle the line between functional and non-functional
- Tests that check the user interface are called **usability tests** or **human factors tests**
- **Internationalization or localization tests** are a kind of usability test that check translations and other cultural information like currencies and the formatting of numbers, times, and dates
- **Accessibility tests** check whether the user interface works for all people, even with significant disabilities
  - There are guidelines for the kinds of disabilities that need support (low visual acuity or color blindness)
  - Testing often involves measuring the time needed to perform tasks

Beta testing

- Beta testing uses external testers, usually users from the population who will use your product
- These users have the duty to record and report failures
- Acceptance testing is a kind of beta testing done by clients to validate that the product meets their needs
  - Done in a controlled environment, like the one alpha testing was done in
- Installation testing is a kind of beta testing using real users in uncontrolled environments
  - Instead of validation, the goal is to verify that the product works properly in a (more) real environment
  - Installation testing can be inefficient, since the users often do not give the most detailed feedback

Testing in traditional processes

- Testing revolves around the software requirements specification
  - Some requirements will be tested with unit tests
  - Others will be tested with system tests
  - Very few requirements are tested with integration tests
- Unit testing starts early in the implementation phase
  - It needs to be good, since other testing doesn't happen until much later
- Since integration testing happens much later, placeholders for methods that will be needed in the future, called **stubs**, are common
- Alpha testing can't start until the end of the traditional process
  - There isn't a working product until then

Test plan

- In traditional processes, a **test plan** is used to map out system testing
- Test plans include:
  - Business and technical objectives of the test suite
  - Test cases
  - Review process and acceptance criteria
  - Estimate of the size of the testing effort
  - Schedule for the testing effort

Testing in agile processes

- Scrum (like other agile processes) doesn't have a formal requirements specification
- Integration testing and alpha testing happen every sprint
- Developers do unit testing, which is supposed to find implementation failures
- Integration and system testing are supposed to find design failure
- Another approach is to tie conditions of satisfaction for each PBI to unit tests

More on agile testing

- Some agile processes use **daily build verification tests**
  - The product is automatically built and tested every day
  - These tests are also called **smoke tests**
  - These tests require developers to test their own code carefully before committing it or risk breaking the whole daily test
- Since finished products are coming out all the time, beta testing is only used on some sprints
  - Sometimes only on fully-featured versions called **release candidates**

Test execution tools

- Software that does not have GUI can be tested using tools that are the same or similar to unit testing tools
  - Sometimes custom test harnesses must be created
- Software with a GUI is much harder to test
  - There are tools to record a series of mouse clicks, key presses, and other interactions and play them back
  - Other tools allow interactions to be described as scripts
  - Yet other tools record the interactions *as* scripts
  - But what happens when you change a button location?

Test recording and reporting tools

- Tools called **bug tracking systems** allow testers to record, track, and report test results
- These tools are built into many modern development tools
- GitHub has an Issues page for each repository
  - You can open an issue when you find a bug
  - Someone can close the issue when a commit fixes the bug
- Sometimes the issues in bug tracking systems are referred to as tickets
  - A developer or a user (even a member of the public) can open a ticket when a bug is found
  - The ticket usually allows for discussion of the bug
  - Sometimes the ticket is assigned to one or more developers to fix

Preparing for system testing

- Some design patterns make it easier to do system testing
  - The Command pattern is used to hold actions, so Command objects can be issued directly instead of waiting for them to be triggered by GUI events
  - The Observer pattern and the Proxy pattern can also be useful
- Some libraries have built-in logging support that makes it easier to instrument tests
- Using exceptions (instead of returning error codes) can also provide flexible ways to build tests

Upcoming

Next time…

- Deployment, maintenance, and support next Monday
- We'll also do review for Exam 2

Reminders

- Read Chapter 11: Deployment, Maintenance, and Support for next Monday
- Work on Project 3
- **Study for Exam 2**
  - **Next Wednesday**