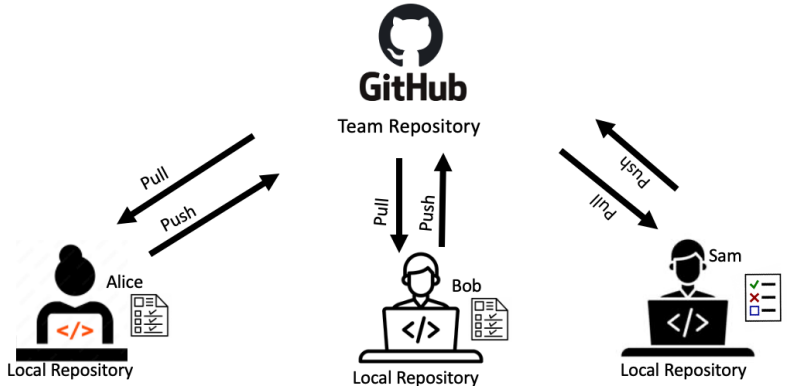
Software Testing

John Businge

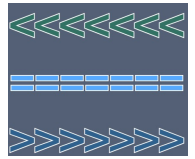
john.businge@unlv.edu

Announcements – Git and GitHub



Understand why merge conflicts happen?

- **add/add**: When both merge, parents (P1&P2) add a new file with the same name but with different contents.
- **content**: When both parents apply different changes to the same file in the same location.
- **modify/delete**: When *P1* modifies a file while *P2* deletes it.
- **rename/add**: When *P1* renames a file, and *P2* adds a new file with the same name.
- **rename/delete**: When *P1* renames a file, and *P2* deletes it.
- **rename/rename**: When both parents rename a file to different names.



How to avoid merge conflicts

- Agree on formatng and linting rules
- Make small commits and frequently review pull requests
- Pay attention and communicate

Announcements - Groups

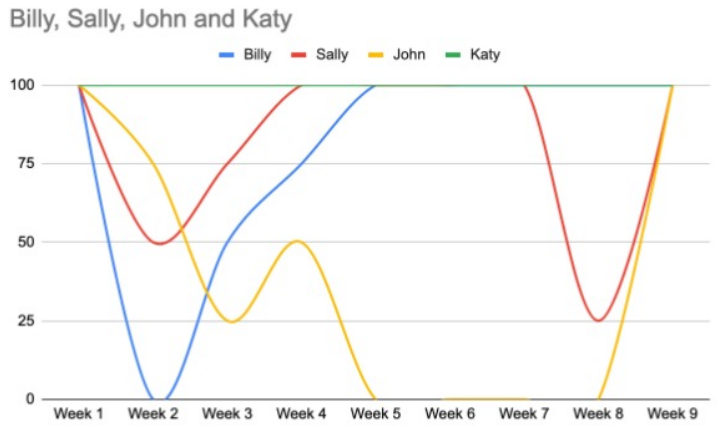
6. Team Reporting:
At each meeting you have with your team, you are required to keep minutes of what was discussed. Minutes should be rough overviews of what was discussed. Importantly, they must create specific tasks

each team member agrees to do and must document, in the next meeting, whether these tasks were completed. I expect each team to meet once per week.

As such, at the end of the semester, each team should have between 12-15 meeting minutes, depending upon when the teams were assigned. Each team may, optionally, meet more than once per week, however minutes are only required for one meeting. These documents should be short and to the point. In other words, they should spell out what team members have agreed to do (and later whether they did them), but should not go into great detail about each task. Any text written for meeting minutes may be re-used in your formal document if it is of high quality.

A. Team Tasks Chart

As part of these meetings, I want to see a graph showing exactly what was assigned **to each team member** and in the following meeting, whether that task was complete. This chart must look exactly, or nearly like this one (-5 if not):



each team member agrees to do and must document, in the next meeting, whether these tasks were completed. I expect each team to meet once per week.

As such, at the end of the semester, each team should have between 12-15 meeting minutes, depending upon when the teams were assigned. Each team may, optionally, meet more than once per week, however minutes are only required for one meeting. These documents should be short and to the point. In other words, they should spell out what team members have agreed to do (and later whether they did them), but should not go into great detail about each task. Any text written for meeting minutes may be re-used in your formal document if it is of high quality.

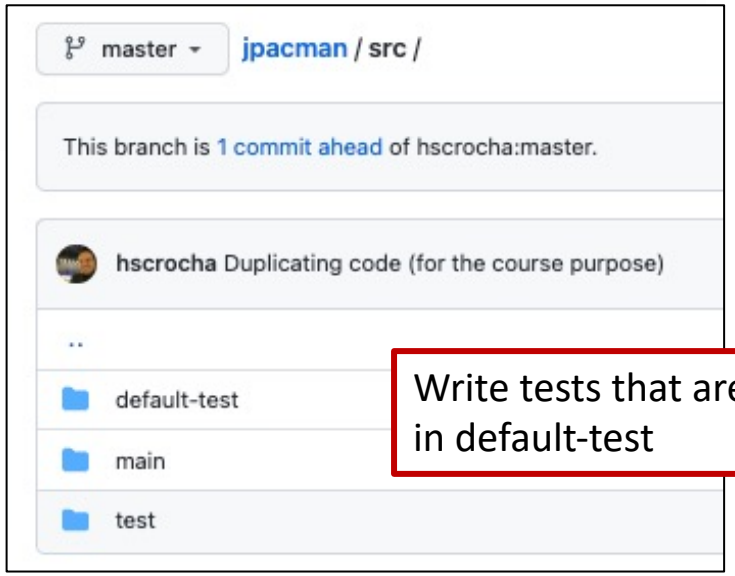
Announcements – Testing report

Task 2.1 - 30 points (10 points each)

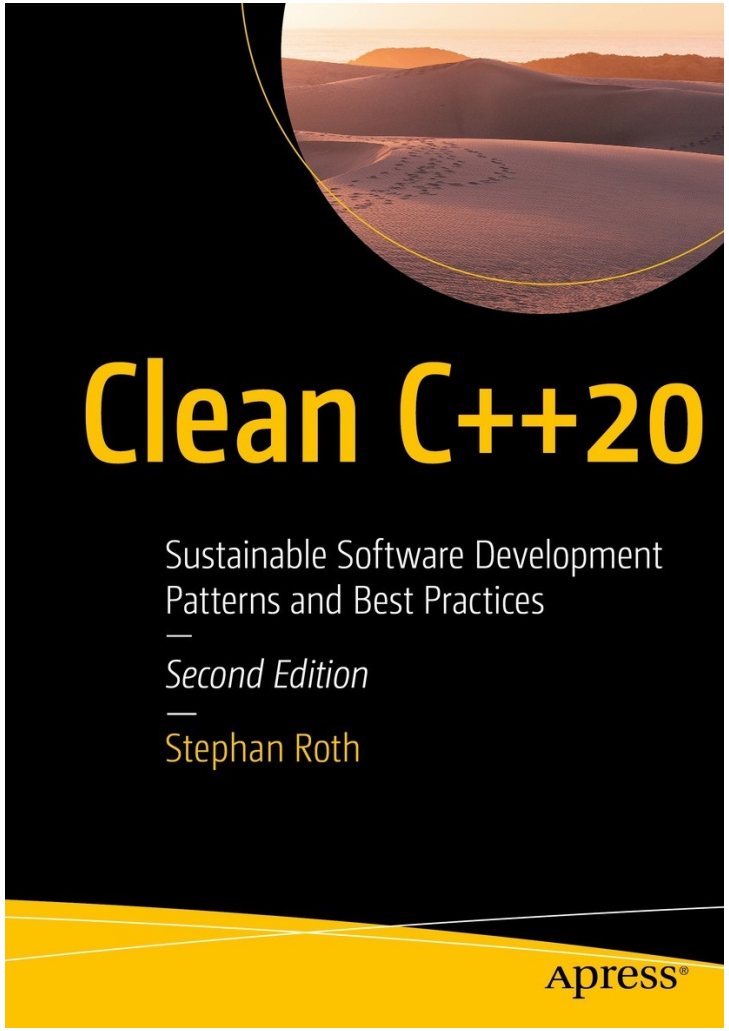
Identify **three or more methods** in any java classes and write `unit tests` of those methods. **Remember to take screenshots of the test coverage before and after creating the unit tests. Since there are many methods in the project, I should not find almost all the group members of a given group attempting the same methods.** Discuss between the group mates what methods you will be writing unit tests for. A simple Google sheet having two columns would help get the group organised.

Names	Fully Qualified Method Name
John Businge	src/main/java/nl/tudelft/jpacman/game/Game.java
John Businge	src/main/java/nl/tudelft/jpacman/board/BoardFactory.java

Report
- Unit test code snippets
- Coverage screenshots



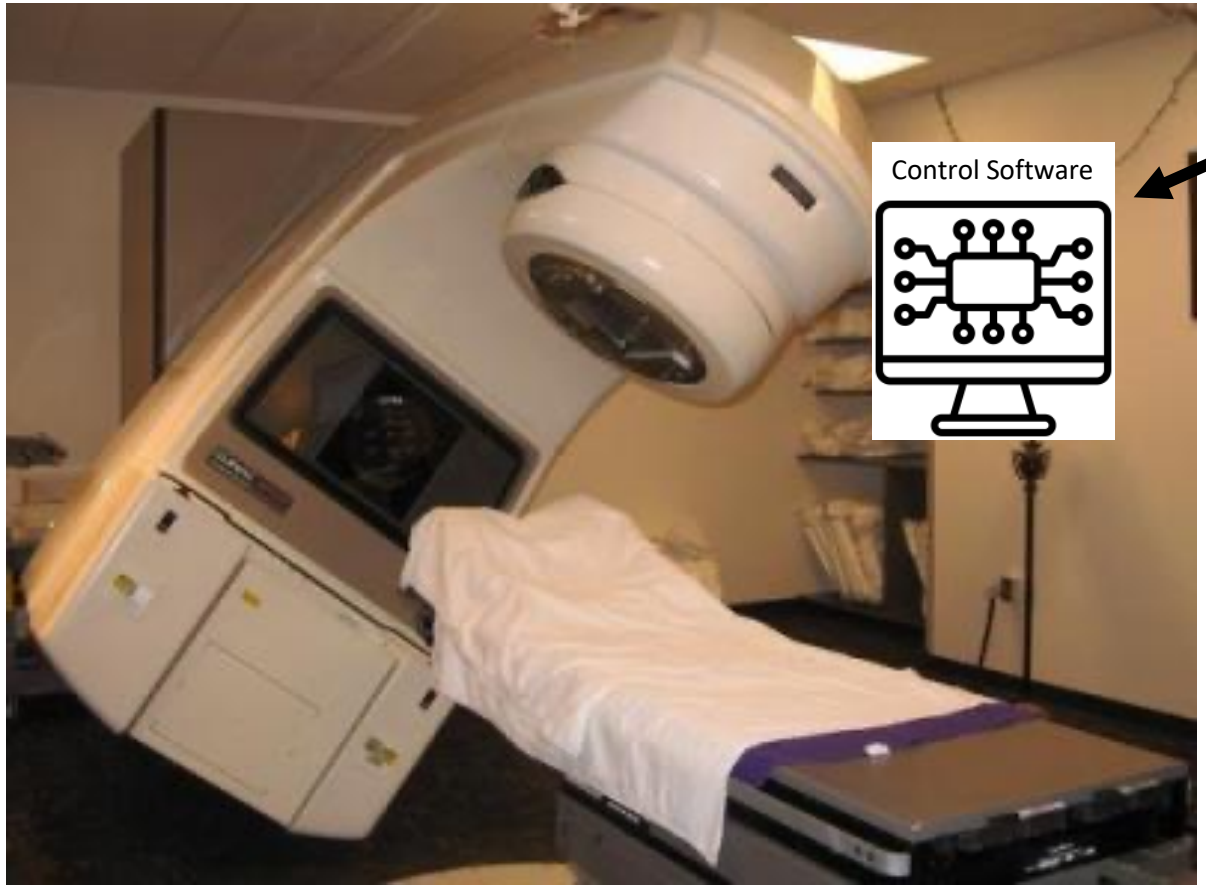
Write tests that are not in default-test

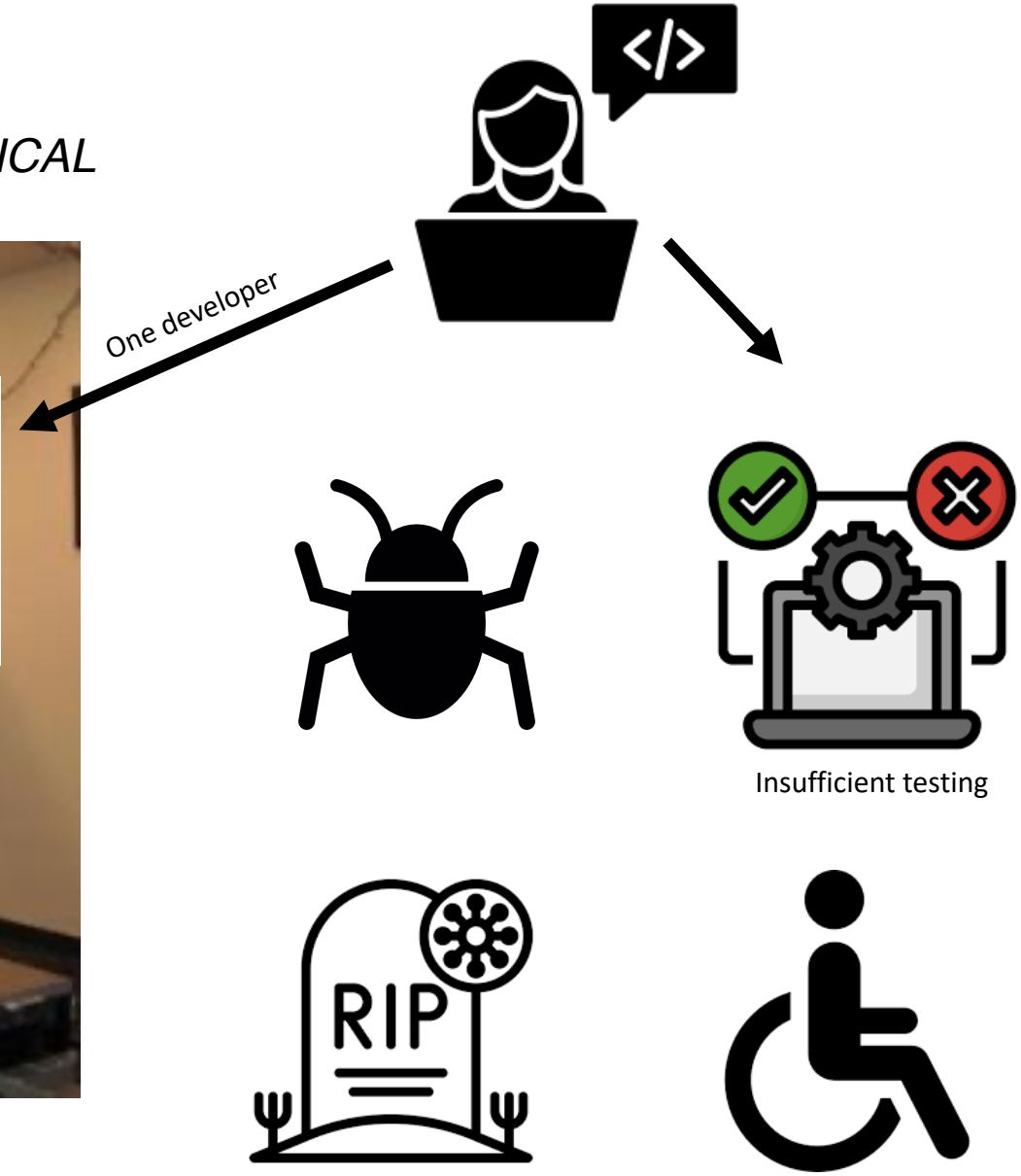


Free PDF version on Springer
https://link.springer.com/book/10.1007/978-1-4842-5949-8

The Need for Testing

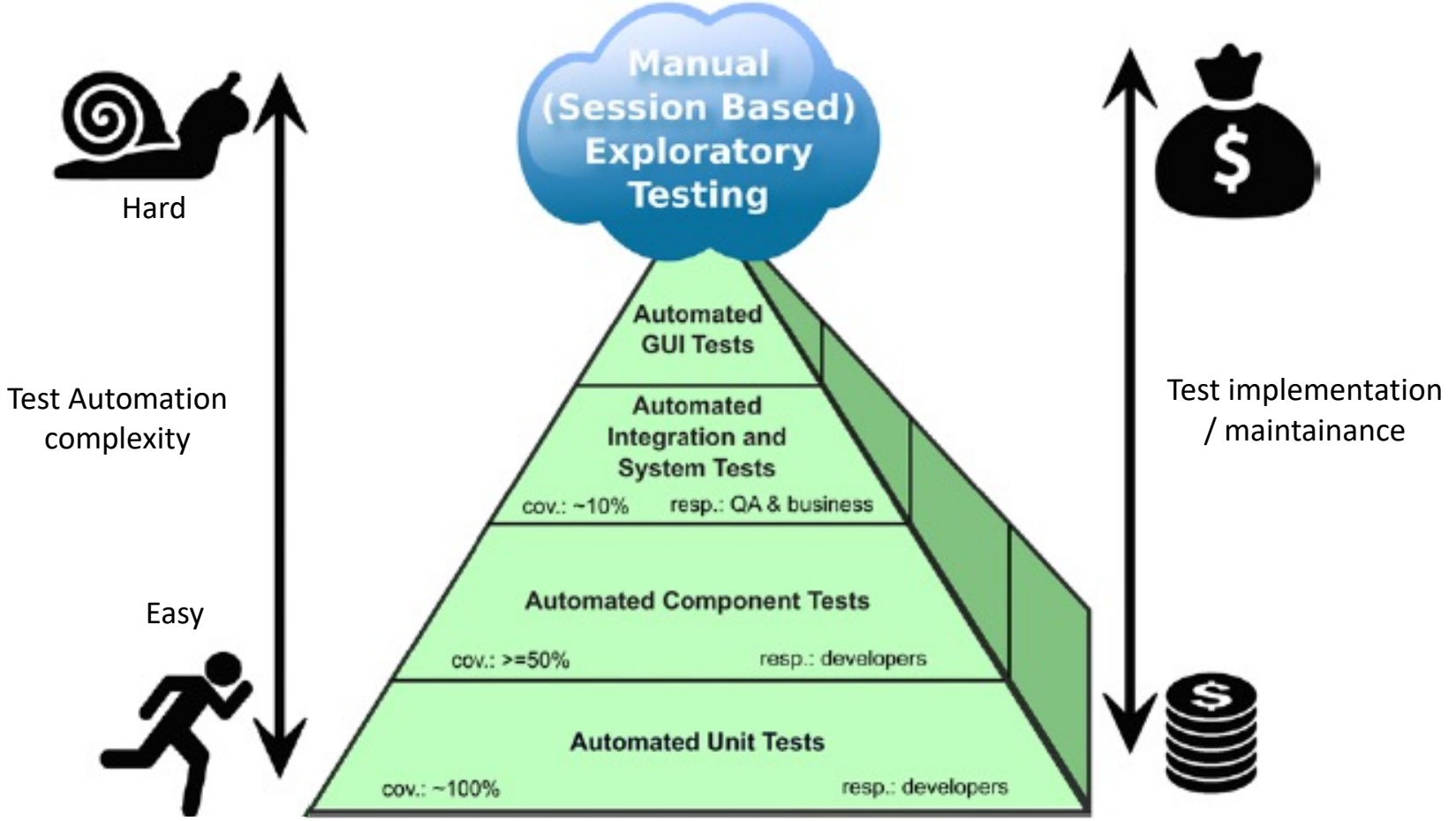
KILLED BY A MACHINE: THE THERAC-251986: MEDICAL ACCELERATOR DISASTER





https://hackaday.com/2015/10/26/killed-by-a-machine-the-therac-25/

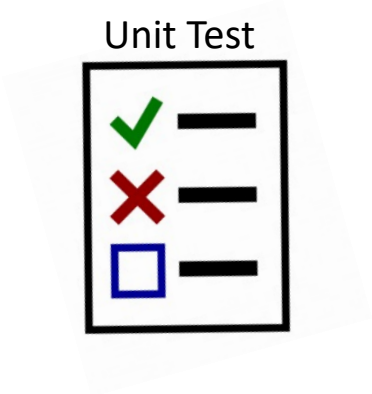
The Test Pyramid



More unit tests (100%)

Unit Tests

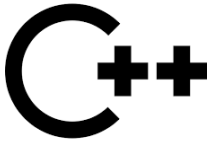
A **unit test** is a piece of code that executes a small part of your production code base in a particular context. The test will show you, in a split second, that your code works as you expect it to work. If unit test coverage is pretty high, and you can check in less than a minute that all parts of your system under development are working correctly



Why are unit tests useful?

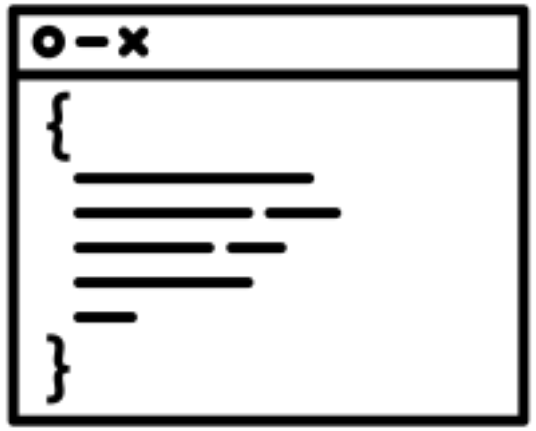
- Unit tests give you immediate feedback about your entire code base. Provided that test coverage is sufficiently high.
- A high coverage with unit tests can prevent time-consuming and frustrating debugging sessions.
- Unit tests are a kind of executable documentation because they show exactly how the code is designed to be used.
- Unit testing makes development go faster.

Unit Tests Frameworks



Rules for Good Unit Tests

```python
def validate_account_number_format(account_string):
    # Return False if invalid, True if valid
    # ...
```



Production Code

```python
# Import the code to be tested
import validator

# Import the test framework (this is a hypothetical module)
import test_framework

# This is a generalized example, not specific to a test framework
class Test_TestAccountValidator(test_framework.TestBaseClass):
    def test_validator_valid_string():
        # The exact assertion call depends on the framework as well
        assert(validate_account_number_format("1234567890"), True)

    # ...

    def test_validator_blank_string():
        # The exact assertion call depends on the framework as well
        assert(validate_account_number_format(""), False)

    # ...

    def test_validator_sql_injection():
        # The exact assertion call depends on the framework as well
        assert(validate_account_number_format("drop database master"), False)

    # ... tests for all other cases
```

Test Code

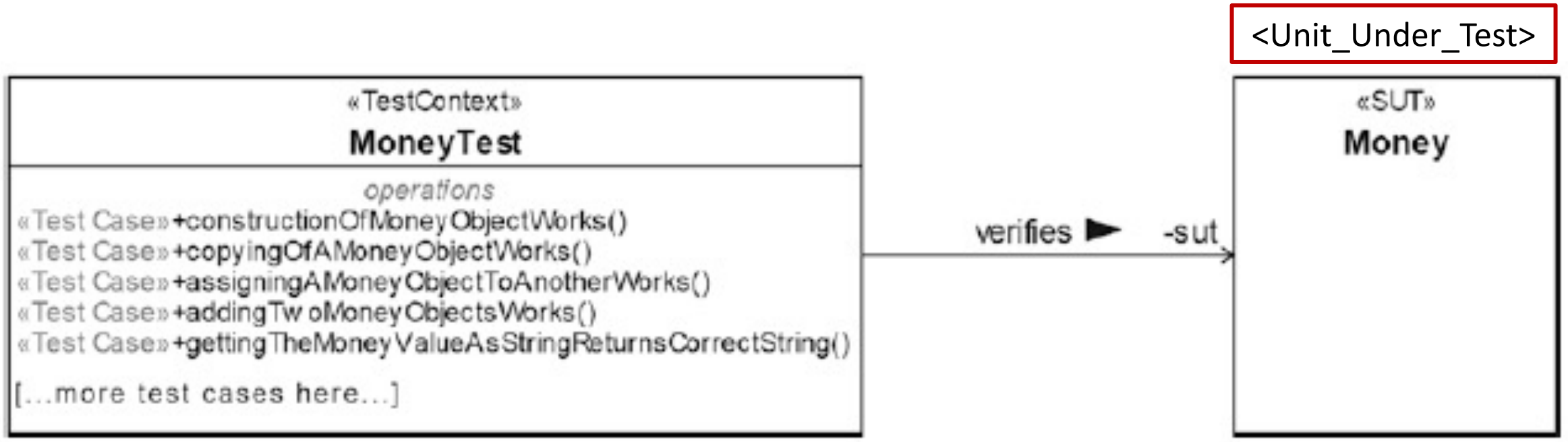
Rules for Good Unit Tests - Unit Test Naming

If a unit test fails, the developer wants to know immediately:

- What is the name of the unit; whose test failed?
- What was tested, and what was the environment of the test
- What was the expected test result, and what was the actual test result of the failed test?

Rules for Good Unit Tests - Unit Test Naming

Test Name: <Unit_Under_Test>Test



The system under test, (SUT) Money and its test fixture, MoneyTest

Rules for Good Unit Tests - Unit Test Naming

Undescriptive unit tests

```
testConstructor()
test4391()
sumTest()
```

Expressive and descriptive unit tests

<PreconditionAndStateOfUnitUnderTest>_<TestedPartOfAPI>_<ExpectedBehavior>

```
cacheIsEmpty_addElement_sizeIsOne()
cacheContainsOneElement_removeElement_sizeIsZero();
givenTwoComplexNumbers_add_Works()
givenTwoMoneyObjectsWithDifferentBalance_InequalityComparison_Works()
invoiceIsReadyForAccounting_getInvoiceDate_returnsToday()
```

Examples of Unit Test Names that Verify Domain-Specific Requirements

```
void ChessEngineTest::aPawnCanNotMoveBackwards();
void ChessEngineTest::aCastlingIsNotAllowedIfInvolvedKingHasBeenMovedBefore();
void ChessEngineTest::aCastlingIsNotAllowedIfInvolvedRookHasBeenMovedBefore();
void HeaterControlTest::ifWaterTemperatureIsGreaterThan92DegTurnHeaterOff();
```

Rules for Good Unit Tests – One Assession per Test

```
void MoneyTest::givenTwoMoneyObjectsWithDifferentBalance_
InequalityComparison_Works() {
const Money m1(-4000.0);
const Money m2(2000.0);
ASSERT_TRUE(m1 != m2);
}
```

```
void MoneyTest::givenTwoMoneyObjectsWithSameBalance_
EqualityComparison_Works() {
const Money m1(-4000.0);
const Money m2(2000.0);
ASSERT_TRUE(m1 == m2);
}
```

```
void MoneyTest::givenTwoMoneyObjectsWithDifferentBalance_
testAllComparisonOperators() {
const Money m1(-4000.0);
const Money m2(2000.0);
ASSERT_TRUE(m1 != m2);
ASSERT_FALSE(m1 == m2);
ASSERT_TRUE(m1 < m2);
ASSERT_FALSE(m1 > m2);
// ...more assertions here...
}
```

- Difficult to find cause of error
- Early failing assertions obsures additional errors
- "*...testAllComparisonOperators()*" not a good name.

Rules for Good Unit Tests – Exclude Getters and Setters

Don’t write unit tests for usual/simple getters and setters of a class

```
public class Student {
  private float gpa;
  private String degreeProgram;

  public float getGPA() {
    return gpa;
  }

  public void setGPA(float newGPA) {
    gpa = newGPA;
  }

  public String getDegreeProgram() {
    return degreeProgram;
  }

  public void setDegreeProgram( String newDegreeProgram ){
    if (gpa > 2.7) {
      degreeProgram = newDegreeProgram;
    }
  }
}
```

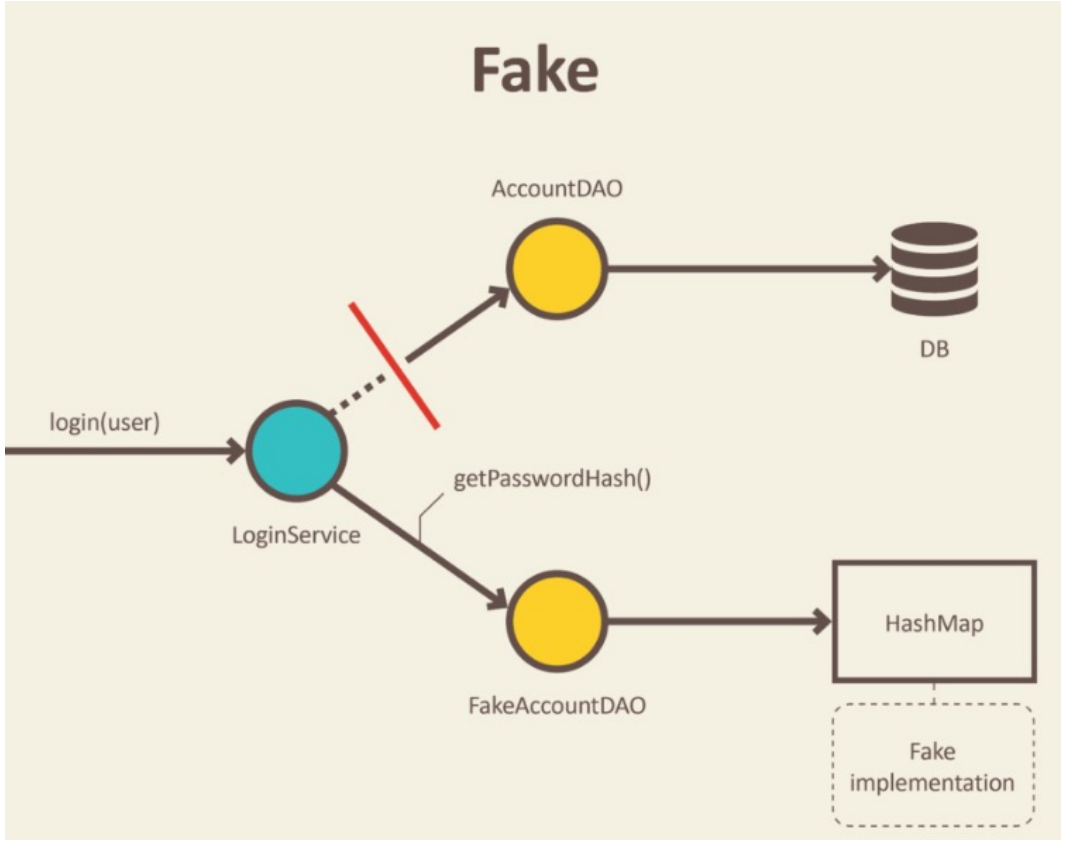
Sometimes, getters and setters are not that simple, especially those that implement information hiding

Rules for Good Unit Tests – Exclude Third-Party Code

We don’t have to verify that libraries or frameworks do work as expected

Rules for Good Unit Tests – Exclude External Systems

Mock things out and test your code, not theirs



https://blog.pragmatists.com/test-doubles-fakes-mocks-and-stubs-1a7491dfa3da

Read – Test Doubles (Fake Objects)

These values can also be depicted on a number line, as shown in Figure 2-6.

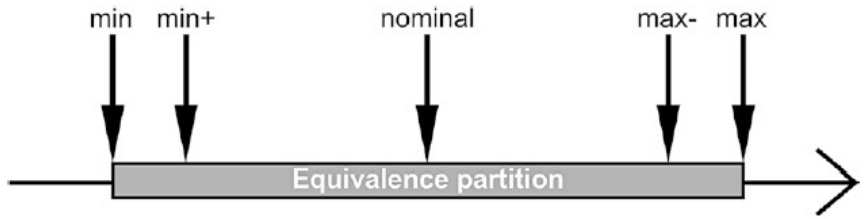


Figure 2-6. *The input parameters derived from a boundary value analysis*

If the boundary values are determined and tested for each equivalence partition, then very good test coverage can be achieved in practice with relatively little effort.

Test Doubles (Fake Objects)

Unit tests should only be called "unit tests" if the units to be tested are completely independent from collaborators during test execution, that is, the unit under test does not use other units or external systems. For instance, while the involvement of a database during an integration test is uncritical and required, because that's the purpose of an integration test, access (e.g., a query) to this database during a real unit test is proscribed (see the section "What Do We Do with the Database?" earlier in this chapter). Thus, dependencies of the unit to be tested with other modules or external systems should be replaced with so-called *test doubles*, also known as *fake objects*, or *mock-ups*.

In order to work in an elegant way with such test doubles, we should strive for loose coupling of the unit under test (see the section entitled "Loose Coupling" in Chapter 3). For instance, an abstraction (e.g., an interface in the form of a pure abstract class) can be introduced at the access point to an unwanted collaborator, as shown in Figure 2-7.

Rules for Good Unit Tests – Don’t Mix Test Code with Production Code

Avoid Dependencies to Test Code

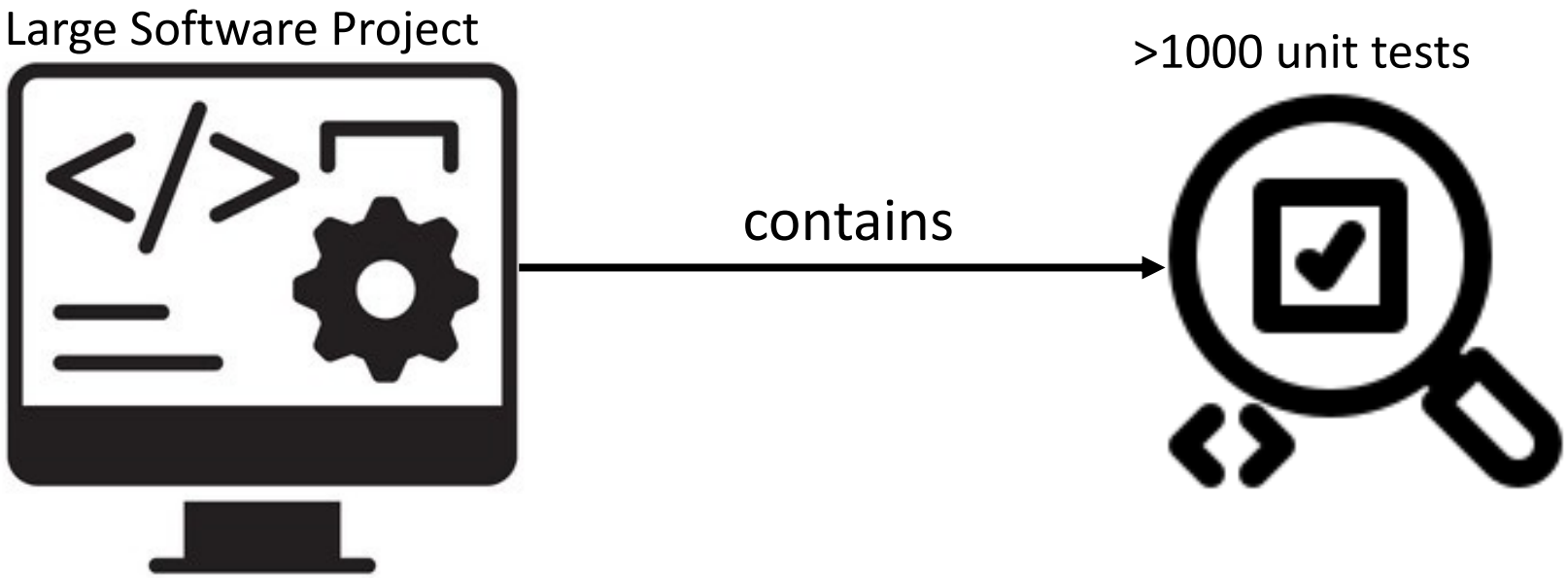
```
class Customer {
public:
  Customer() = default;
  explicit Customer(const bool testMode) : inTestMode(testMode) {}

  void save() {
    DataAccessObjectPtr dataAccessObject = getDataAccessObject();
    // ...use dataAccessObject to save this customer...
  }

  // ...

private:
  DataAccessObjectPtr getDataAccessObject() const {
    if (inTestMode) {
      return std::make_unique<FakeDAOForTest>();
    } else {
      return std::make_unique<CustomerDAO>();
    }
  }
```

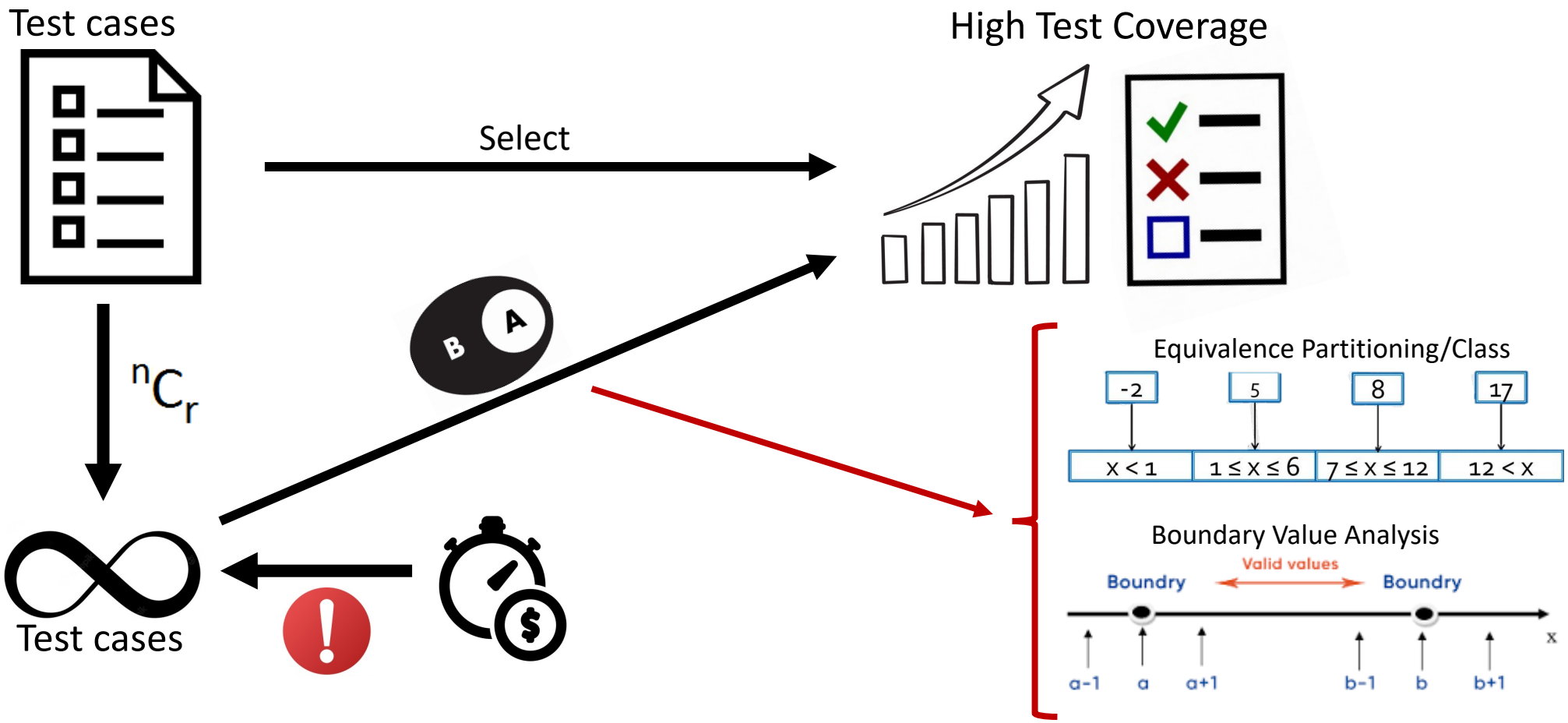
Rules for Good Unit Tests – Tests Must Run Fast



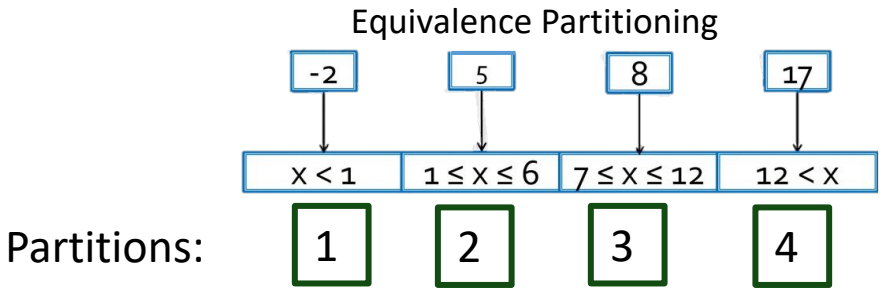
- If running all unit tests takes 15 minutes, ½ hour, or more, developers are impeded in doing their work
- If 1-test tales 0.5 secs -> 1,000-tests will take 8 minutes
- Executing the whole test suit 10 times will cost 1.5 hours of waiting time.
- This might result developers running tests less times.
- A good test suit should run under 3-minutes

Rules for Good Unit Tests – Tests Input Data

How do I find all test cases that are necessary to ensure good fault detection?



Rules for Good Unit Tests – Tests Input Data



Test input data/test cases:
{-2, 5, 8, 17}

Bank Interest Calculator API

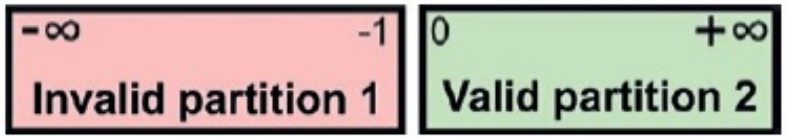
- The bank charges 4 percent penalty interest on overdrafts.
- The bank offers 0.5 percent interest for the first 5,000 USD savings.
- The bank offers 1 percent interest for the next 5,000 USD savings.
- The bank offers 2 percent interest for the rest.
- Interest is calculated on a daily basis.

Two Classes

Amount of money and, as interest is calculated on a daily basis

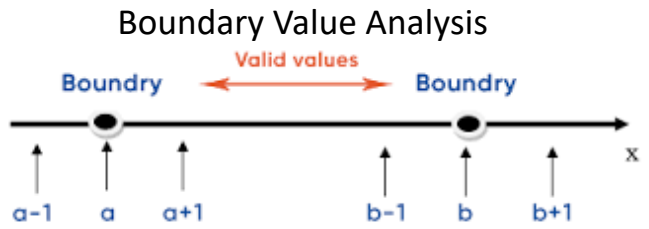
The equivalence partitioning for the amount of money

number of days for which this amount is valid

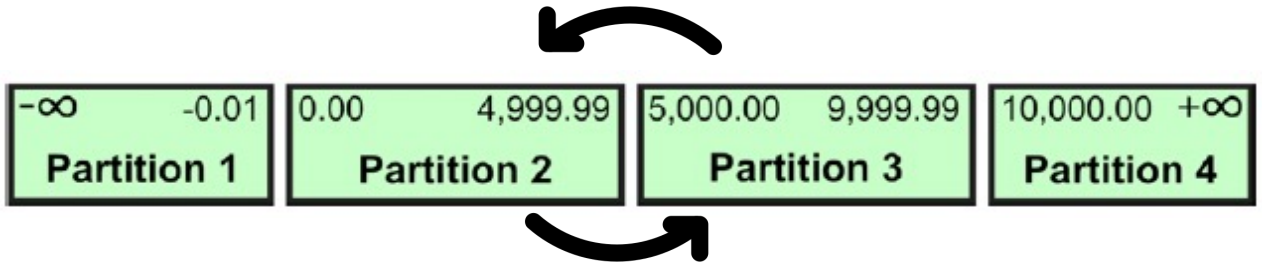


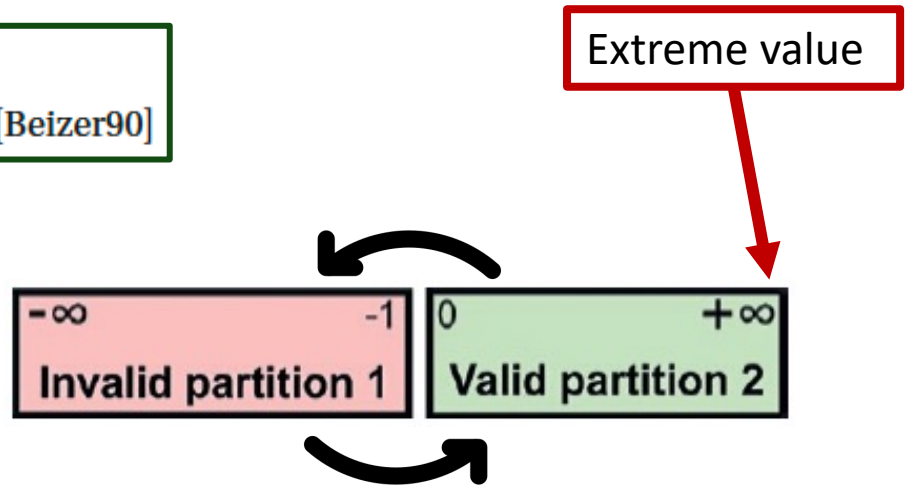
The equivalence partitioning for number of days

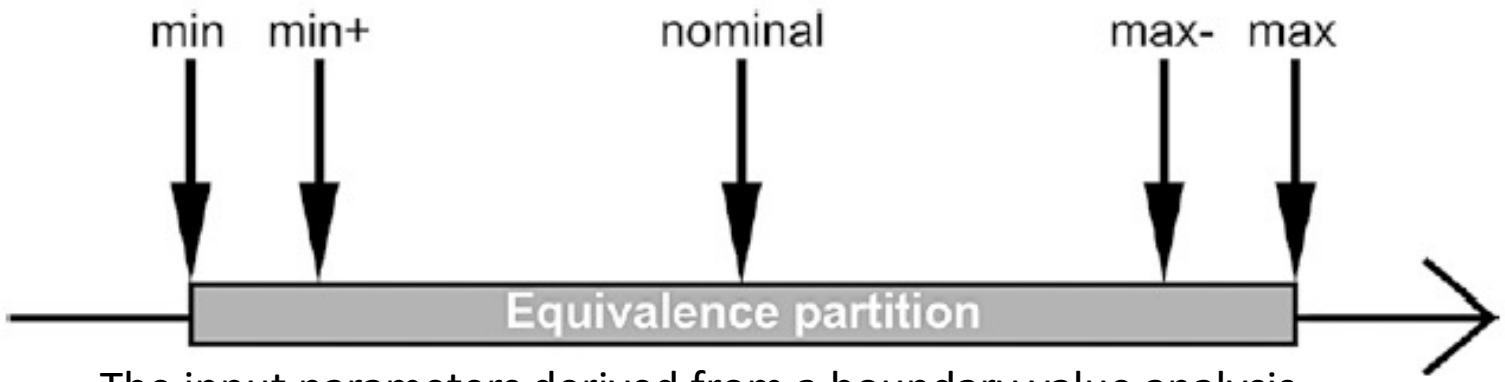
Rules for Good Unit Tests – Tests Input Data



> *"Bugs lurk in corners and congregate at boundaries."*
>
> —Boris Beizer, Software Testing Techniques [Beizer90]







The input parameters derived from a boundary value analysis

High Test Coverage

Quality of a Test Suite

- How do you know if your unit test cases are good enough?
- Are they really testing the application?
- When do we stop testing?

Solution: Test Coverage!

Test Coverage

$$\text{Coverage} = \frac{Number\ of\ Covered\ Items}{Total\ number\ of\ items} \text{ X } 100\%$$

- Examples:
  - Statement (Line, or Code) Coverage.
  - Branch (Condition) Coverage
  - Path Caverage
  - Mutation Caverage

Example: a function to test

```
int foo(int input, bool b1, bool b2, bool b3){
    int x = input;
    int y = 0;
    if(b1)
        x++;
    if(b2)
        x--;
    if(b3)
        y=x;
    return y;
}
```

Statement/Line/Code Coverage

Test Case(s)

```
ASSERT foo(0, true, true, true) == 0;
```

```
int foo(int input, bool b1, bool b2, bool b3){
    int x = input;
    int y = 0;
    if(b1)
        x++;
    if(b2)
        x--;
    if(b3)
        y=x;
    return y;
}
```

Statement/Line/Code Coverage

Test Case(s)

```
ASSERT foo(0, true, true, true) == 0;
```

```
int foo(int input, bool b1, bool b2, bool b3){
    int x = input;
    int y = 0;
    if(b1)
        x++;
    if(b2)
        x--;
    if(b3)
        y=x;
    return y;
}
```

100% Statement Coverage

Statement/Line/Code Coverage

Test Case(s)

```
ASSERT foo(0, false , true, true) == -1;
```

```
int foo(int input, bool b1, bool b2, bool b3){
    int x = input;
    int y = 0;
    if(b1)
      x++;
    if(b2)
        x--;
    if(b3)
        y=x;
    return y;
}
```

$\frac{8}{9}$ x100% = 88.9%

~~100%~~ Statement Coverage

Branch/Condition Coverage

Test Case(s)

```
ASSERT foo(0, true, true, true) == 0;
```

```
int foo(int input, bool b1, bool b2, bool b3){
    int x = input;
    int y = 0;
    if(b1)
        x++;
    if(b2)
        x--;
    if(b3)
        y=x;
    return y;
}
```

50% Branch Coverage

Branch/Condition Coverage

Test Case(s)

```
ASSERT foo(0, true, true, true) == 0;
Assert foo(0,false, false, false) == 0;
```

New Test

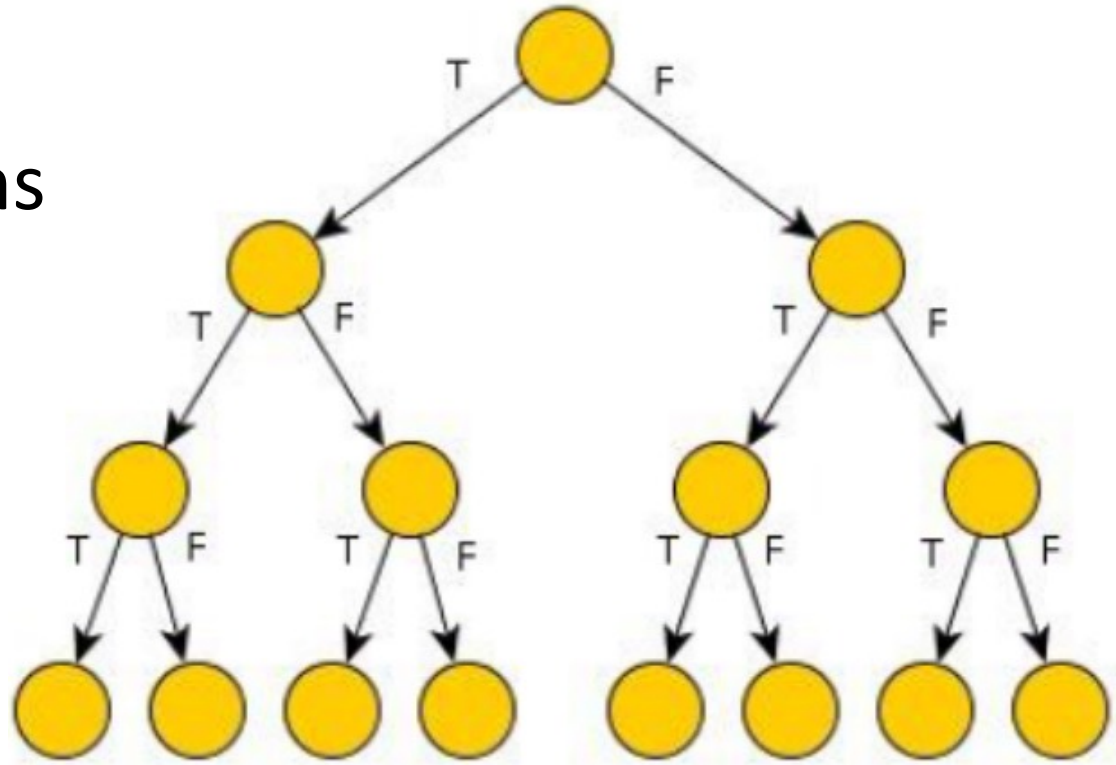
```
int foo(int input, bool b1, bool b2, bool b3){
    int x = input;
    int y = 0;
    if(b1)
        x++;
    if(b2)
        x--;
    if(b3)
        y=x;
    return y;
}
```

100% Branch Coverage

Path Coverage

Paths for three “if” each can be either true (T) or false (F)

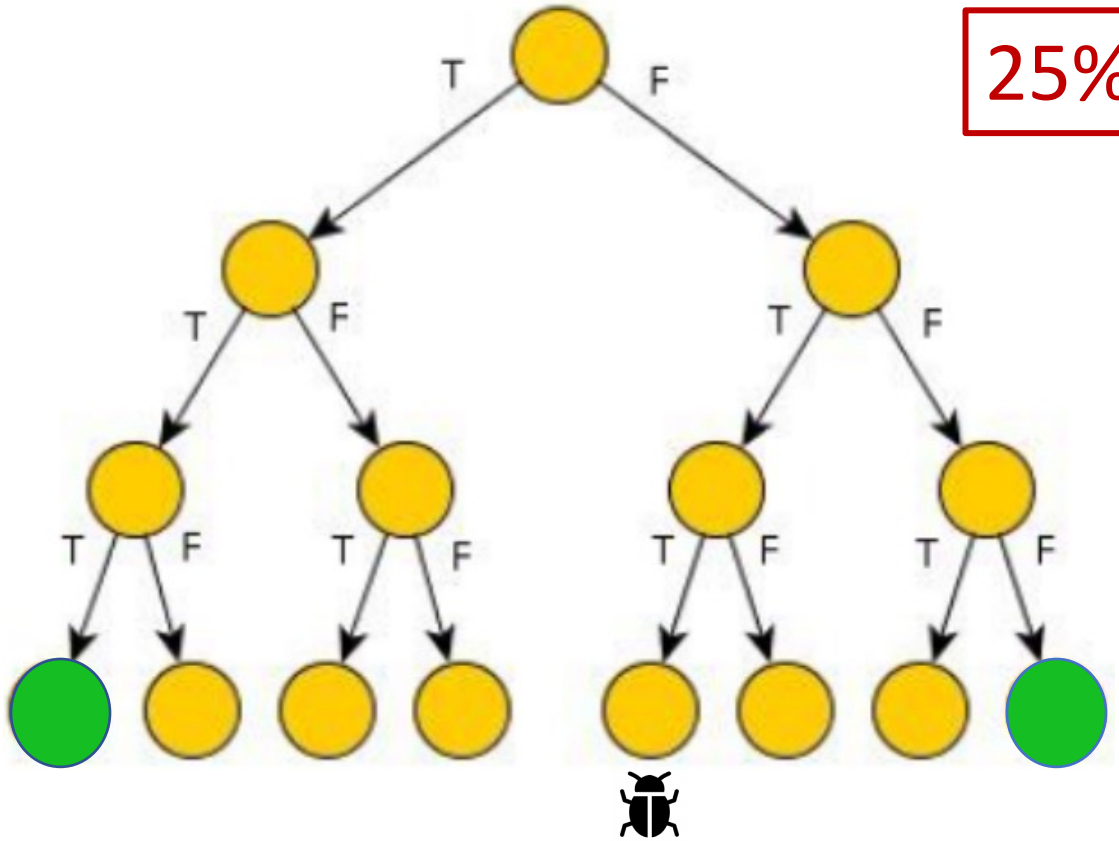
8-Paths



Path Coverage

Test Case(s)

```
ASSERT foo(0, true, true, true) == 0;
ASSERT foo(0,false, false, false) == 0;
```



25% Path Coverage

Testing Coverage for the Project

- It is required to show coverage for your Project (in both the Intermediate and the Final Report)
  - At least Statement Coverage, but Branch Coverage is better.
- If your project has very low coverage, you better have a good explanation for that.
- Focus on increasing the coverage for the system parts that will be affected by your change.

Tests

O Not present (= no tests)	O Incorrectly used (= manual test)	O Limited (= nt. everything tested) - domain tests	O Sufficient (= input) - input tests - multiple errors	O good (= output) - output tests - scenarios - code coverage >70%	O Excellent (= test code good) - subclasses - scenarios - Code coverage > 85%

Comments: