Software Life Cycle

1. Analysis: analyze if a solution is possible and if so come up with a requirements specification
2. Design: A program design for the requirements specification – can be a set of objects/classes
3. Coding: the actual implementation of the design in a particular programming language
4. Testing: unit and integration testing, debugging (activity to eliminate a program error)
5. Operation: software is put into actual use, includes maintenance, added features requested by the customer, cleanup of newly discovered bugs

Design and Test

- Coffee.java
  - a small program with problems
- At: http://www.cs.rit.edu/~jdb/cs1/

Pragmatic Programmers

Refactor

- When you take a program and redesign ruthlessly to make future maintenance easier.
- Why does Coffee.java need to be refactored?

Reduce Redundancy

- How do we reduce redundancy?
  - What happens if we change the price of Coffee?
  - How do we split the program up to make it more flexible?

Testing and Debugging

- Testing: the process of finding errors in the system implementation
- Debugging: the process of finding the source of errors and fixing such errors
  - Testing is done before debugging

Why Test?

- The purpose of testing is to identify implementation errors before the product is shipped
- The errors may be:
  - Actual bugs in the code
  - Incorrect implementation of the requirements or functional specifications
    - Misunderstandings
    - Incomplete requirements or functional specifications
What Testing is Not

- Testing is not a random process
- Testing is not debugging
  - Testing identifies the problem
  - Debugging finds the location of a problem and fixes it.

Who Does Testing?

- Multiple people
  - Programmers
  - A testing team (especially for system level testing – also called integration testing)
  - The customer (alpha and beta testing)

Test early. Test often. Test ruthlessly.

- Test when writing code or even before
- During the implementation phase
  - Programmers are responsible for testing their unit to ensure the code meets the design and functional specifications
  - As multiple units become available and can be combined, integration testing can begin

Testing is like fishing with a net

1. Unit tests: Use fine-grained nets to catch the minnows (small bugs)
   - Tests single classes
2. Integration tests: Use large-grained nets to catch the killer sharks
   - Tests how classes work together

Other kinds of testing

1. Validation and Verification:
   - Does the program meet the functional specifications of the system?
2. Resource exhaustion, errors, and recovery
   - How does the program behave with real world memory, disk space, video resolution, etc
3. Performance testing
   - Does the software meet performance requirements under real-world conditions
4. Usability testing
   - Involves human factors and making sure that the customer can actually use the system

How tests are passed

- A system passes the tests if it produces results that are consistent with the functional specification and requirements
  - The program does what it is supposed to do and doesn’t do anything it’s not supposed to.
Types of testing

Formal Verification

- Formal Verification: a process that uses mathematical and logical assertions to prove that the program is correct
  - Very difficult to do

Empirical Testing

- Empirical testing: the process of generating test cases and running the tests to show that errors exist.
  - Involves observing the results of using the system
  - Can only prove that an error exists. Can not prove that there are no errors.

Types of Testing

Why you can’t prove there are no errors

```java
public int test(int a, int b) {
    return a / (a+b);
}
```

- Say that each integer can be a value from 0 to 999
- In theory, this 3 line function has 1 million logical states, 999,999 of which will work correctly and 1 of which will not
- “The sun will be a cold hard lump before you can test all states in your program.”
  - comment from the Pragmatic Programmer

Damage Limitation

- If exhaustive state testing on 3 lines of code yields that many tests, imagine testing for all possible states in a class with public instance variables!
- Limit possible damage caused by outside classes by declaring your instance/class variables to be private.

White Box Testing

- Requires access to the actual implementation code
- Requires development of test cases that will exercise each unit of the system and possible “flows” through the system based upon the actual implementation
  - All statements, decisions, conditions, and all inputs
- This type of testing is not very practical, but sometimes is required

Black box testing

- Typically a testing team develops use cases based upon the requirements and functional specification without looking at the actual implementation
- Tests valid and invalid inputs, but can not possibly test all inputs
  - Must determine what subset of inputs provide sufficient coverage
- You want to be devious and imaginative in trying to break the system with black box testing
Black Box Testing

- Equivalence partitioning
  - A set of inputs that are processed identically by the program
    - Legal input values
    - Numeric/non-numeric values
- Boundary testing
- Error guessing

Debugging

- It is a painful thing
  To look at your own trouble and know
  That you yourself and no one else has made it
  - Sophocles, Ajax

- Embrace the fact that debugging is just problem solving, and attack it as such.

The debugging mindset

- If your first reaction upon hearing of a bug is that “it’s impossible”, then you are plainly wrong. Don’t waste a single neuron on that train of thought.
- Don’t assume it – prove that it works correctly
- The problem is normally in your code, not in the compiler or the OS

Who debugs

- Not the customer
- If there is a testing team responsible for testing the system, this team will also attempt to precisely identify the problem and report it to the appropriate programmer.
- The programmer is responsible for determining the actual problem and repairing it.

Debugging

- Should be a formal process of attempting to narrow down the location of the problem and then identifying the problem
  - Does not mean simply changing code until the problem goes away!
  - Requires thinking about what the problem might be

Methods of determining the location of a bug:

- Use extra output statements in the program to trace the program execution
- Use a debugger to trace the program execution (jdb is a sorry excuse for a debugger)
- Possibly write special test code to exercise parts of the program in special ways that will allow you to better understand the error.
- Rubber ducking
More on Debugging

- Potentially test a certain range of values to see which ones fail
- Attempt to eliminate parts of the program as the problem, thus narrowing the search
- Check that the data is valid
- Many times, the location where you see the first instance of the bug is not the source of the bug

Fixing Bugs

- Fix only one bug at a time and then rerun the same exact tests
  - Changing multiple things makes it difficult to identify which change caused the behavior change
  - If the problem appears to be fixed, still run a full test suite to ensure that the “fix” didn’t break something else

More General Rules

- Steps for systematic testing
  - Test incrementally by writing part of the system, test it, then write some more code, test that code, etc.
  - Test the simple parts of the system first
  - Know what output you are expecting

Debugging exercise

General Rules to Follow

- Test your code as it’s written (or even before)
  - Test code boundaries
  - Test pre- and post- conditions
    - The necessary or expected properties before and after the code is executed
  - Program defensively by adding code to handle the “can not happen” cases
  - Check error returns

General Rules

- Ensure that testing covers every statement of the program
  - Every line of the program should be exercised by at least one test
We should test the test code

```java
public class TimeCounter {
    private int time;
    private static final int SEC_TO_TSEC = 60;
    private static final int MIN_TO_SEC = 600;
    private static final int HOUR_TO_SEC = 36000;

    public TimeCounter(int tenthSecs) {
        time = tenthSecs;
    }

    public int getTime() {
        return time;
    }

    public int getTSeconds() {
        return ((time % HOUR_TO_SEC) %
                MIN_TO_SEC) / SEC_TO_TSEC;
    }
}
```

How would you test this?

```java
public class TimeCounter {
    private int time;
    private static final int SEC_TO_TSEC = 60;
    private static final int MIN_TO_SEC = 600;
    private static final int HOUR_TO_SEC = 36000;

    public TimeCounter(int tenthSecs) {
        time = tenthSecs;
    }

    public int getTime() {
        return time;
    }

    public int getTSeconds() {
        return ((time % HOUR_TO_SEC) %
                MIN_TO_SEC) / SEC_TO_TSEC;
    }
}
```