TESTING

Today

- Today we will look at software testing.
- This material isn't covered in the textbook, but is an important topic nonetheless.
- It is an important part of making sure that you deliver software that works as it is designed to work.
- (There is another topic, which we won't cover here, which is making sure you deliver software that is designed to do what the customer wants.)
- Here we touch on the boundary between knowing how to program, and *software engineering*.

- First, let's consider what you want to achieve by testing.
  - What is the aim?
- One thing to aim for is that you achieve what is called *statement coverage*.
- That means making sure every line in the program has been executed.
- That sounds like a good idea, but achieving statement coverage is not all that is needed.
- Why not?

- Consider this:

```
int returnInput(int x, int y, int z){
   if(x > 2){
      z++;
   }
   if(y > 3){
      z--;
   }
   if((x > 2) && (y > 2)){
      z * z;
   }

   return z;
}
```

- If we aim to execute each statement, we might call the function with `x` having the value 3 and `y` having the value 4.
- That will make each line of code execute and we'll return the same value of `z` as was passed to the function.
  - We'll assume that is what we wanted.
- However, that might not allow us to spot that `z`'s value will alter if the first condition is true and the second is false.
  - We'll assume this is a bad thing.
- The point? Just looking at positive conditions and focussing on executing every line of code doesn't mean we check all the functionality.
- We need to do more.

- One way to improve on statement coverage is to make sure we execute all the different branches of the code.
  - *branch coverage*
- Each `if` has two branches (where it is true, and where it is false).
- Each `switch` has as many brances as there are `case` statements.
- The above example has six branches that need to be covered — each condition has to be evaluated so that it is both true and false.

- We can extend the notion of branch coverage bythinking about all the paths through the code.
- Our example above has eight paths, which correspond to the following combinations of conditions:

first	second	third
true	true	true
true	true	false
true	false	true
true	false	false
false	true	true
false	true	false
false	false	true
false	false	false

- The advantage of path coverage over branch coverage is that it tests all the conditions independently.
- This may happen with branch coverage, but it *must* happen in path coverage.

- Note that path complexity is *exponential* in the number of decisions.
- Code with `n` binary decisions will have 2^n paths.
- This means that it is often not possible to achieve path coverage.
- Note that some of the possible paths may be *infeasible*, meaning that they cannot be executed.
- For example, in our 3 condition case above, if the last condition included `y > 3` rather than `y > 2`, it would be impossible to make that condition false while the first two conditions were true.
- For that reason it would be nice to have a program that would always identify infeasible paths.
- But such a program is *impossible*.

- BTW, the exponential number of paths is another argument for making sure functions are small.
- 3 functions with a single conditional in each would have 6 paths, not 8 (two for each function).

- There are other forms of code coverage that we can aim for.
- These include:
  - *function coverage*, making sure each function has been called.
  - *entry/exit* coverage, making sure all possible call of the function and possible return have been executed.
- Some of these forms of coverage are related to one another.
- If you have path coverage, for example, then you also have condition, and statement coverage, and all the interesting cases of entry/exit coverage.
- However, as we showed above, if you have statement coverage, you may not have decision or path coverage (depending on what decisions you have).

- Once you have decided what type of coverage you want to achieve, you have to decide on a set of *test cases*.
- This are the tests that you will apply to your code.
- How do you do this?
- Well, to establish path coverage, you need to:
  1. identify all the `if/else` and `switch` statements.
  2. identify the parameters in the conditions in all those statements.
  3. pick parameter values that ensure every combination of the conditions is triggered.
  4. make one case for each set of parameter values.

- The kind of testing we have been considering so far, which needs access to the source code of the program being tested, is known as *white-box* or *glass-box* testing.
- Another approach is *black-box* testing, where the tester doesn't see the source code.
- Instead they might work from the specification.
- This is an approach that allows the testing to be done by someone other than the developer.
- (The testing may even be done by people who don't know how to program).

- One way to construct black-box tests is to create four classes of test, tests using different kinds of data:
  - Easy-to-compute data.
  - Typical data
  - Boundary, or extreme, data.
  - Bogus data
- Let's look at examples of these types of data.
- To do that we will consider computing the solutions to the quadratic equation:

$$ax^2 + bx + c$$

- If we remember our high-school mathematics, we know — without looking at the code — that the code has to compute:

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

- In this case, easy to compute data could be data that makes the square root easy to calculate.
- Data like:

a	b	c	roots
1	2	1	-1, -1
1	3	2	-1, -2

- The easy to compute data gives us a quick check that the code seems to be functional.

- Typical data might be:

a	b	c	roots
1	4	1	-3.73, -0.27
2	4	1	-1.70, -0.29

- This data checks that the program really is doing the right calculation.

- Good boundary data will often check that zero, in some form or other, can be handled.
- Here some good boundary data would be:

a	b	c	roots
2	-4	2	1, 1
2	-8	8	2, 2

- This data makes the bit under the square root (the discriminant) zero.

- Finally, bogus data really goes all out to break the program with values that it shouldn't really have to handle.
- Suitable bogus data would be:

a	b	c	roots
1	1	1	square root of a negative number
0	1	1	division by zero

- To handle these values gracefully (as a good program should), the code will need some exception handling.
- Thus there is a direct tie in between exception handling and testing.

- All the testing we have described so far will usually be applied to relatively small pieces of code.
- As in the hoeworks you have written small test programs for each class as you develop it.
- This is known as *unit testing*.
- Each unit test will involve writing a program with a `main` which creates an instance of the class, and tests its methods.
- Each method should be tested using some set of cases that either incorporates the different classes of data we just covered for balck-box testing, or attempts something like path coverage.

- The need to write such programs is another reason for using `make`.
- Though they don't provide very good coverage you can think of `testpoint` and `testlist` from the previous lecture as (poor) examples of unit tests.

- As you put units/classes together, you will often find that as you write new classes, unit testing them exposes problems in earlier classes.
- So you go back and fix the earlier classes.
- Once you have fixed them, you need to perform a *regression test* which checks that in fixing the problems, you haven't also introduced new problems.
- The regression test will include the old unit test (usually with additional test cases).
- So you don't throw away your old unit tests.

Summary

- This lecture described some aspects of testing.
- First we talked about some aspects of *code coverage*:
  - statement coverage
  - branch coverage
  - path coverage

  and discussed how to achieve these forms of coverage.
- We then distinguished between black-box and white-box testing and showed how to develop black-box tests.
- Finally we talked about unit and regression testing.