Testing

Importance & Techniques
(for your project and job)
Testing: Why?

• It is quite, quite difficult to write code completely correctly on the first attempt.

• Having incorrect code decreases program usability at minimum, and can often be quite costly.
  – We all “love” it when a program hangs, or when we get the “blue screen of death.”
Testing: Why?

- The **first flight** of the ESA Ariane 5: $1 billion in damages.
  - At its core, the error was triggered by an arithmetic overflow exception.
  - The affected software component was reused from the Ariane 4 system without any re-examination of the system requirements.
  - The difference in flight path between the two rockets led to the arithmetic overflow.
The **first flight** of the ESA Ariane 5: $1 billion in damages.

- “An underlying theme in the development of Ariane 5 is the bias towards the mitigation of random failure. The supplier of the SRI was only following the specification given to it, which stipulated that in the event of any detected exception the processor was to be stopped. The exception which occurred was not due to random failure but a design error. The exception was detected, but inappropriately handled because the view had been taken that software should be considered correct until it is shown to be at fault.”
Testing: Why?

- Other known software catastrophes:
  - Mars Climate Orbiter - wrong units
  - 1999
  - Mars Polar Lander – engine cutoff too early, SW bug
  - 1999
  - Mars Rover – too many files open
  - 2004
Testing: Why?

- Other known software catastrophes:
  - Misuse of medical diagnosis SW
  - 5-8 deaths – 2000
  - Y2K bug
  - Mars Global Surveyor – battery failure due to SW error
  - 2006
Testing: Who?

• All too often, we’d like to assume that our code “simply works.”

• Unfortunately, that is rarely the case, even for experienced programmers.
  – Even methods one or two lines long can have errors at times.
  – How many times have we made accidental sign errors on math assignments?
Testing: How?

- What makes testing hard?
  - One classic approach – “why not just try it out and see if it works?”
  - The most thorough variety: brute-force testing.
  - The problem with this?
    - public f(int x, int y) can have many different combinations of values that would have to be tested!
Testing: How?

- The thing is... the behavior for a method typically is the same for large subsets of possible inputs.
  - Aside from pseudo-random number generation methods, nobody writes a method with random behavior.
Testing: How?

- While there is a very large range of potential values a method may have to handle, there should typically be a small number of sets which would have different behavior, at least for smaller, localized methods.
  - Ideally, then, we’d like to be able to identify subsets in which every value behaves the same.
Testing: Where?

- Problem: knowing which input values correspond to each behavior set requires perfect knowledge of the method.
  - Is this always possible?
  - Often, we’ll have to make guesses.
Testing: Where?

• For example, consider the absolute value function.

```java
public int abs(int x)
```

• Natural behavior sets we’d expect:
  - \( \{x : x < 0\} \)
  - \( \{x : x \geq 0\} \)
Testing: Where?

- So, one might naturally test with one value from each set there.
  - Say, \{-4, 4\}.
  - This would definitely work... whenever the method is written correctly.
  - But what if the programmer didn’t actually write the method properly?
Testing: Where?

```c
int abs(int x)
{
    if(x < -1) return -x;
    else return x;
}
```

- Sometimes it’s easy to be one value off on boundary conditions when implementing methods.
Testing: Where?

```c
int abs(int x)
{
    if(x < -1) return -x;
    else return x;
}
```

• In this case, there would be exactly one input value with an incorrect output: -1.
Testing: Where?

• Therefore, it’s often vital to test at the edges of what one would expect to be the boundaries of behavior sets.

  – Catches:
  – Unimplemented cases
  – Off-by-one errors
  – Arithmetic overflow
Black Box Testing

- One paradigm for testing is that of the “black box.”
  - Not quite the airlines’ black boxes, though.
Black Box Testing

- The “black box” testing paradigm takes the code to be tested as if it were opaque.
  - Testing patterns designed for this paradigm thus do not require any examination of the code.
  - Black box testing seeks to test each possible behavior of a particular method, given the nuances of its specification.
Black Box Testing

// Finds the first index such that
// a[index] == val.

int indexOf(int* a, int val, int len)

• What would be possible execution
  paths for this method?

• What are important possible situations
  which should be examined?
Black Box Testing

// Finds the first index such that
// a[index] == val.
int indexOf(int* a, int val, int len)

- a[] = {3, 4, 5}, val = 4
- a[] = {3, 5, 6}, val = 4...
- a[] = {3, 4, 4}, val = 4
- Finds the first index...a
Black Box Testing

- Other potential error sources to test for:
  - Null reference on input?
  - Circular referencing within a structure?
  - Aliasing – the same object/referenced used for two+ parameters of a method.
Black Box Testing

• Since black box testing is done without examining the code, it has a few advantages...
  – No bias in the code’s implementation will carry over to tests.
  – The tests won’t need to be rewritten whenever the code is implemented differently.
White Box Testing

- A different testing paradigm is that of white box (or glass box) testing.
  - This paradigm is designed to operate with knowledge of the implementation of coded methods and objects.
  - It’s more prevalently done for control flow and for features not explicitly part of a program’s specification.
White Box Testing

- A different testing paradigm is that of white box (or glass box) testing.
  - If an object has multiple code paths within it designed to help with differing areas of execution, white-box testing is useful to ensure that each code path is fulfilling its requirement.
Testing: Where?

- Whenever you do find a bug in code, make sure to save the error-generating input for later tests.
  - If the bug was made once, it’s easily possible for similar errors to happen again later upon object reimplementation.
Testing: When?

- The more frequently you test, the better off your development is likely to be.
  - As feasible, of course, since testing every single additional line of code is likely overkill, and may not even add functionality with clearly visible results that can be tested.
Testing: When?

• If possible, don’t even wait for a complex object or method to even be completely written before testing it.
  – White box testing – test the parts that you have written to see if the “method so far” is reasonably correct before proceeding to complete it.
Testing: When?

• When possible, plan your path of implementation such that it is reasonably possible to test your code earlier, rather than later.
  – The earlier you test, the sooner you can catch bugs in your program, and the easier it is to isolate them and fix them.
Testing: When?

- If you make a change in the implementation of a method or object, re-test it!
  - It’s easy to accidentally reintroduce old errors – keep those test cases around for re-trials.
Testing: When?

• When and where possible, aim for automated testing.
  – If you can automate some of your tests, it becomes a lot less tedious to track down bugs in the long run.
Testing: How?

• Whenever testing your code, be thorough and have a clear plan.
  – Haphazardly testing different aspects of your code randomly is not the best strategy.
  – It’s easy to accidentally forget to test critical aspects of your program without keeping track of what has and has not been tested!
Testing: How?

• Use the project’s specifications to determine a thorough set of tests.
  – This helps to ensure a higher-quality product...
  – It also helps to clarify the specifications sometimes.
  – Thoroughly examining them may reveal critical but ignored details of the spec.
  – There could be missing aspects of the specifications which need to be determined.
Testing (unless brute force... and good luck there!) cannot prove total and complete correctness of a program.

- However, the more testing is done, the higher quality that the final product will be.
- It’s very important to be thorough and have a solid strategy.