Software Development 5: Testing

1. Introduction
2. Incremental Software Development
3. Incremental Development Example
4. Software Bugs
5. **Testing**
   - **software testing**
   - **test cases and test suites**
   - **adequate test suites**
   - **error testing**
   - **automating testing**
   - **limits of testing**
6. Debugging

Software Testing

Testing is a critical component of software development.

Software testing involves execution of a software system or one of its components, for the purpose of evaluating whether the unit under test meets the *software requirements specification*.

This generally means evaluating whether the unit under test satisfies one of more of the following properties:

- accepts the required range of invocations and inputs
- produces the correct input-output behavior
- has desired/acceptable user interface
- handles input/user errors appropriately
- acceptable performance (CPU time, memory, etc.)
- runs in intended CPU/OS environments

Software Testing (contd.)

A range of **software testing methodologies** have been developed.

See *"software testing"* on Wikipedia for more information.

The manner in which testing enters into the development process depends on the development approach being used.

This is particularly true when software is being built by a *team* of developers.

Large software projects frequently employ people whose only job is testing the code being developed.

Testing will then tend to be highly automated, typically supporting both **unit testing** and **regression testing**.

Software Testing Terms

We will concentrate on testing issues appropriate for any developer.

Here are a few key testing terms one should know:

- **test case** – data/call that has particular characteristics
- **test suite** – set of all test cases for program
- **test script** – program to automate use of test suite
- **test harness** – all testing tools and data for a program
- **black box** – testing without code knowledge (API only)
- **white box** – testing while aware of code (e.g., branches)
- **code coverage** – refers to lines of program code tested
- **edge/boundary case** – case with min/max value for a parameter
- **corner case** – usually means error case (beyond boundaries)
- **regression** – bug introduced as result of fixing another bug

Test Cases

A **test case** involves a single *invocation* of a program and/or a particular *input sequence* for an interactive program.

Program invocation involves:

- the combination of options, if any
- the number of arguments, if any
- the characteristics of each of the arguments
- whether call is valid or invalid

Input sequences can involve:

- input values (e.g., particular filenames or numeric values)
- input characteristics (e.g., length of or characters in inputs)
- single vs. multiple input iterations
- termination of input sequence/iteration
- valid or invalid inputs

Test Suites

The set of all *test cases* for a unit is known as the **test suite**.

The goal of a *test suite* should be to ensure that the unit under test works *"as it is supposed to."*

In other words, that it does *not* contain any *bugs!*

Developing a test suite that is adequate to meet this goal is not easy with programs of any complexity.

Experience has shown that students tend to do a very poor job of meeting this goal.

They develop only one or two *test cases*, and consider themselves done when their program works on this small number of cases.

Adequate Test Suites

In general, the only way to be certain a program is bug free would be to test it on *every possible input* (and be able to check its output is correct).

That is obviously *impossible* for all but a trivial ("toy") program.

A program that takes files as input, for example, must be able to deal with an effectively *infinite* number of possible inputs.

Even when a program takes only a large finite number of possible inputs, there would seldom be any way to check outputs.

Doing so would require a program that is known to be correct, but then why would you need another program?

Adequate Test Suites (contd.)

Luckily, there are a few relatively straightforward methods that can help one design a set of test cases that provide reasonable assurance a program is (fairly) bug free.

The testing design approaches we will cover are:

- **equivalence class testing**
- **boundary value testing**
- **code coverage testing**

Equivalence Class Testing

Equivalence class testing is also known as: **equivalence testing**, **equivalence partitioning**, and **equivalence class partitioning**.

The basic idea is that the set of all possible program inputs can be **partitioned** into *subsets*, such that if the program correctly handles *one* input from a subset, it should handle *all* the rest.

In other words, while there may be a large/infinite number of inputs, there will be only a *finite* number of **classes** of inputs.

By testing at least one input from each class, you will have provided (some) coverage of all possible inputs.

The concept of input classes is related to the mathematical notion of **equivalence classes**.

Equivalence Class Testing (contd.)

With simple programs, it may be fairly easily to identify input classes—e.g., see the Wikipedia article "equivalence partitioning."

Identifying input classes is also easier when doing *white box* or *grey box* testing—i.e., when one has access to program code.

Consider the following function:

```
int compute(int x)
{
  if (x <= 0)
    ...code fragment A...
  else if(x > 0 && x < 100)
    ...code fragment B...
  else
    ...code fragment C...
}
```

There are *three* input classes, easily identifiable from the three if-else cases (code fragments A, B, C).

Boundary Value Testing

Boundary value testing (or **analysis**) is related to *equivalence class testing*.

The difference is that instead of testing with *any* value from an *input class*, **boundary/edge case** inputs are chosen.

The reason is that boundary/edge case values are more likely to reveal bugs than would values in the "middle" of an input class.

Identifying boundary values will be easier with *white box* testing, by looking at conditional and loop conditions.

E.g., edge cases for `x` in the above function would be:
(class 1) `0`; (class 2) `1` and `99`; (class 3) `100`.

Code Coverage Testing

Code coverage testing refers to testing that aims to test every one of particular "elements" of program code.

A *test suite's* coverage may be evaluated at various levels:

- **functions** – each function in the program gets called
- **calls** – each function call gets executed
- **statements** – each statement in the program gets executed
- **decisions/branches** – each alternative branch of each control structure gets executed (condition made both true and false)
- **conditions** – each Boolean subexpression gets evaluated to true and false (not just condition overall)
- **paths** – each path (sequence of branches) gets executed
- **loops** – each loop body gets executed zero, one, and multiple times

Error Testing

One area that students often ignore is testing their code to make certain it detects and handles invalid inputs and errors.

In other words, they fail to test that their program does not contain **error handling errors**.

One aspect of error testing is calling the program with **corner cases**—i.e., values that are beyond the boundaries for valid values.

For example, if your program need handle input file names of at most 100 characters, you need to test with file names of 101+ characters to make certain invalid inputs are caught and dealt with appropriately.

Error Testing (contd.)

Unfortunately, testing some errors can be difficult because the error conditions cannot be easily produced.

E.g., `fgetc()` will return `EOF` for both file-end and an *error*.

How can one force an error return from `fgetc()` (with valid code)?

Getting an i/o error would require something like having the file on an external storage device and pulling the plug in the middle of looping reading—not really practical.

Often it is easiest to temporarily invert the error check condition in the program—to force the error handling code to be run, to verify that it properly handles errors.

Automating Testing

All but the simplest programs will require a significant number of test cases.

Since any modification to the program could introduce a **regression**, all tests should ideally be rerun after any program changes.

Doing this by manually rerunning the program (and checking its output/behavior) could become extremely tedious and time consuming.

Luckily, it is relatively easy to provide basic automation of testing for many programs via **shell scripts**.

The simplest possible approach is to simply place a sequence of the program calls for testing into a script file.

Automating Testing (contd.)

E.g., a simple testing script might look like:

```
#!/bin/bash
./myprog ...arguments for test case #1...
./myprog ...arguments for test case #2...
./myprog ...arguments for test case #3...
...
```

Create such a file, make it *executable* ("`chmod +x script`") and you can then rerun all your tests by simply doing "`./script`".

Of course you must be able to see if your program is doing the right thing, so it is good to have it *pause* between tests to scrutinize output.

An easy way to do this is to put a `read` command between each `myprog` call, making you type "Enter" between tests.

Automating Testing (contd.)

If you want something a bit more sophisticated, insert lines like this: `echo -n "Hit Enter to continue"; read`

One can even automate the checking of a test cases's output via `diff` or `sdiff`.

With each test, **redirect** program output to a file:

```
./myprog ...arguments for test case #1... > output1
```

Assuming you have the correct output in a file `correct1`, simply then `diff` the two files:

```
if ! diff -q outpu1 correct1 > /dev/null; then
  echo "Program failed on test #1"
fi
```

Automating Testing (contd.)

Some programs are **interactive**—i.e., they take input from the terminal during execution.

These programs can also be automated, by using **input redirection**:

```
./myprog ...arguments for test case #1... < input1
```

and even:

```
./myprog ...arguments for test case #1... < input1 > output1
```

`input1` is a file that contains the sequence of lines that would have been typed as interactive inputs to the program.

Another tool for testing interactive programs is **Expect** (and later relatives).

The Limits of Testing

Testing is not a substitute for thoughtful design and careful implementation.

For all but the simplest programs, *"adequate"* testing is not guaranteed to reveal every bug.

In fact, some bugs are impossible catch with certainty no matter how much testing is done.

For example, bugs due to **uninitialized variables** may manifest themselves only when the computer memory contains particular values.

This is something you have little control over when testing, so such bugs may remain uncaught even after running many tests.

The Limits of Testing (contd.)

Particularly difficult to test are **concurrent programs**: programs consisting of multiple **processes** or **threads** executing "in parallel."

Concurrent programs may contain **race condition** errors, which manifest only under particular process/thread scheduling orders.

Again, this is something one has little/no control over in testing.

Bottom line: don't simply assume testing will catch any bugs so you don't need to be careful in design and implementation!