Testing

Tuesday, October 9

Announcements

Bug fix to the front end; see #announcements channel for fix

Sprint 1 due Friday at 5.

Testing

How do you know your code will work?

If it doesn't work, how do know when you fixed it?

You can manually check the program each time.

- Expensive, slow, and unrealistic

Automated testing

Testing provides a **consistent** mechanism for checking that your code meets all requirements.

Testing is **automated,** so you can continuously check that your code is "up to spec."

Testing

We'll only focus on **unit-testing.**

CS 362 goes in-depth into testing.

Verification vs Validation

Validation: are we building the right product?

Acceptance testing, user demos, beta testing

Verification: are we building the product right?

Formal methods, unit tests, regression tests

Types of testing

Development testing: done during development, by developers and testers.

Release testing: done prior to a release or other milestone. Done outside the dev team (e.g. the customer)

User testing: done by the actual users (beta testing)

Unit testing

“Unit tests test individual units (modules, functions, classes) in isolation from the rest of the program.”

- Kent Beck

JUnit

In Maven, by default, tests live under `src/test/java`

Each production class has a test class, named *Test

- e.g. `Game` -> `GameTest`

Test classes live in the same package as production classes

- You can access package protected method or fields

Test methods

All test methods are annotated with `@Test`

```
@Test
public void testSomething() {
 // …
}
```

Test fixtures

If you need some set up, annotate the method with `@Before`. This executes before the each test case.

```
@Before
public void setUp() {
}
```

This allows you to set up **test fixtures.** A fixture is the context in which a test is run.

Test fixtures

To release resources after a test, annotate the clean-up method with @After.

```
@After
public void tearDown() {
}
```

Test fixtures

`@BeforeClass` and `@AfterClass` allow you to perform text fixture setup and teardown only once per class run

`@BeforeClass` is executed once, before any of the test methods in the class are run

`@AfterClass` once after all the tests are run).

Assertions

Allow you to check for expected values.

`org.junit.Assert` class contains assert methods that you can use: `assertTrue`, `assertFalse`, `assertEquals`, etc.

`assert` is a Java keyword and will crash the JVM if it fails (similarly to C/C++). They are also disabled by default.

Test execution

JUnit executes tests in a deterministic, but **unpredictable** order.

Don't rely on the execution order for your tests to pass.

- This breaks the independence assumption of unit tests
- It might not work on a different JVM, or even on a different run

How do we write tests?

Blackbox testing: You don't know the implementation

Whitebox testing: You know the implementation

Blackbox testing

You don't know the structure of code (it's a black box, you can't see inside)

Tests are derived from the specification, documentation etc.

Useful for testing 3rd party libraries as well

Equivalence class partitioning

Partitioning the input into "equivalence classes" allows to reasonable cover a good range of the input

e.g. validation for the username field: length between 1 and 25 characters, with no spaces

ECP

username: length = [1, 25] with no spaces

Input	Output
""	Invalid
"averylongusernamethatislongerthan25characters"	Invalid
"username"	Valid
"with space"	Invalid
"averylongusernamewitha spaceandmorethan25characters"	Invalid

Boundary Testing

Similar to ECP, but focuses on edge cases.

For the username validation:

Input	Output
""	Invalid
"a"	Valid
"25characters1234567890123"	Valid
"26characters12345678901234"	Invalid

Whitebox testing

You know the structure of the code being tested.

The goal is execute all lines of the code, branches, etc.

Measuring *testedness*

How good are you tests?

Statement/line coverage

Branch coverage

Path coverage

Statement/line coverage

Measure how many (executable) statements did you cover?

Expressed as a % (number of covered statements / all statements) at different levels: method, class, package, system.

Built into IntelliJ IDEA.

Statement/line coverage

Does 100% line coverage guarantee a bug free code?

No!

You can cover all lines, but not cover all branches.

Branch coverage

Make sure that all branches in a program are covered.

All decisions points must be exercised.

Path coverage

Makes sure that all possible **executions paths** are covered

Impossible for programs with loops

Is our program bug free?

No. How does our component interact with others?

Integration testing

Test that your system's components interact as expected

Can reveal more subtle bugs. A failed integration test is a missing unit test.

- You can expose the bug by writing a unit test on the right component.

Regression testing

Tests to make sure that bugs reappear.

For every bug, first write a test to expose the bug (fails), then fix.

The test is now part of your regression suite.

User testing

The users just use the system

This allows you to check that everything works as the **users expect to.**

It also exposes incomplete features, missing/misunderstood requirements etc.

Assertions

Your tests are only as good as your assertions!

Weak assertions will let bugs through.

Bad

```
@Test
public void testAttack() {
 // ..some setup code..
 board.attack(3, 'A');
}
```

OK…

```
@Test
public void testAttack() {
 // ..some setup code..
 Result r = board.attack(3, 'A');
 assertNotNull(r);
}
```

Good

```
@Test
public void testAttack() {
 // ..some setup code..
 Result r = board.attack(3, 'A');
 assertEquals(AttackStatus.HIT,
  r.getResult());
}
```