Chapter 22: Developer Testing

- Testing is the most popular quality improvement activity
    - Unit testing
        - Execution of a complete class, routine or small program
        - Tested in isolation from the more complete system
    - Component testing
        - Execution of a class, package, small program or other program element that has typically been worked on by multiple teams or team members
        - Tested in isolation
    - Integration testing
        - The combined execution of two or more classes, packages, components or subsystems
        - Typically done as soon as there are two classes to test
        - Continues until entire system is complete
    - Regression testing
        - Testing previously executed test cases with the purpose of finding defects in software that previously passed the same tests
    - System testing
        - Execution of the software in its final configuration
        - Tests for security, performance, resource loss, timing and other things that cant be tested at lower levels of integration
- Two broad types of testing
    - Black box testing
    - White box testing

Role of Developer Testing in Software Quality

- Testing can be hard for developers to want to do because
    - Testing goal runs counter to the goals of other development activities → to find errors
    - Testing can never completely prove the absence of errors
    - Testing itself does not improve software quality
        - Rather they are an indication of a lack of it
    - Testing requires you to assume that you'll find errors in your code
- Pretty much write a test after you write a routine

Recommended Approach to Developer Testing

- Test for each relevant requirement to make sure that the requirements have been implemented
    - **Plan the test cases for this step at the requirements stage, or as early as possible**
- Test each relevant design concern to make sure the design has been implemented
    - **Plan the test cases for this at the design stage or as early as possible**
    - Before coding has begun
- Use "basis testing" to add detailed test cases to the tests that test requirements and design
    - Try to test every line of code (uhh really???)

- Use a checklist of the kinds of errors you've made on the project to date or on previous projects
- **Design the test cases along with the product**
    - **This is huge**

Test First or Test Last

- Writing test cases first will minimize the amount of time between when a defect is inserted into the code and when the defect is detected and removed
- Writing the test cases before writing the code doesn't take any more effort than writing the test cases after
- When you write the test cases first, you detect defects earlier and can correct them more easily
- Writing test cases forces you to think a little bit about the requirements and design before writing the code, which tends to produce better code
- Writing test cases first exposes requirements problems sooner, before the code is written, because its hard to write a test case for a poor requirement
- If you save your test cases, you can always test last haha

Limitations of Developer Testing

- Developer tests tend to be "clean tests"
    - Developers tend to test for whether the code works
    - As opposed to testing for all the ways the code can be broken
    - **Want 5 dirty tests for every 1 clean test**
- Developers tend to have an optimistic view of test coverage
- Developer testing tends to skip more sophisticated kinds of test coverage
-

Bag of Testing Tricks

Incomplete Testing

- Impossible to test every possible case
    - Need to focus on a handful that will give you the most information
- Eliminate those tests that don't tell you anything new

Structured Basis Testing **(Control Flow Testing)**

- Basic idea is you need to test each statement in a program at least once
    - Logical statements like if or while
- Count the number of test cases needed
    - 1) Start with 1 for the straight path through the routine
    - 2) Add 1 for each of the following keywords
        - If, while, repeat, for, and, or

Data-Flow Testing

- Data flow testing is based on the idea that data usage is at least as error prone as control flow
- Data can exist in one of three states:
    - Defined
        - Data has been initialized, but it hasn't been used yet
    - Used
        - The data has been used for computation, as an argument to a routine, or for something else
    - Killed
        - The data was once defined, but it has been undefined in some way
- In addition to these, there are some other terms
    - Entered
        - The control flow enters the routine immediately before the variable is acted upon
        - Like a working variable initialized at the top of a routine
    - Exited
        - The control flow leaves the routine immediately after the variable is acted upon
        - Like a value assigned to a status variable at the end of a routine
- Combinations of Data States
    - Defined-defined
        - Shouldn't define variables twice
    - Defined-exited
        - If variable is local, doesn't make sense to define it and exit without using it
        - If routine parameter or global, might be alright
    - Defined-killed
        - Suggests that either the variable is useless or code that was supposed to use the variable is missing
    - Entered-killed
        - This is a problem if the variable is local
        - As long as the variable is defined somewhere else before its killed its ok
    - Entered-used
        - Only problem if local
        - Variable needs to be defined before its used
        - Unless global or parameter, then needs to be defined somewhere else
    - Killed-killed
        - A variable shouldn't need to be killed twice
    - Killed-used
        - Logical error
        - Don't use variables that don't exist anymore
    - Used-defined
        - Don't want to use a variable before its been defined
- Typically test all defined-used combinations

Equivalence Partitioning

- A good test case covers a large part of the possible input data
    - If two test cases flush out exactly the same errors, you only need one of them

Error Guessing

- In addition to formal test techniques, good programmers use a variety of less formal, heuristic techniques to expose errors in their code
- Legit just guessing where things might go awry and writing test cases accordingly

Boundary Analysis

- Trying to root out < <= errors
- Test
    - Below the max (min)
    - The max (min)
    - Above the max (min)

Classes of Bad Data

- Typical cases include
    - Too little data (or no data)
    - Too much data
    - The wrong kind of data (invalid data)
    - The wrong size of data
    - Uninitialized data
- Some of these test cases are probably already covered

Classes of Good Data

- When trying to find errors in a program, easy to overlook the fact that the nominal case might contain an error
- Good cases to check
    - Nominal cases (expected values)
    - Minimum normal configurations
    - Maximum normal configurations
    - Compatibility with old data

Use Test Cases That Make Hand-Checks Convenient

Typical Errors

Which Classes Contain the Most Errors

- Most errors tend to be concentrated in a few highly defective routines
    - 80% of errors found in 20% of projects classes or routines
    - 50% of errors found in 5% of a projects classes

    - Super interesting that 20% of a projects routines contribute 80% of cost development….
- Regardless of exact percentages, highly defective routines are extremely expensive
- Pretty much want to avoid troublesome routines

Errors by Classification

- Scope of most errors is fairly limited
    - 85% of errors could be corrected without modifying more than one routine
- Many errors occur outside of construction
    - Thin application-domain knowledge is big
    - Fluctuating and conflicting requirements are no good
    - Communication and coordination break down aren't great either
- "If you see hoof prints, think horses – not zebras. The OS is probably not broken. And the database is probably just fine" – our boi Andy Hunt Dave Thomas
    - LMAOOOOOOOOOOOOOOOOOOOOOO

Errors in Testing Itself

- Want to eliminate errors in the testing suite for sure
- Check your work
    - Develop cases carefully as you develop code
    - Run through debugger and all that
- Plan test cases as you develop software
    - Effective planning should start super early avoid test cases made on bad assumptions
- Keep your test cases
    - Don't dump them and give them quality time
    - Aids in reuse, plus feels bad to chuck them
- Plug unit tests into a test framework

Test-Support Tools

Building Scaffolding to Test Individual Classes

- Scaffolding is built so workers can reach parts of a building that they couldn't reach otherwise
- OOO like this "built for the sole purpose of making it easy to EXERCISE code"
- **Mock objects**
    - Return control immediately, having taken no action
    - Test the data fed to it
    - Print a diagnostic message
    - Get return values from interactive input
    - Return a standard answer regardless of input
    - Burn up the number of clock cycles allocated to the real object or routine
    - Function as a slow, fat, simple, or less accurate version of the real object or routine
- **Driver or Test Harness**
    - Fake routine that calls a real routine that's being tested
    - Call the object with a fixed set of inputs

    - Prompt for input interactively and call the object with it
    - Take arguments from the command line and call the object
    - Read argument from a file and call the object
    - Run through predefined sets of input data in multiple calls to the object
- **Dummy files**
    - Small version of the real thing

Test-Data Generators

- Properly designed random data generators can generate unusual combinations of test data that you wouldn't think of
- Random data generators can exercise your program more thoroughly than you can
- You can refine randomly generated tests cases over time so they emphasize a realistic range of inputs
- Modular design pays off during testing
- You can reuse a test driver if the code it tests ever has the be changed

Coverage Monitors

- Keep track of code that's exercised and code that isn't

Data Recorder/Logging

- Good to log stuff so you can see what's up
- Write to external file

Symbolic Debuggers

- Ability to walk through code line by line
- Tracks variables values
- Kindof like having a peer review

System Perturbers

- Basically, find random cases where things don't work
    - Like a program working 99/100 times because 99/100 times the uninitialized variable happens to be 0
- Desired capabilities
    - Memory filling
        - Finding uninitialized variables
    - Memory shaking
        - Swaps memory locations to find code that depends on data being in absolute rather than relative locations
    - Selective memory failing
        - Can simulate low memory conditions

Error Databases

- Good idea to track errors generally and project specific
- Help give you an idea of where to start

Improving Your Testing

- Steps are similar to improving any other process
- Need to know exactly what the process does so you can vary it slightly and observe the effects of the variation

Planning to Test

- Putting testing on the same level of importance as design or coding means time will be allocated to it, and will be a high quality process
- Also key to making it repeatable
  - And thus, improvable

Re-testing (Regression Testing)

- Be sure to test old things that have already been tested
- Making sure software hasn't taken any steps backwards
- Nearly impossible to produce high-quality software unless you can systematically re-test after changes have been made
  - Running different tests after each change is no good, need to run the same tests

Automated Testing

- Only practical way to manage regression testing is through automation
- Benefits
  - An automated test has a lower chance of being wrong than a manual one
  - Once a test is automated, is readily available for the rest of the project with incremental effort
  - If tests are automated, can run frequently
  - Automated tests improve chances of detecting problems at the earlies moment possible
  - Automated tests provide a safety net for large scale code changes
  - Automated tests are especially useful in new, volatile technology environments because they flush out changes in the environment sooner rather than later