

Testing and Debugging

Errors

All programmers make errors.

- *Syntax errors* are errors in conforming to the programming language definition.
 - The compiler usually spots these early on.
- *Logical errors* are more subtle mistakes in the program.
 - Often spotted much later.
 - Compiler usually cannot help much.

GOAL: Find errors as early as possible in development.

Validation

Validation is a process designed to increase our confidence that a program works as intended.

Some validation techniques are:

- **Testing:** running a program on a set of test cases and comparing the actual results with the expected results.
- **Verification:** constructing a formal or informal argument that a program works on all possible inputs.
- **Walkthroughs:** tracing through possible execution paths, either alone or in a group setting, and documenting expected state at various points.

Debugging

Debugging is the process of pinpointing the location of errors in a program and removing them.

- Debugging involves finding and removing errors.
- Testing increases our confidence in program correctness.

Types of Testing

- **Unit** testing refers to testing of individual program modules in isolation.
- **Integration** testing refers to testing that modules in a program function properly together.
- **Regression** testing refers to systematic retesting performed after program modifications.

So far, we have mostly focused on implementing single modules (classes), so we will concentrate on unit and regression testing.

Unit Testing

In *unit testing* each program module is tested individually.

- Modules may be methods, classes, or packages.
- Should be done early to minimize development costs.
- A test suite should be built up.

Testing Strategies

A variety of testing strategies are used in industry.

- **Exhaustive** testing involves running the program on all possible inputs. (Usually not feasible.)
- **Black-box** testing checks whether the input/output behavior of the program satisfies its specification.
- **Glass-box** (or *white-box*) testing uses knowledge of the internal structure of the code in generating test cases.

Constructing Test Cases

- Understand the module *specification* (or *contract*).
- Look for violations:
 - *Positive* tests check that the module works properly in normal situations.
 - *Negative* tests check that the module behaves reasonably in abnormal situations.

Adopt an adversarial attitude!

Black-box Testing: Selection of Test Cases

- Test *typical* input values.
- Test *boundary* input values.
 - zero, one, full.
 - Search an empty collection.
 - Add to a full collection.
- Test all combinations of *maximum* and *minimum* input values.
- Test “incorrect” or “garbage” input.
- Test cases in which each declared exception is thrown.
- Test objects in all possible states (requires setup).

- Test all “paths” through the specification.
 - “if key is found in the table then return corresponding value, else return null”
 - “returns -1, 0, or 1, depending on whether x is less than, equal to, or greater than y”

Glass-box Testing: Selection of Test Cases

- Always include test cases for each branch of a conditional.
- Test at least two iterations for loops with a fixed amount of iteration.
- Test zero, one, and two iterations, plus all ways to terminate the loop, for loops with a variable amount of iteration.
- For recursive procedures, test zero and one recursive call.

Regression Testing

Regression testing involves re-running tests after program modifications, to verify that output is the same.

- Saving “correct” output from the “before” version and compare with output from the “after” version.
- Should be self-checking and automatic.
- Should be silent unless something is wrong.

Unit Testing in BlueJ

The interactive features of BlueJ help with manual unit testing:

- Can create objects of individual classes.
- Individual methods can be invoked.
- Can inspect object's state.

Automated Testing in BlueJ

Manual testing can be tedious. Testing should be automated.

- Construct a *test rig*, (also *test harness*, or *test driver*) to run tests automatically.
- Can be a separate class, or can be part of the class (e.g. `main`).
- Can setup and run tests and check results.

See [diary-testing](#) project.

Automated Testing using JUnit

See `diary-testing-junit-v1` and `v2` projects, and Exercises 6.16 through 6.19 on page 165 of the BlueJ book.

Testing the Bignum Class

Large Programs: Bottom-Up Testing

Unit testing ideas are best employed by testing a large program “bottom-up.”

- First test classes that don't depend on any other classes.
- Next test classes that depend only on classes already tested.
- Repeat until all classes have been tested.

Bottom-up testing can give the benefits of unit testing, while often avoiding the need for stubs.

Systematic Testing (pointers from K&P)

The single most important rule of testing is to *do it*.

- **Test incrementally.**

- Design and run tests as you build the code.
- Use *assertions* to check for problems as part of normal execution (e.g. checking that a statically initialized array is sorted).
- When you find bugs, create new tests that would uncover those bugs in the broken code.
- Never throw away a test.

- **Test simple parts first.**

- Test simplest and most commonly executed features first.
- Move on to more complex features only when the simple ones are working properly.

- **Know what output to expect.**
 - Don't depend on "eyeballing" to check correctness.
 - Check sophisticated algorithms against slower but simpler versions.
- **Automate testing**, because machines don't make mistakes or get tired or fool themselves into thinking something is working when it isn't.
 - Write test drivers that read test inputs and outputs as lines from a file and check the results.
 - Write programs to generate test cases, if possible.
- **Keep records.**
 - It's easy to forget what's been tested and what hasn't.