

Testing

Today we are going to talk about testing. Before you all lapse into comas in anticipation of how exciting this lecture will be, let me say that testing actually is kind of interesting. I can't say that I'm an expert, but it seems (at least to a person that wasn't "brought up" in a culture of testing) that thinking about testing and testability is a pretty interesting way to think about creating software.

So let's start at the beginning, shall we...

Aspects of Quality Software

- Flexibility
- Extensibility
- Correctness
- Robustness
- Reliability
- Safety
- Usability

In the 7 minutes I spent thinking about it I came up with this list of aspects of good software. I think that the items on this list are necessary, if not sufficient, conditions for good systems. These aren't the focus of this lecture, and I just came up with them off the top of my head, but, I need an ego boost so let's go through them and you can pretend like I'm dispensing sage wisdom.

Aspects of Quality Software

- Flexibility – Applicability in novel contexts
- Extensibility
- Correctness
- Robustness
- Reliability
- Safety
- Usability

Example: Google maps – just look at all the mash-ups! This thing started of as a way to get directions.

Aspects of Quality Software

- Flexibility – Applicability in novel contexts
- Extensibility – Ability to add functionality
- Correctness
- Robustness
- Reliability
- Safety
- Usability

Example: Firefox allows for easy addition of functionality through plugins and an API designed for extension.

Aspects of Quality Software

- Flexibility – Applicability in novel contexts
- Extensibility – Ability to add functionality
- Correctness – Consistency with specs.
- Robustness
- Reliability
- Safety
- Usability

I mean this in the absolute sense. The code should map inputs to outputs within acceptable error bounds that are pre-specified.

Aspects of Quality Software

- Flexibility – Applicability in novel contexts
- Extensibility – Ability to add functionality
- Correctness – Consistency with specs.
- Robustness – Resilience to abnormality
- Reliability
- Safety
- Usability

If I enter Hamlet's soliloquy into a text box asking for my after-tax income, my hard-drive should not be formatted.

Aspects of Quality Software

- Flexibility – Applicability in novel contexts
- Extensibility – Ability to add functionality
- Correctness – Consistency with specs.
- Robustness – Resilience to abnormality
- Reliability – Requests handled consistently
- Safety
- Usability

Part of this is that the system responds when we ask it to do something, it doesn't periodically not do something when we ask for no particular reason. This is a bit narrow. What we want is for the system to respond the same way every time we ask it to do something. It needs to be predictable and repeatable.

Aspects of Quality Software

- Flexibility – Applicability in novel contexts
- Extensibility – Ability to add functionality
- Correctness – Consistency with specs.
- Robustness – Resilience to abnormality
- Reliability – Requests handled consistently
- Safety – Avoids dangerous behaviors
- Usability

It should be very, very hard for me to use my cock-pit navigation tools to plot a course through a mountain range. On a less tragic scale, clicking the close box on an application shouldn't allow a close before asking if I want to save my work.

Aspects of Quality Software

- Flexibility – Applicability in novel contexts
- Extensibility – Ability to add functionality
- Correctness – Consistency with specs.
- Robustness – Resilience to abnormality
- Reliability – Requests handled consistently
- Safety – Avoids dangerous behaviors
- Usability – Affords its functionality

Review the slides on Usability for this one.

Ok, so why are we going through these in a lecture about testability? Well, at least four (and you could argue five) of these are directly effected by our ability to test them. What ones are they?

And now for the typical scare tactics: Software bugs *do* cause planes to fall (or get taken) out of the sky. They cause patients to get overdoses and/or not receive critical care. They cause space craft to miss entire planets. They cause the loss of billions of dollars to the economy every year.

And besides, creating software is a craft and software that doesn't at least satisfy all these principles is poorly crafted. And if you don't care about the craft, you are in the wrong business.

So, testing is one way to approach the craft of designing and implementing software.

Verification and Validation

- Validation – Are we building the right product?
- Verification

The system fulfills the informal requirements – it solves the problem.

Verification and Validation

- Validation – Are we building the right product?
- Verification – Are we building the product right?

The system meets the (semi-)formal requirements.

Verification is really about strict, absolute adherence to a formal specification.
Validation is more about the fuzzier customer satisfaction.

Kinds of Testing

- Unit
- Integration
- System
- Regression
- Acceptance

Different kinds of tests have been conceptualized to address the verification-validation spectrum.

Unit tests are meant to test each intended behavior of each module. However, the granularity of the unit being tested can vary and unit test frameworks are often useful for other kinds of testing. We'll come back to this.

Integration testing is all about getting the units to hook up correctly. It's less about functionality and more about plumbing, though you often can't tell anything about the plumbing without a bit of functionality.

System testing is, in some sense, coarse-grained unit testing. It's testing large parts of the system, or the whole system, for functionality.

Regression testing is "testing for change" – you know, like "design for change". As you add functionality to your system and want to keep existing functionality intact, you run a suite of tests after each modification so that you can see when things break.

Acceptance testing is about validation. Does the end product solve the problem it was intended to solve?

Things fall apart...

- Failures
- Faults (a.k.a. bug)
- Errors
- Fault \nRightarrow Failure

So let's look more deeply at verification.

When we do testing, what are we actually looking for? Some people break it down into failures, faults, and errors.

Failures

Observable incorrect behavior of a program.

The program performs conceptually wrong. This does not refer to code. Maybe the spec is wrong...

Faults

Related to the code. Necessary (not sufficient!) condition for the occurrence of a failure.

Errors

Cause of a fault. Usually a human error (conceptual, typo, etc.)

Fault \nRightarrow Failure

Coincidental Correctness

So we are looking for failure and their causes.

What is a Test?

- What is a program?
- Test Suite – some subset of the inputs.
- Test Case – is one particular input.
- Ideal Test – a test case where the correctness of the test implies the correctness of the program for all other inputs.

A program is a function from the space of all possible inputs to the space of all possible outputs.

A “Test Suite” is some subset of the inputs.

A “Test Case” is one particular input.

An “Ideal Test” is a test case where the correctness of the test implies the correctness of the program.

So let's just come up with ideal tests!

Do you think it easy to come up with ideal tests?

In general, it is impossible to define ideal test cases.

Test This!

Triangle 2000

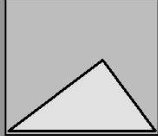
Enter three numbers. These will be treated as the dimensions of a triangle. When you press the "Check" button, the program will display the type of triangle that you specified.

Side A

Side B

Side C

Results



Test this!

- 16 digits & up: loss of mathematical precision.
- 23 digits & up: can't see all of the input.
- 310 digits & up: input not understood as a number.
- 1,000 digits & up: exponentially increasing freeze when navigating to the end of the field by pressing <END>.
- 23,829 digits & up: all text in field turns white.
- 2,400,000 digits: crash (reproducible).

What you didn't try input with 2,400,000 digits!?

- Seduced by what's visible.
- Think they need a spec that tells them the max.
- If they have a spec, they stop when the spec says stop.
- Satisfied by the first boundary (16 digits).
- Let their fingers do the walking instead of using a program like notepad to generate input.
- Use strictly linear lengthening strategy.
- Don't realize the significance of degradation.
- Assume it will be too hard and take too long.
- Think "No one would do that" (hackers do it)

Exhaustive Testing

```
int sum(int a, int b) {return a + b;}
```

How long would it take to exhaustively test this program?

$2^{32} \times 2^{32} = 2^{64} \approx 10^{19}$ tests

Assume 1 test per nanosecond (10^9 tests/second) = we get 10^{10} seconds...

About 600 years!

And this is one of the simpler programs you can think of...

When there is any complexity at all, it is often theoretically impossible to exhaustively test.

Since we probably can't come up with tests that guarantee correctness, we settle for tests that improve our confidence.

What to Test?

- **Blackbox**
 - Based on a functional specification of the software
 - Scales to different granularity levels
 - Cannot reveal dataflow errors
- **Whitebox**
 - Based on the dataflow of the program
 - Does not scale with granularity levels
 - Cannot reveal errors due to missing paths
- **Test Criteria**
 - Whitebox – Cover execution paths
 - Blackbox – Cover input space

Blackbox

Is based on a functional specification of the software

Depends on the specific notation used

Scales because we can use different techniques at different granularity levels (unit, integration, system)

Cannot reveal errors depending on the specific coding of a given functionality

Whitebox

Is based on the code; more precisely on coverage of the control or data flow

Does not scale (mostly used at the unit or small subsystem level)

Cannot reveal errors due to missing paths (i.e., unimplemented parts of the specification)

Test Criteria:

Whitebox: Cover execution paths

Blackbox: Cover input space

Blackbox Testing

- Identify independently-testable features
- Defining all the inputs to the features
- Identify representative classes of values
- Generate test case specifications
- Generate and instantiate test cases

Equivalence partitioning

Identify independently-testable features

Defining all the inputs to the features

Identify representative classes of values

Which values of each input can be used to form test cases (categories, boundary or exceptional values)

A (partial) model may help (e.g., a graph model)

Generate test case specifications

Suitably combining values for all inputs of the feature under test (subset of the Cartesian product---cost, constraints)

Generate and instantiate test cases

Whitebox Testing

- Identify all path in the module under test
- Formulate an input that will cause each path to be traversed
- Generate specifications for test success/failure
- Instantiate tests

Tools for testing

- NUnit
- Build Systems
 - Ant
 - Maven
- Coverage Tools
 - Coverlipse
 - Clover

Demos

Test First Technique

- 1. Write a test that expresses an intent of a class in the system
- 2. Stub out the class (enough to allow the test to compile)
- 3. Fail the test (don't skip this)
- 4. Change the class just enough to pass the test
- 5. Pass the test
- 6. Examine the class for coupling, cohesion, redundancy, and clarity problems. Refactor.
- 7. Pass the test
- 8. Return to 1. until all intentions are expressed

Best Practices

- Tests should fail the first time
- Start with the simplest tests
- Don't use constructors in test cases – use setup routines (@Before for JUnit)
- Test should be independent: non-side-effecting and order independent
- Coupling and cohesion matter in tests too!
- Make tests small and fast.

After you've written your test, run it immediately. It should fail

The essence of science is falsifiability. Writing a test that works first time proves nothing

Often, these are the simplest tests to write, and they give you a good starting-point from which to launch into more complex interactions