Introduction to Software Engineering
Agile Testing Practices

CMPS115 – Spring 2017
Richard Jullig

Computer Science – University of California at Santa Cruz

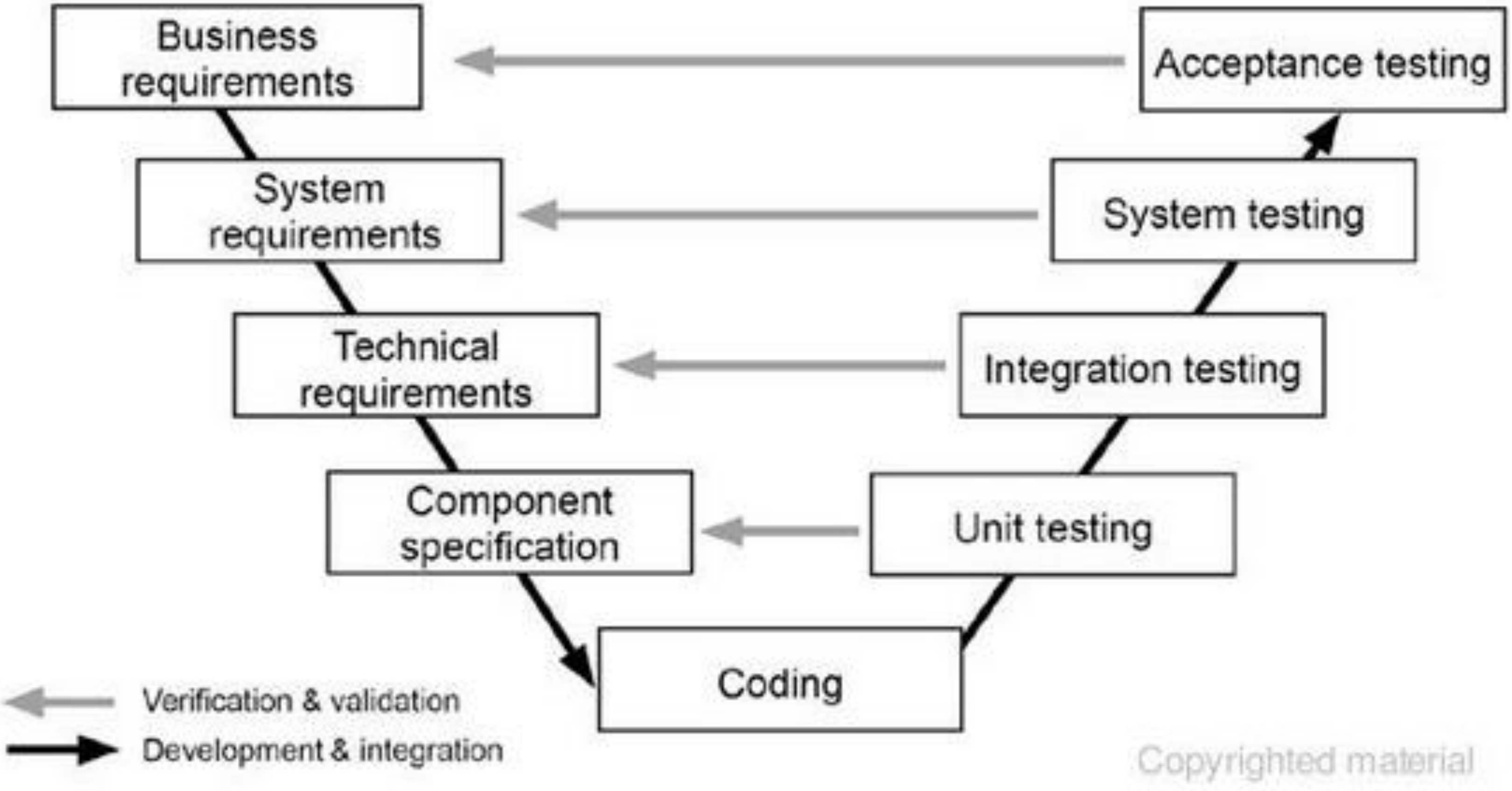
Helpful Material

- Tilo Linz, *Testing in Scrum* (2014)
  - Introduction to Agile testing with examples
  - Discusses difference to traditional testing regimes
  - See Piazza > Resources > Reading Material
- Gerard Meszaros, xUnit Test Patterns (2007)
  - The “bible” of Agile Unit testing
  - Worth investing in
  - Link to my copy on Piazza > Resources > Reading Material
- Bruegge, *Object Oriented Software Engineering Using UML* (2012)
  - Chapter 11: Testing
  - Overview; definition of test concepts

Team Exercises

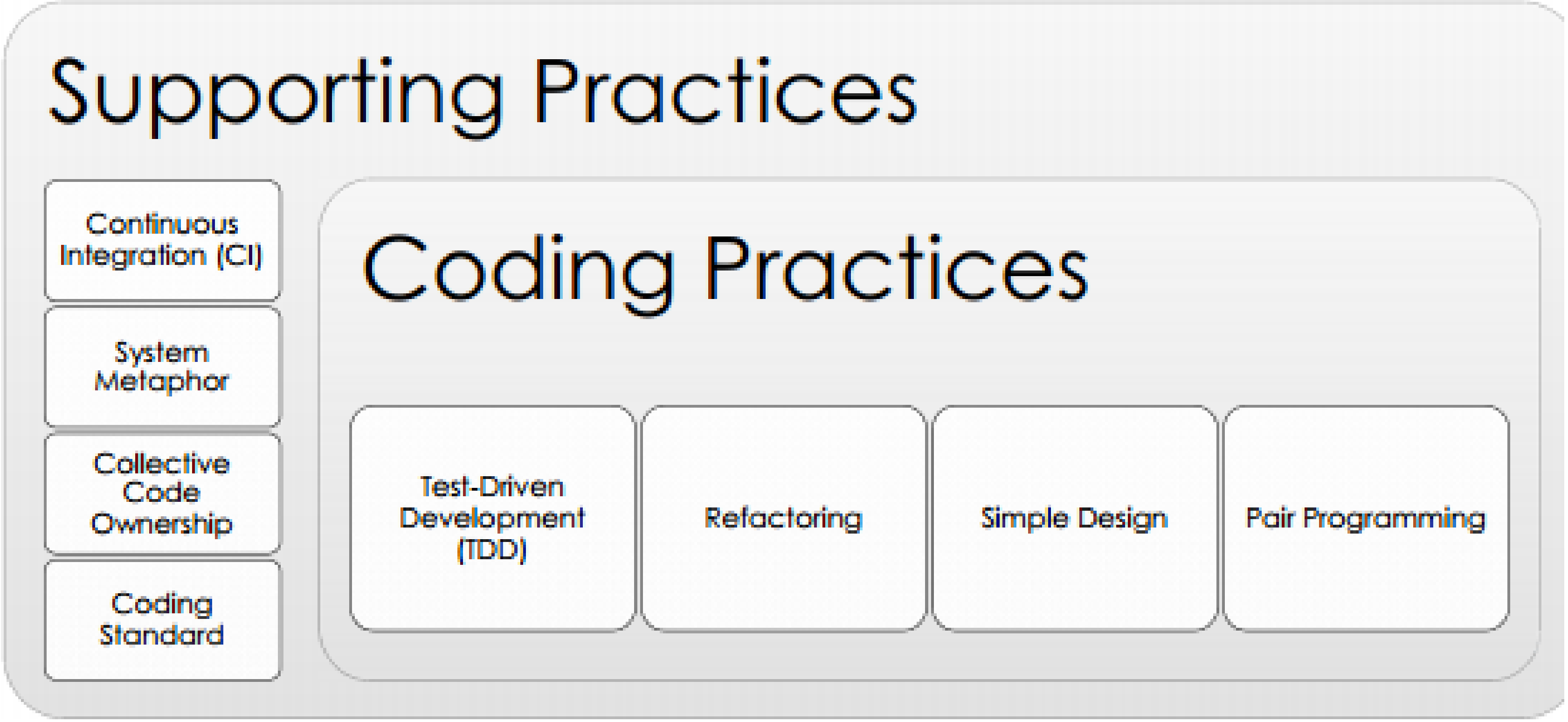
- Team working agreement(s)
- Definition of Done
- Acceptance Criteria
- Coding Standard
  - Google Python Style Guide
  - Google Java Style Guide

V-Model of Software Development



From Tilo Linz, Testing in Scrum

Technical Practices



Supporting Practices

Continuous Integration (CI)

System Metaphor

Collective Code Ownership

Coding Standard

Coding Practices

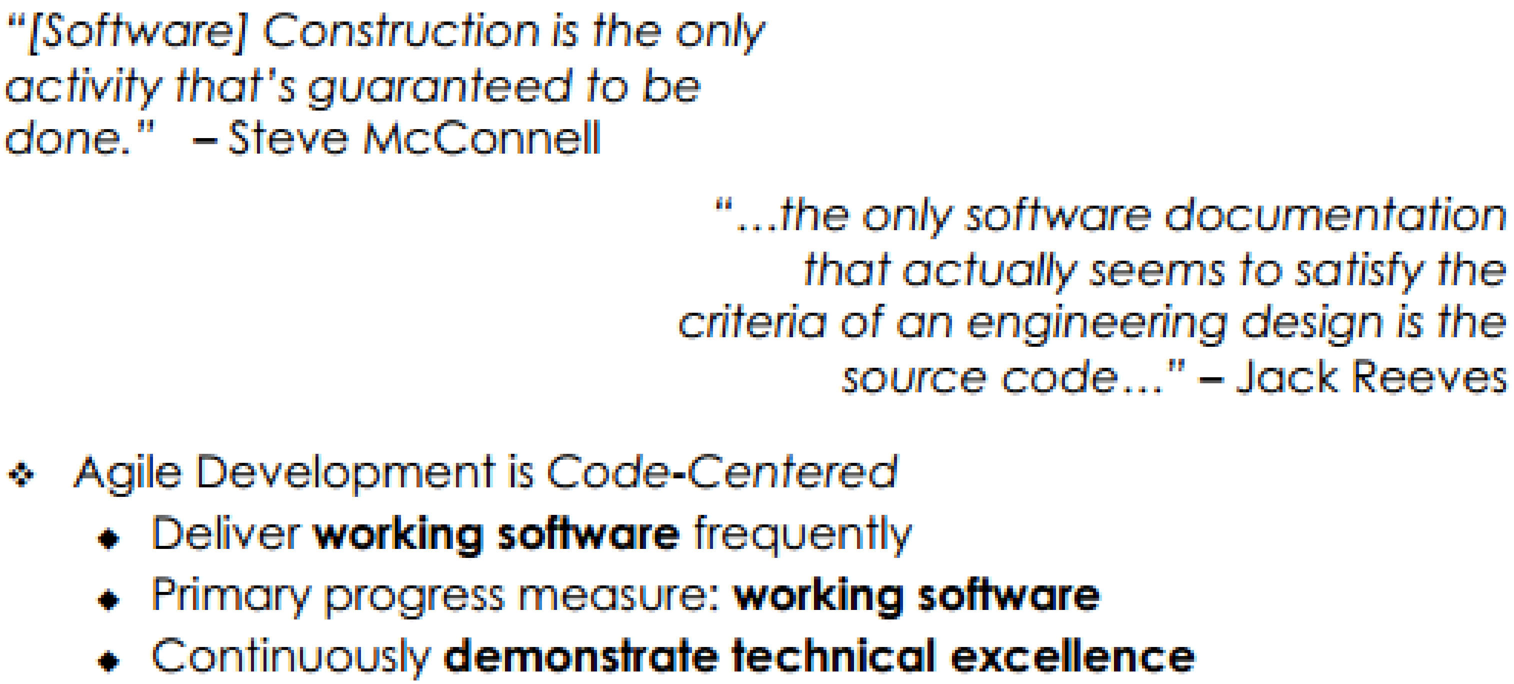
Test-Driven Development (TDD)

Refactoring

Simple Design

Pair Programming

There will be code …



"[Software] Construction is the only activity that's guaranteed to be done." – Steve McConnell

"…the only software documentation that actually seems to satisfy the criteria of an engineering design is the source code…" – Jack Reeves

- Agile Development is *Code-Centered*
  - Deliver **working software** frequently
  - Primary progress measure: **working software**
  - Continuously **demonstrate technical excellence**

Minimal Agile Toolkit

- Project Wiki (or equivalent, e.g. Google Docs)
- Version Control System/Software repository (GitHub, …)
- xUnit Framework
- Build automation/Continuous Integration
- Static Code Analysis
  - Coding standard compliance
  - Software quality measures
  - Test coverage

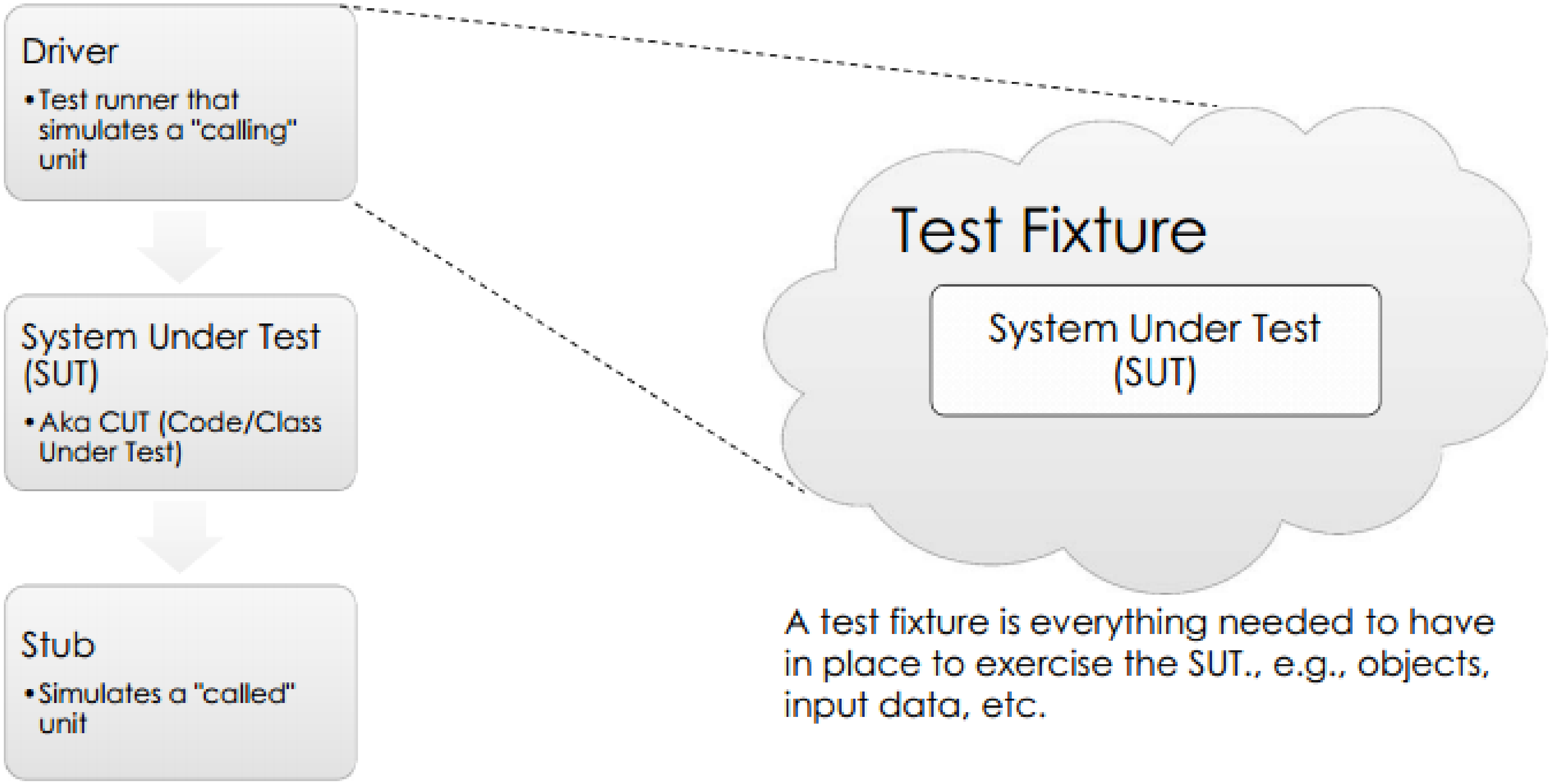
Development Practices

- Acceptance tests
  - Automate execution
- Unit tests
  - Automate execution
- Write *Clean Code*
- Keep the design simple
- Grow code and tests in small increments

Unit Tests

- Unit
  - Elementary component
  - E.g. functions, methods, classes
- Unit test
  - Collection of code
  - When executed, stimulates the unit
    - E.g. calls method with certain input
  - Verifies response
    - E.g. output, new state, exception
- Unit testing
  - Isolate the unit to be tested (eliminate dependencies)
  - Execute and check small portions of functionality at a time

Unit Test Framework



Driver

- Test runner that simulates a "calling" unit

System Under Test (SUT)

- Aka CUT (Code/Class Under Test)

Stub

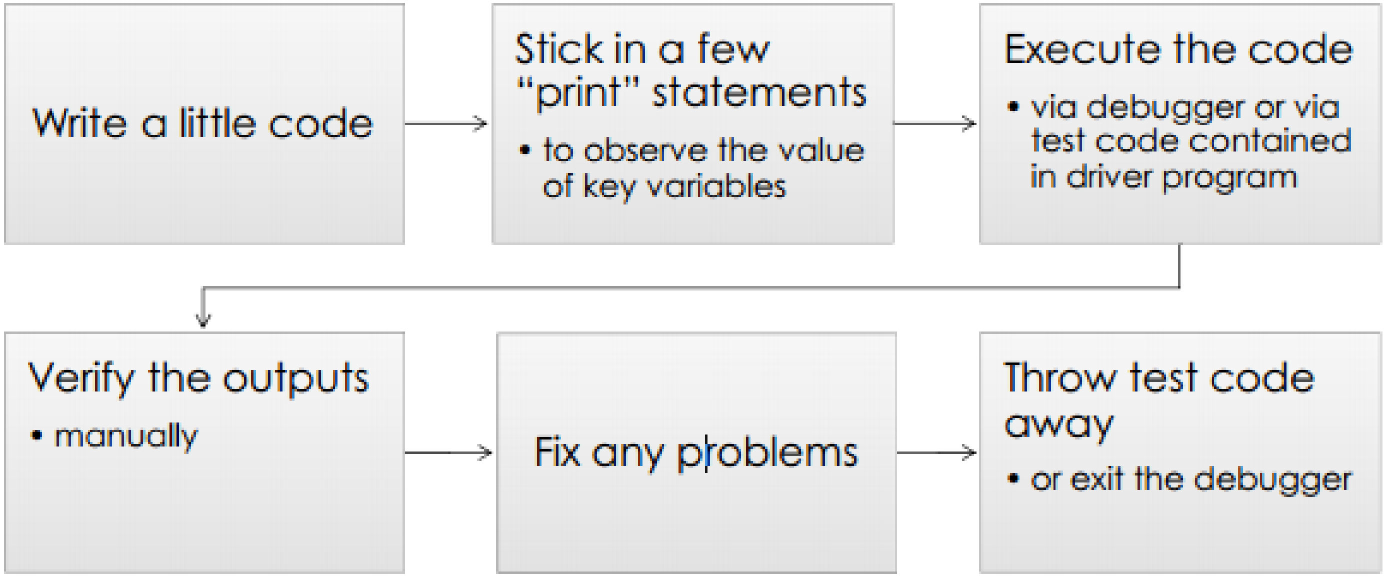
- Simulates a "called" unit

Test Fixture

System Under Test (SUT)

A test fixture is everything needed to have in place to exercise the SUT., e.g., objects, input data, etc.

Legacy (Old School) Coding Feedback



Write a little code

Stick in a few "print" statements
- to observe the value of key variables

Execute the code
- via debugger or via test code contained in driver program

Verify the outputs
- manually

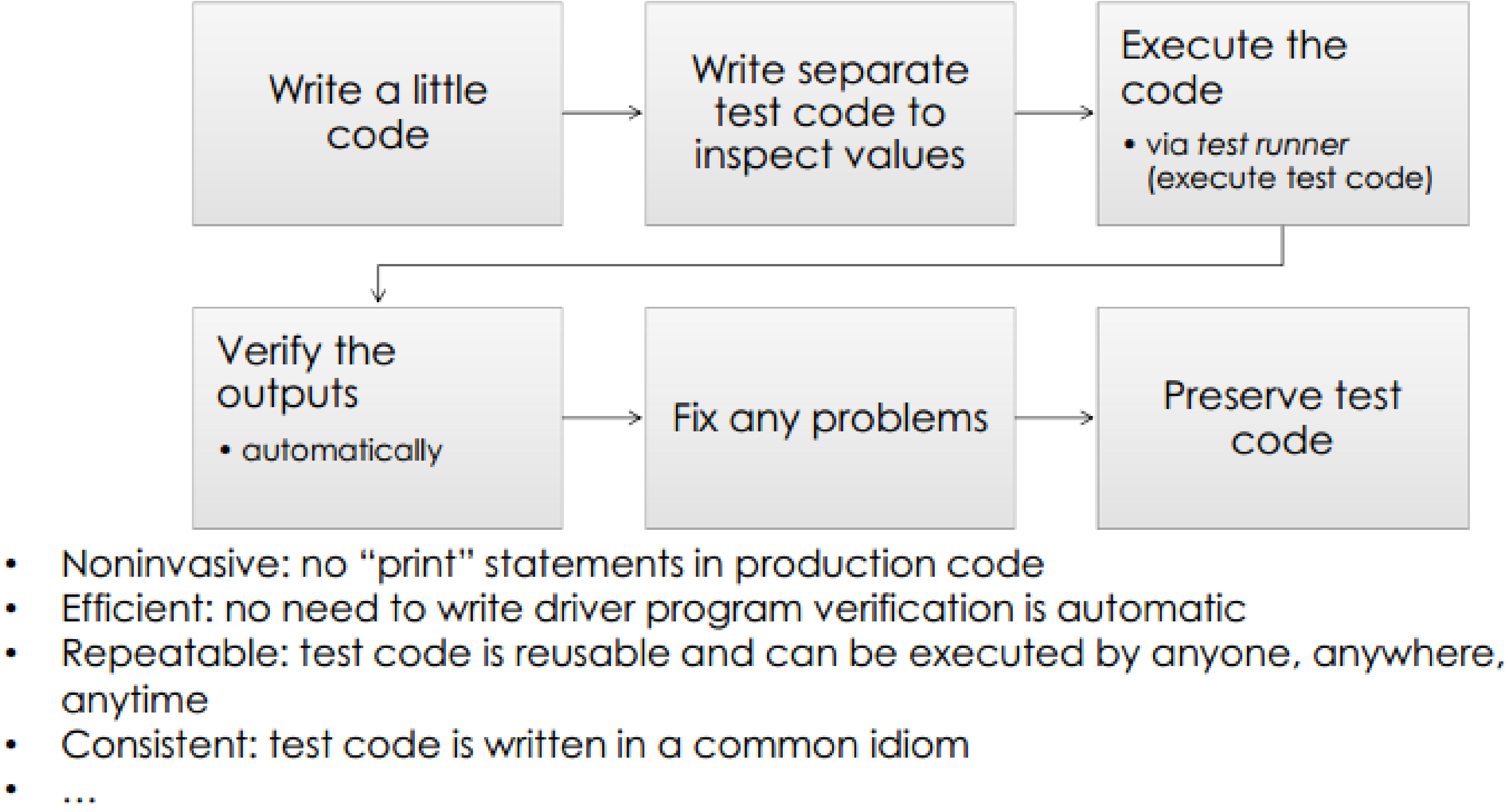
Fix any problems

Throw test code away
- or exit the debugger

- Invasive: risk of introducing bugs in production code while adding print statements (which might as well remain in production code)
- Inefficient: manual verification and lack of partial execution
- Unrepeatable: print statements removed, test code thrown away
- Inconsistent: different people write test code differently

Adapted from (Subramaniam, 2011)

Agile Testing Feedback



Write a little code

Write separate test code to inspect values

Execute the code

- via *test runner* (execute test code)

Verify the outputs

- automatically

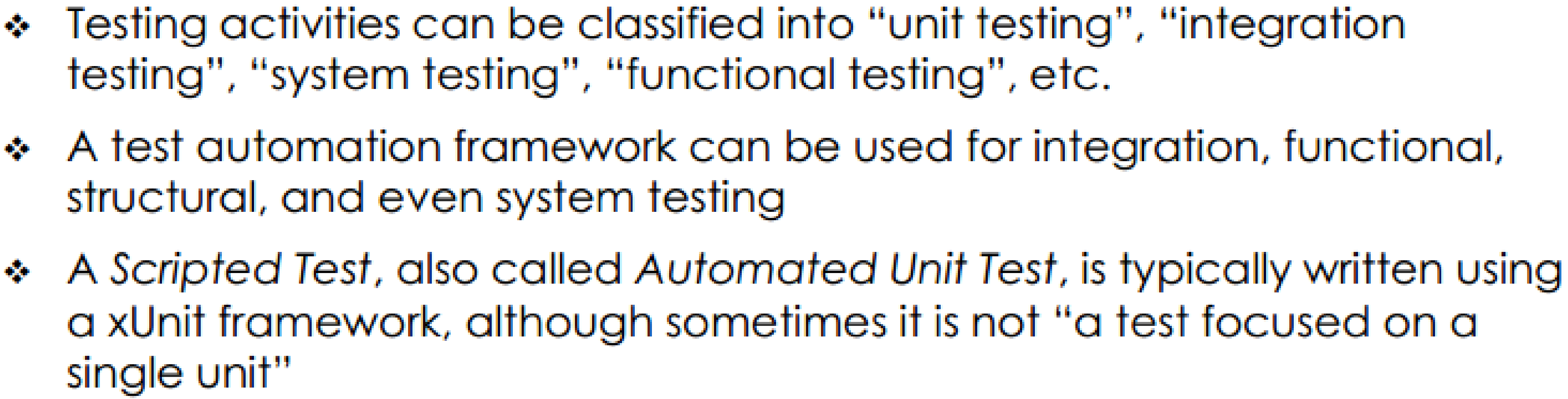
Fix any problems

Preserve test code

- Noninvasive: no “print” statements in production code
- Efficient: no need to write driver program verification is automatic
- Repeatable: test code is reusable and can be executed by anyone, anywhere, anytime
- Consistent: test code is written in a common idiom
- ...

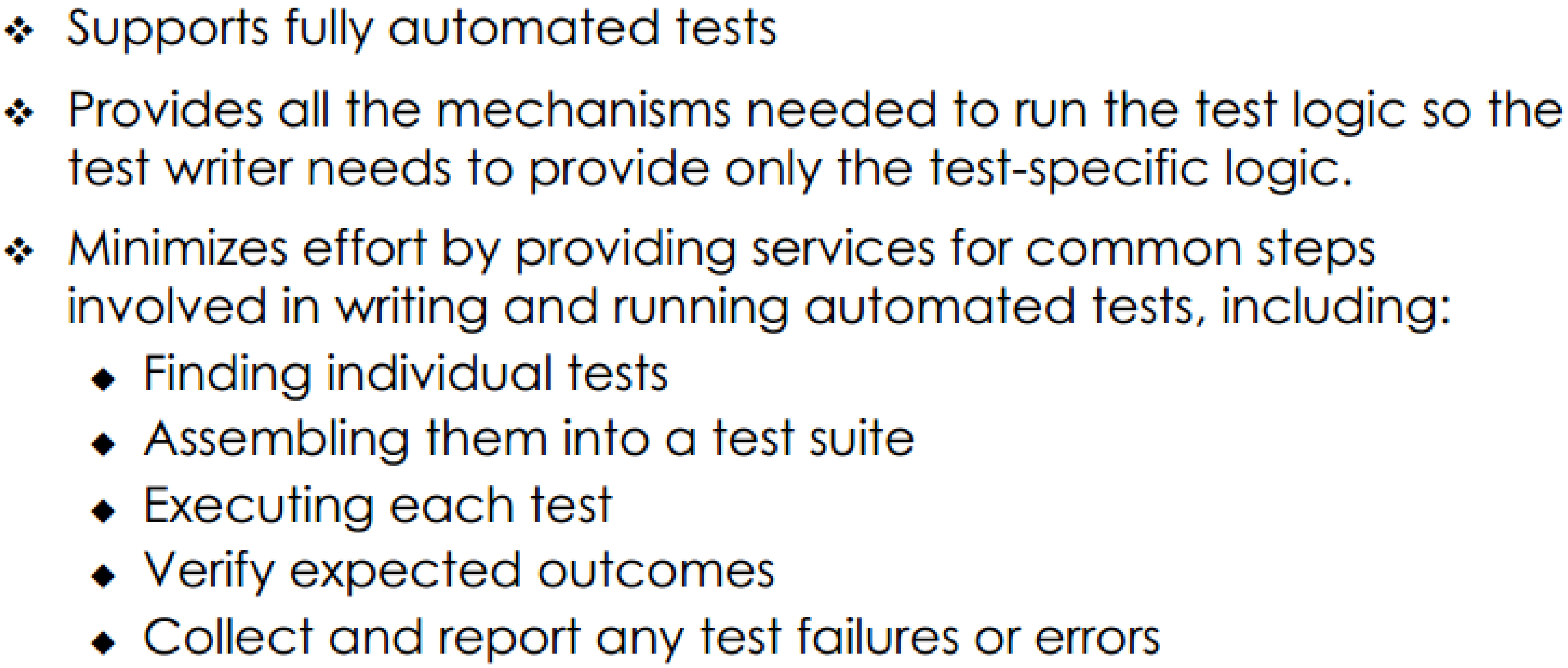
- In Test First (TFD), reverse the first two steps

Test Automation Framework



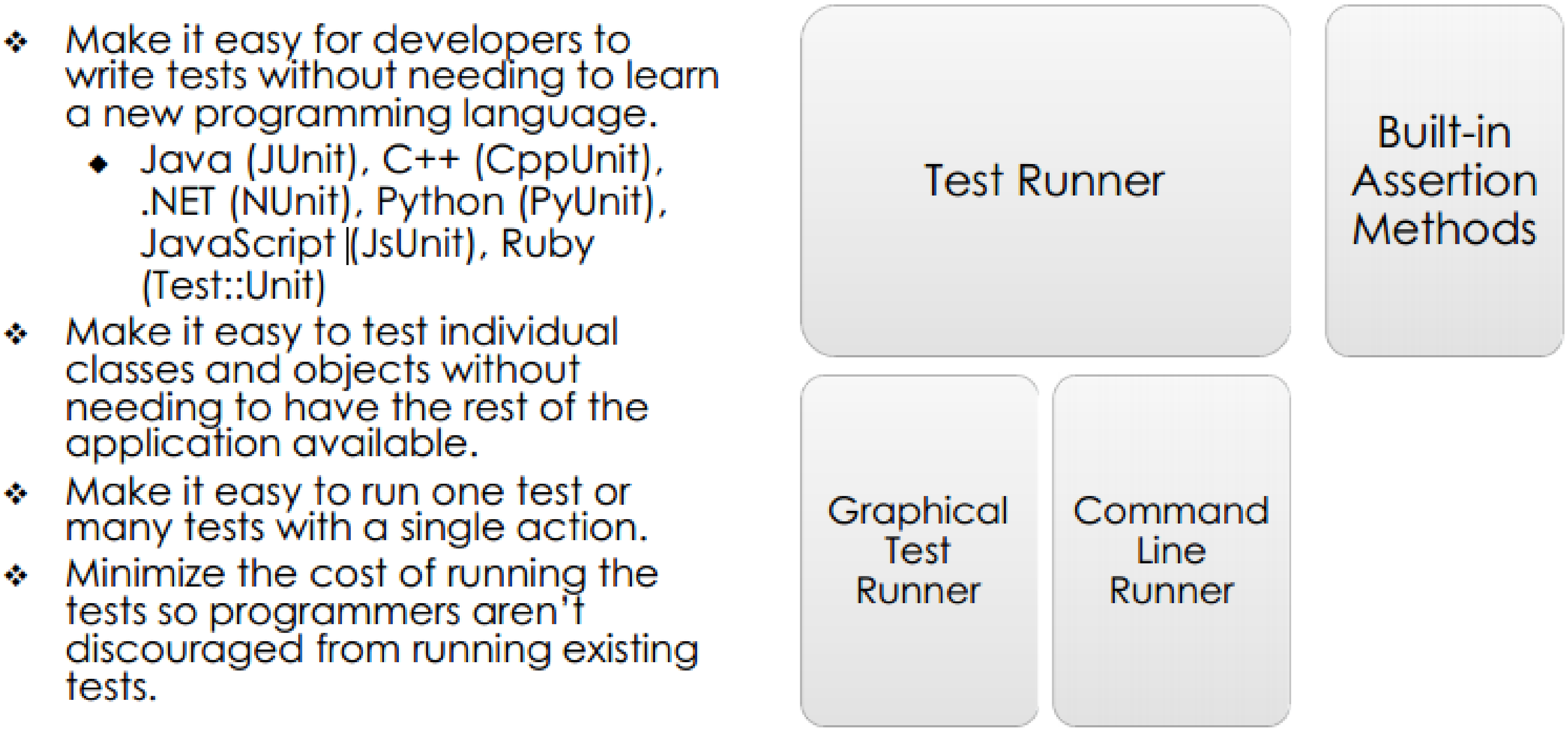
- Testing activities can be classified into "unit testing", "integration testing", "system testing", "functional testing", etc.
- A test automation framework can be used for integration, functional, structural, and even system testing
- A *Scripted Test*, also called *Automated Unit Test*, is typically written using a xUnit framework, although sometimes it is not "a test focused on a single unit"

Test Automation Framework (2)



- Supports fully automated tests
- Provides all the mechanisms needed to run the test logic so the test writer needs to provide only the test-specific logic.
- Minimizes effort by providing services for common steps involved in writing and running automated tests, including:
  - Finding individual tests
  - Assembling them into a test suite
  - Executing each test
  - Verify expected outcomes
  - Collect and report any test failures or errors

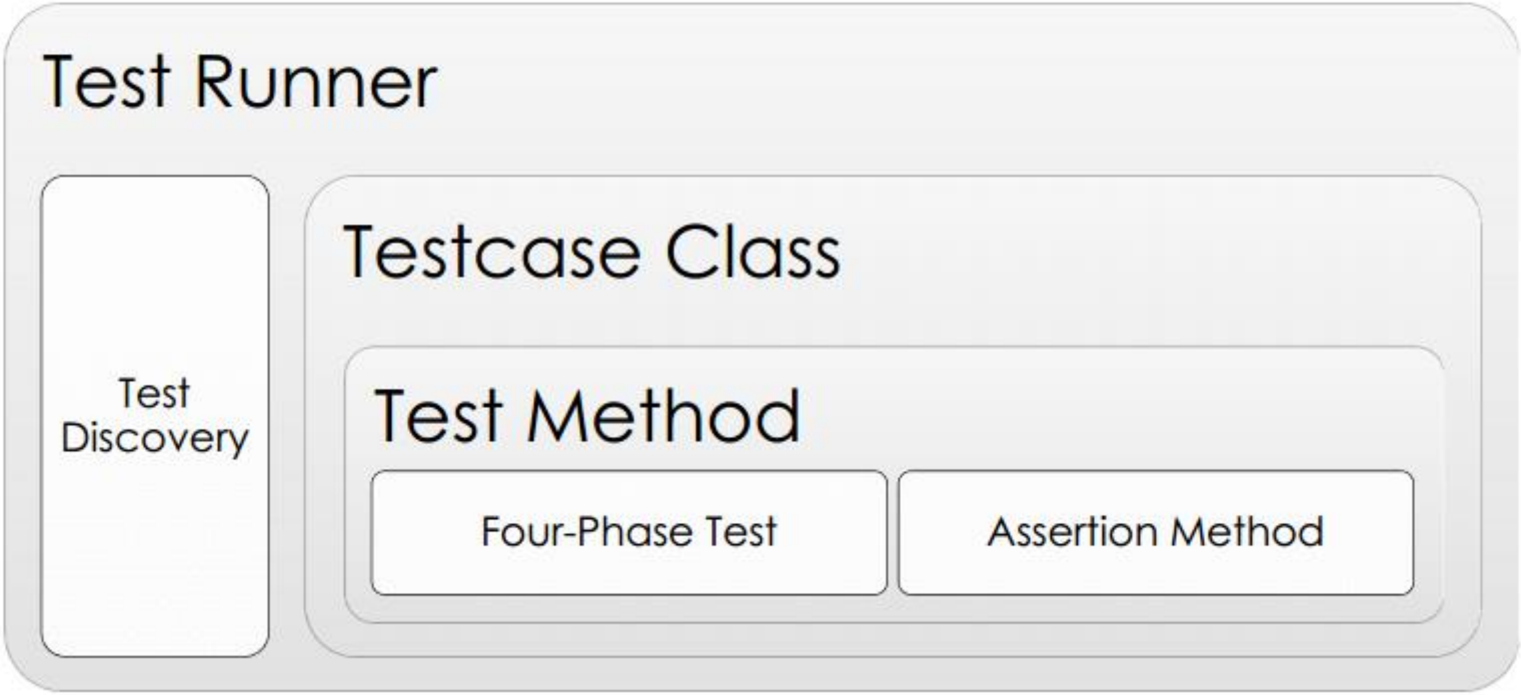
xUnit Frameworks



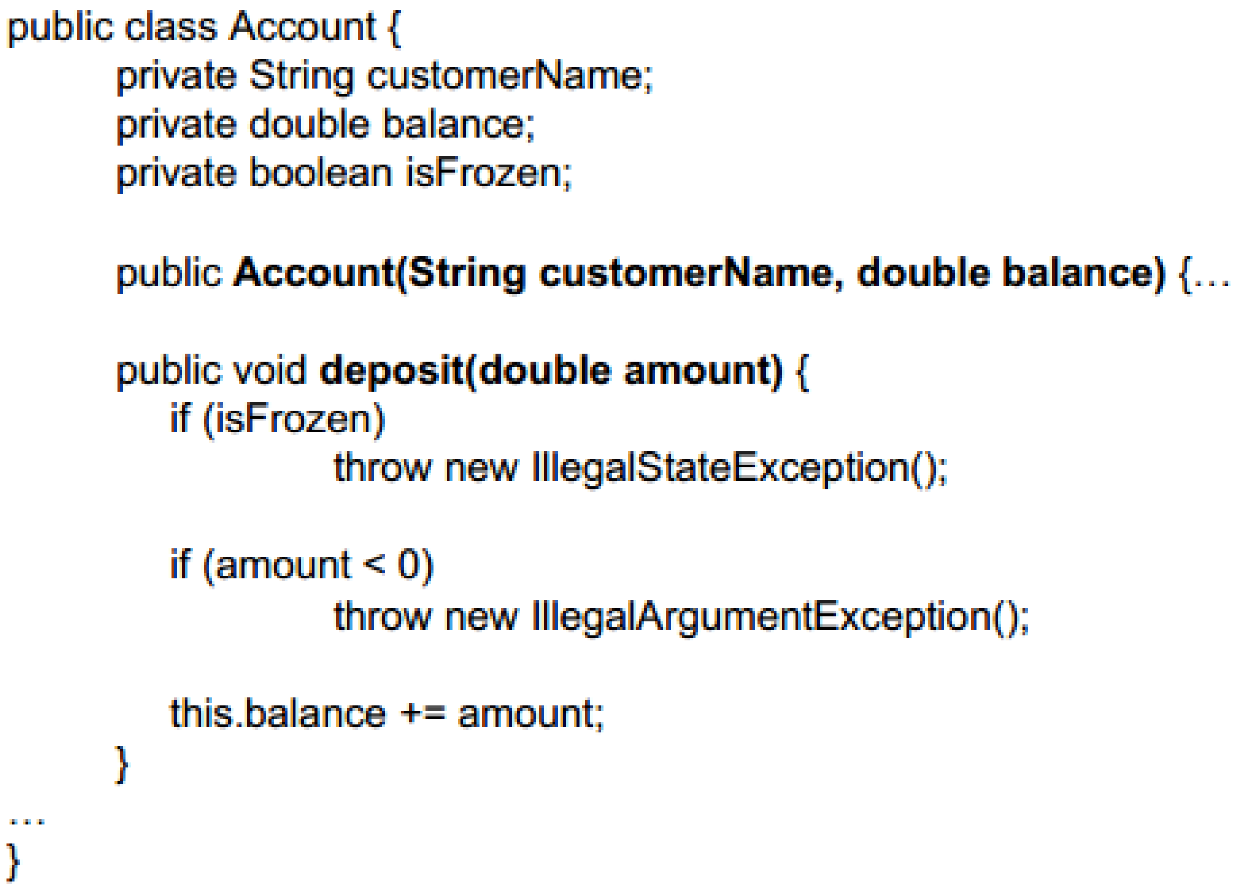
- Make it easy for developers to write tests without needing to learn a new programming language.
    - Java (JUnit), C++ (CppUnit), .NET (NUnit), Python (PyUnit), JavaScript |(JsUnit), Ruby (Test::Unit)
- Make it easy to test individual classes and objects without needing to have the rest of the application available.
- Make it easy to run one test or many tests with a single action.
- Minimize the cost of running the tests so programmers aren't discouraged from running existing tests.

- See www for more information on each of the language-specific frameworks

xUnit Basics: JUnit

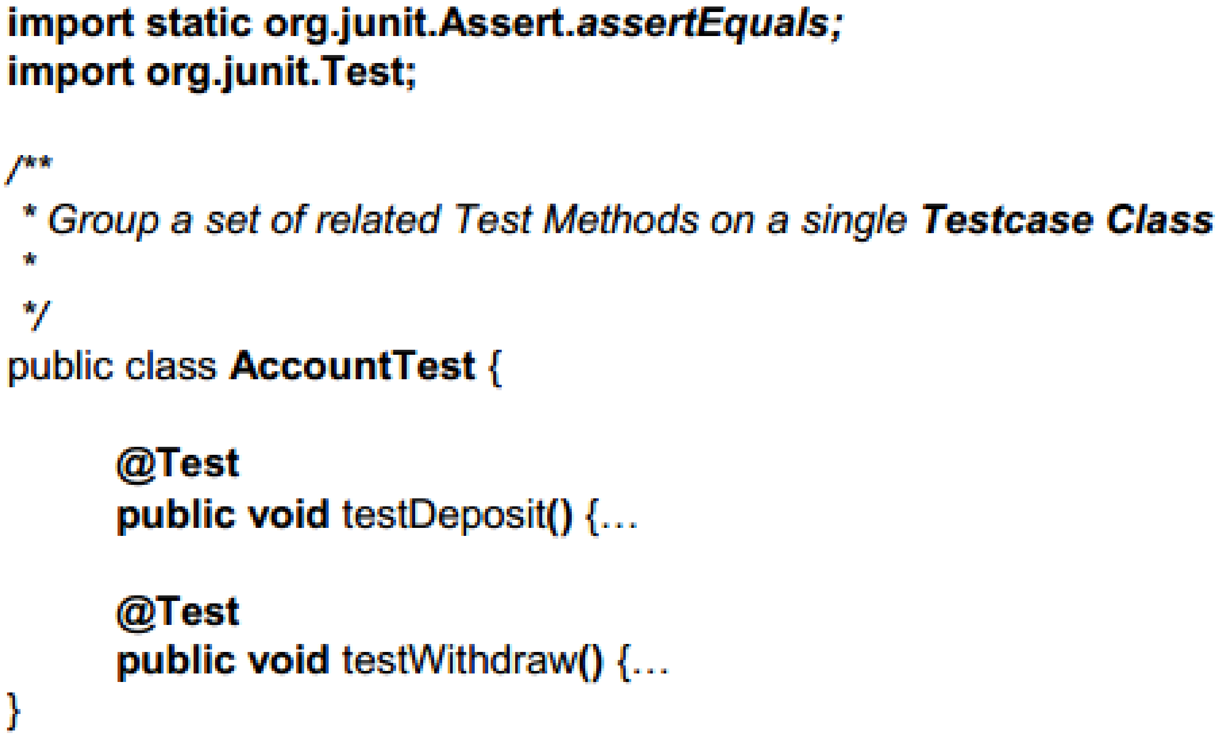


System (Component, Unit) under test: Example



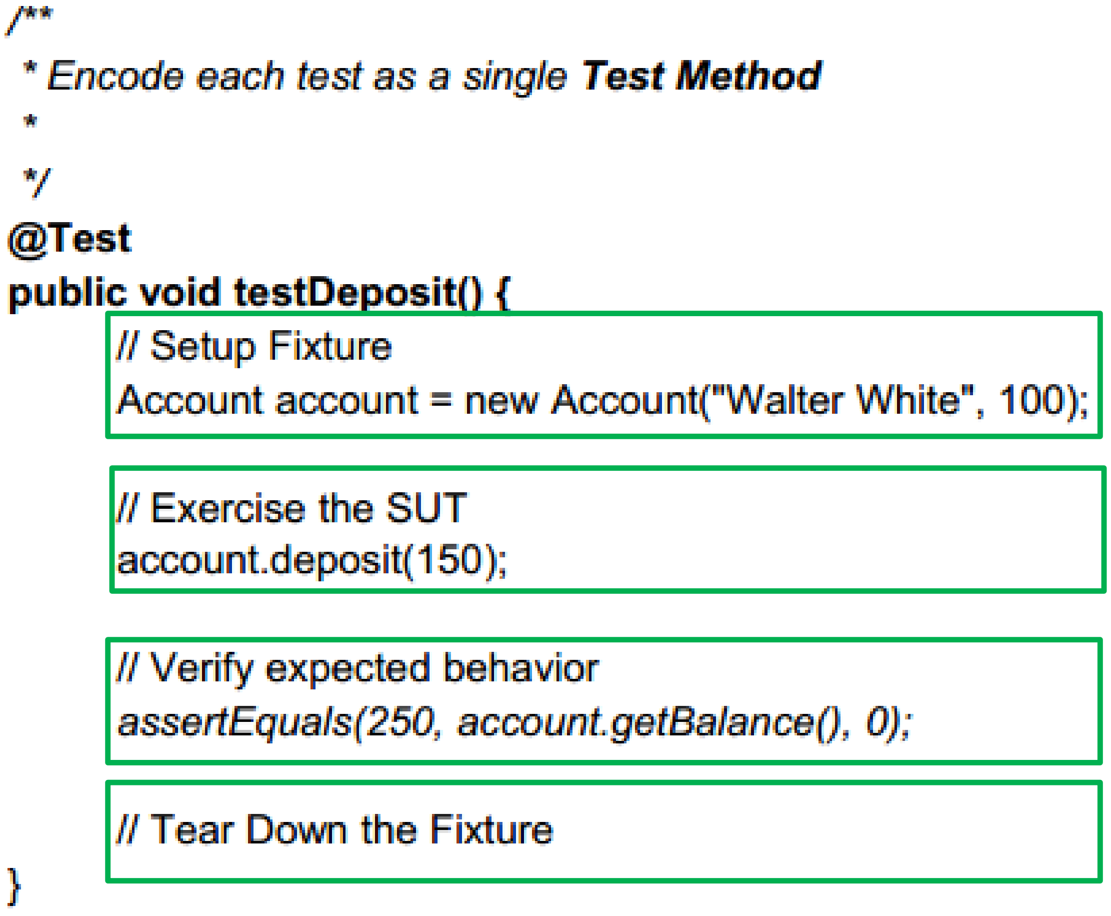
```
public class Account {
        private String customerName;
        private double balance;
        private boolean isFrozen;

        public Account(String customerName, double balance) {…

        public void deposit(double amount) {
            if (isFrozen)
                        throw new IllegalStateException();

            if (amount < 0)
                        throw new IllegalArgumentException();

            this.balance += amount;
        }
…
}
```

Test Case Class



```
import static org.junit.Assert.assertEquals;
import org.junit.Test;

/**
 * Group a set of related Test Methods on a single Testcase Class
 *
 */
public class AccountTest {

        @Test
        public void testDeposit() {…

        @Test
        public void testWithdraw() {…
}
```

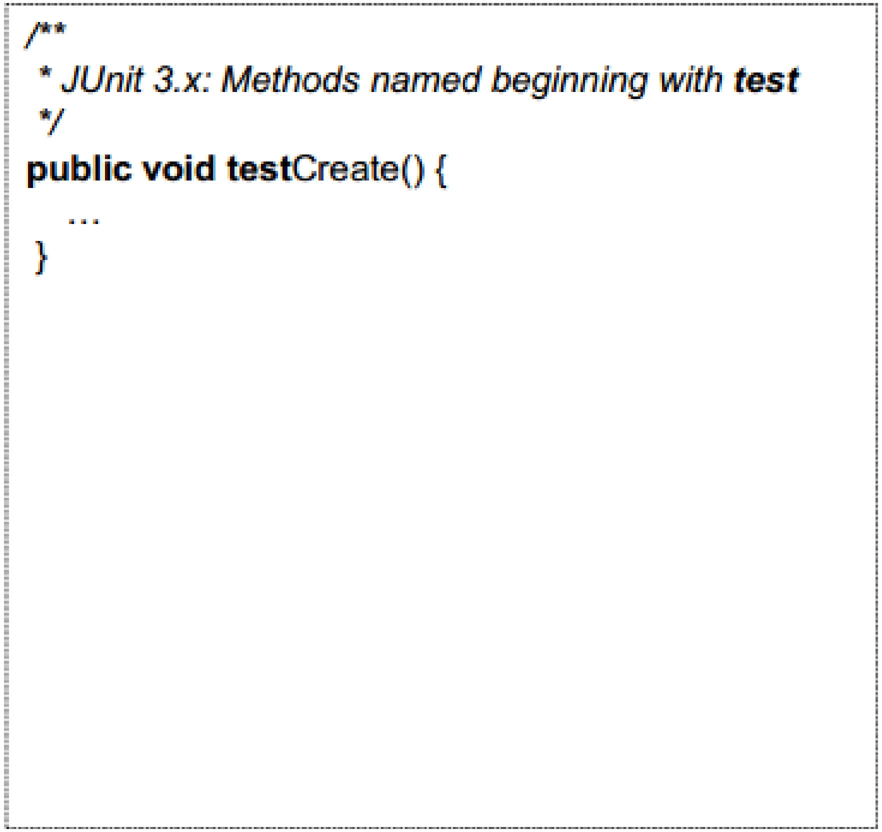
Test Method



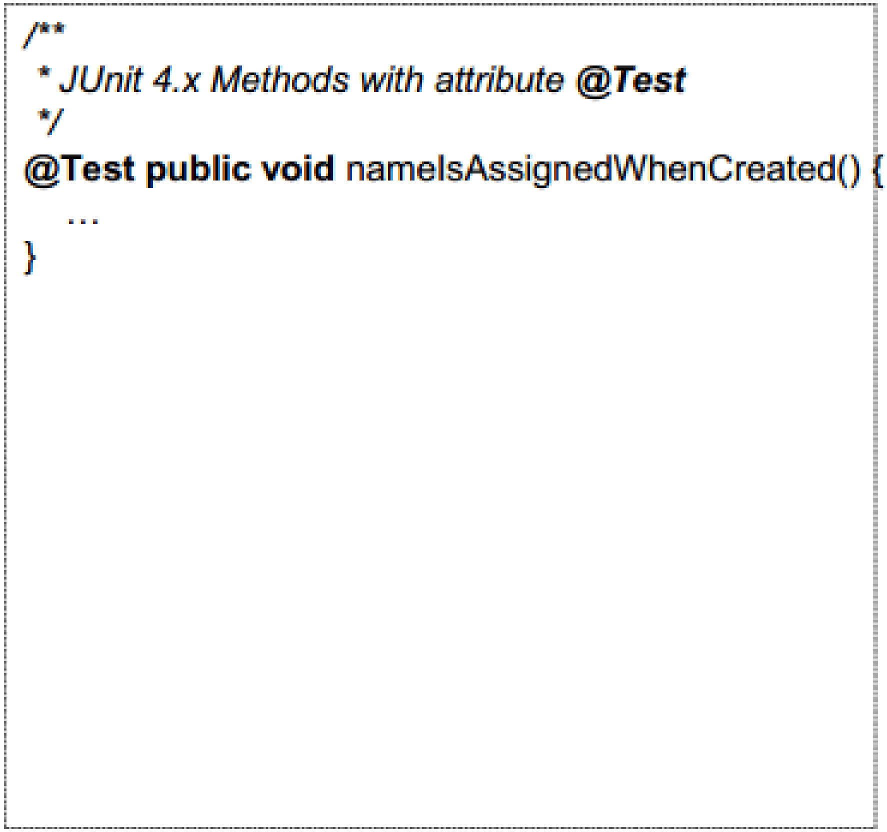
```
/**
 * Encode each test as a single Test Method
 *
 */
@Test
public void testDeposit() {
        // Setup Fixture
        Account account = new Account("Walter White", 100);

        // Exercise the SUT
        account.deposit(150);

        // Verify expected behavior
        assertEquals(250, account.getBalance(), 0);

        // Tear Down the Fixture
}
```

- Four phases:
  set up fixture; exercise the SUT; verify expected behavior; tear down fixture

Test Discovery

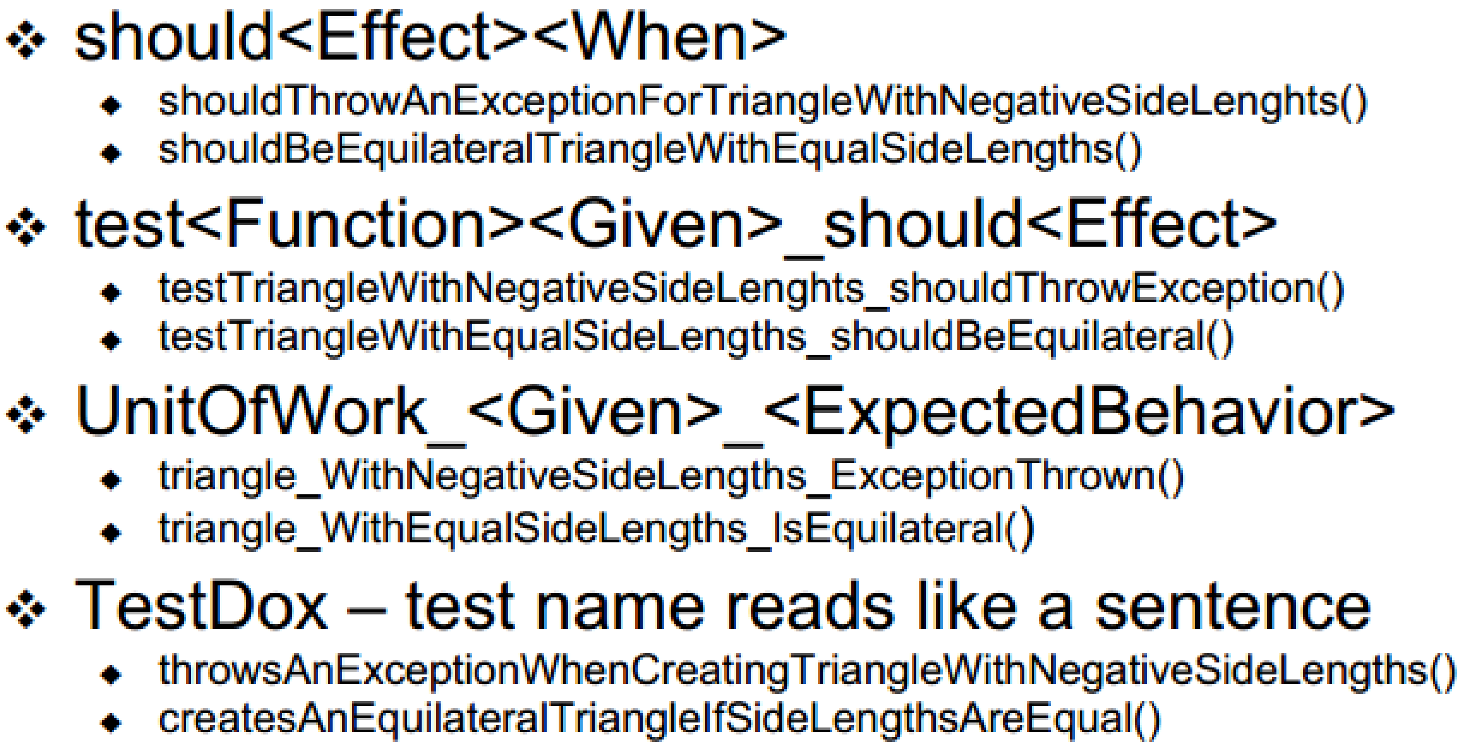


```
/**
 * JUnit 3.x: Methods named beginning with test
 */
public void testCreate() {
    ...
}
```



```
/**
 * JUnit 4.x Methods with attribute @Test
 */
@Test public void nameIsAssignedWhenCreated() {
    ...
}
```

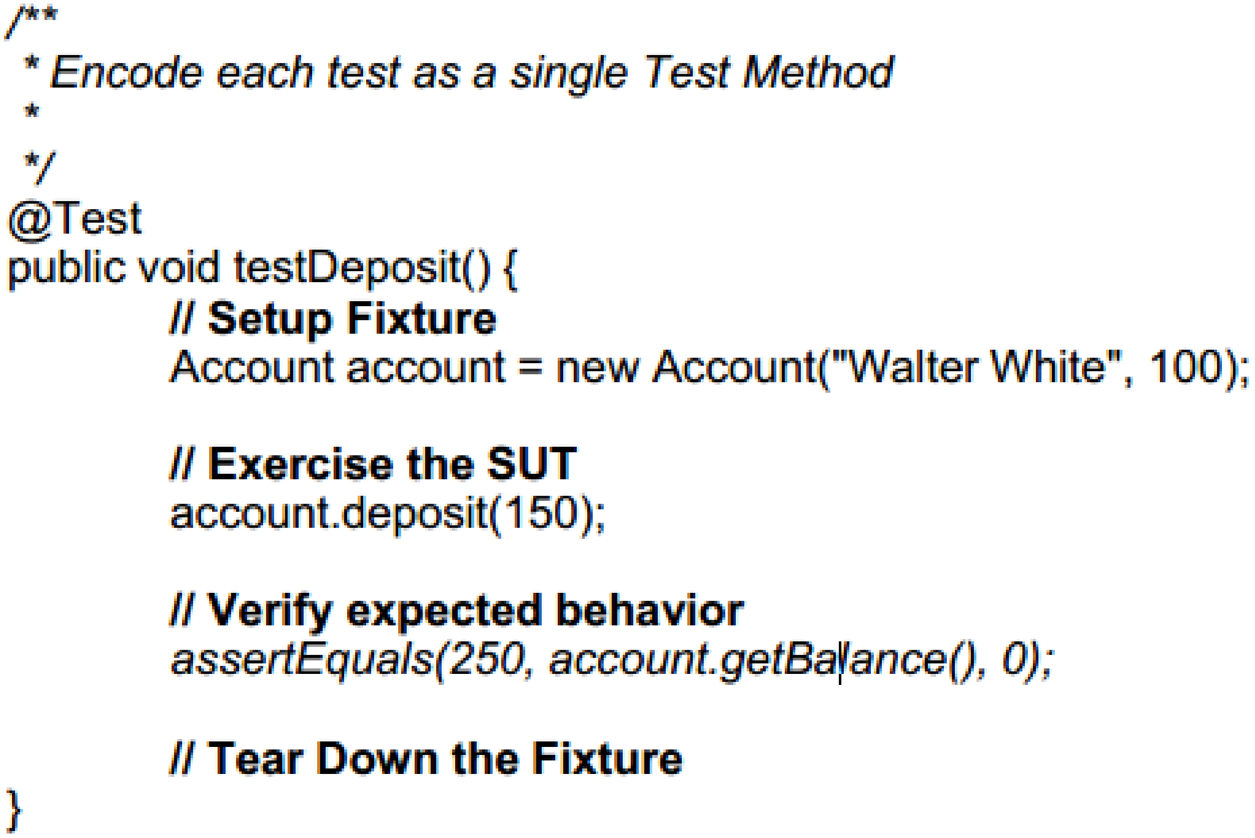
Naming Unit Tests



- should<Effect><When>
  - shouldThrowAnExceptionForTriangleWithNegativeSideLenghts()
  - shouldBeEquilateralTriangleWithEqualSideLengths()
- test<Function><Given>_should<Effect>
  - testTriangleWithNegativeSideLenghts_shouldThrowException()
  - testTriangleWithEqualSideLengths_shouldBeEquilateral()
- UnitOfWork_<Given>_<ExpectedBehavior>
  - triangle_WithNegativeSideLengths_ExceptionThrown()
  - triangle_WithEqualSideLengths_IsEquilateral()
- TestDox – test name reads like a sentence
  - throwsAnExceptionWhenCreatingTriangleWithNegativeSideLengths()
  - createsAnEquilateralTriangleIfSideLengthsAreEqual()

- *Your Coding Standard/Style Guide should specify a Unit Test naming convention*

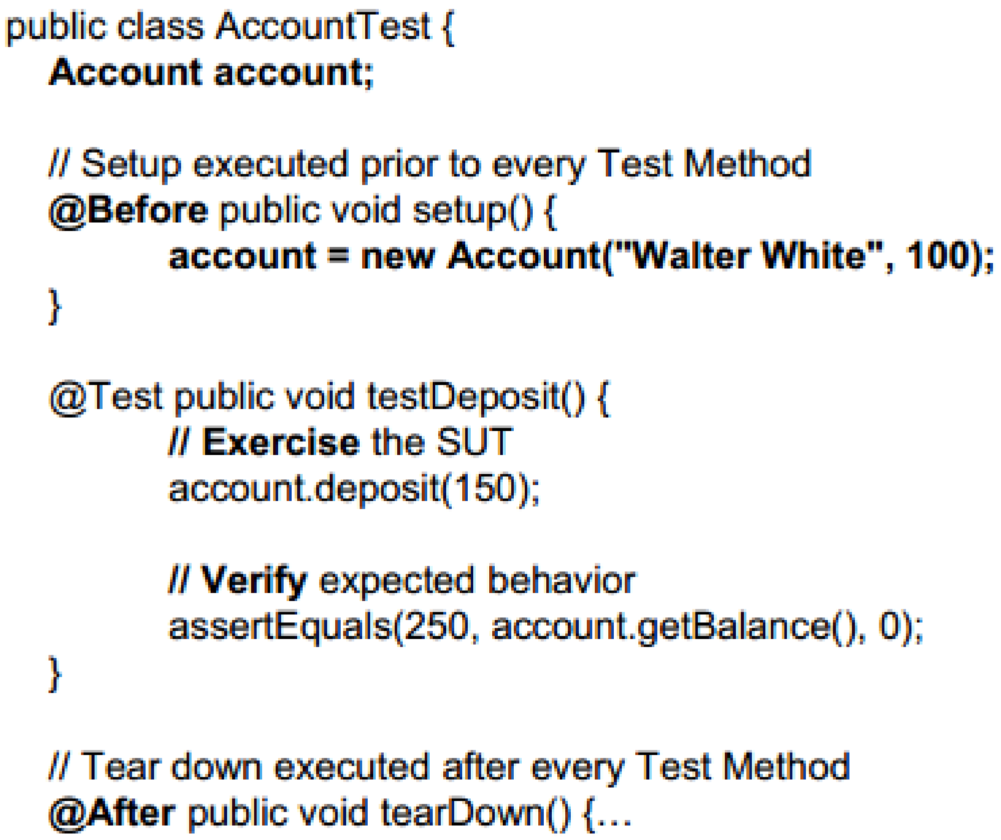
Four Phase Test -- In Line



```
/**
 * Encode each test as a single Test Method
 *
 */
@Test
public void testDeposit() {
        // Setup Fixture
        Account account = new Account("Walter White", 100);

        // Exercise the SUT
        account.deposit(150);

        // Verify expected behavior
        assertEquals(250, account.getBalance(), 0);

        // Tear Down the Fixture
}
```

Non scholae sed vitae discimus.

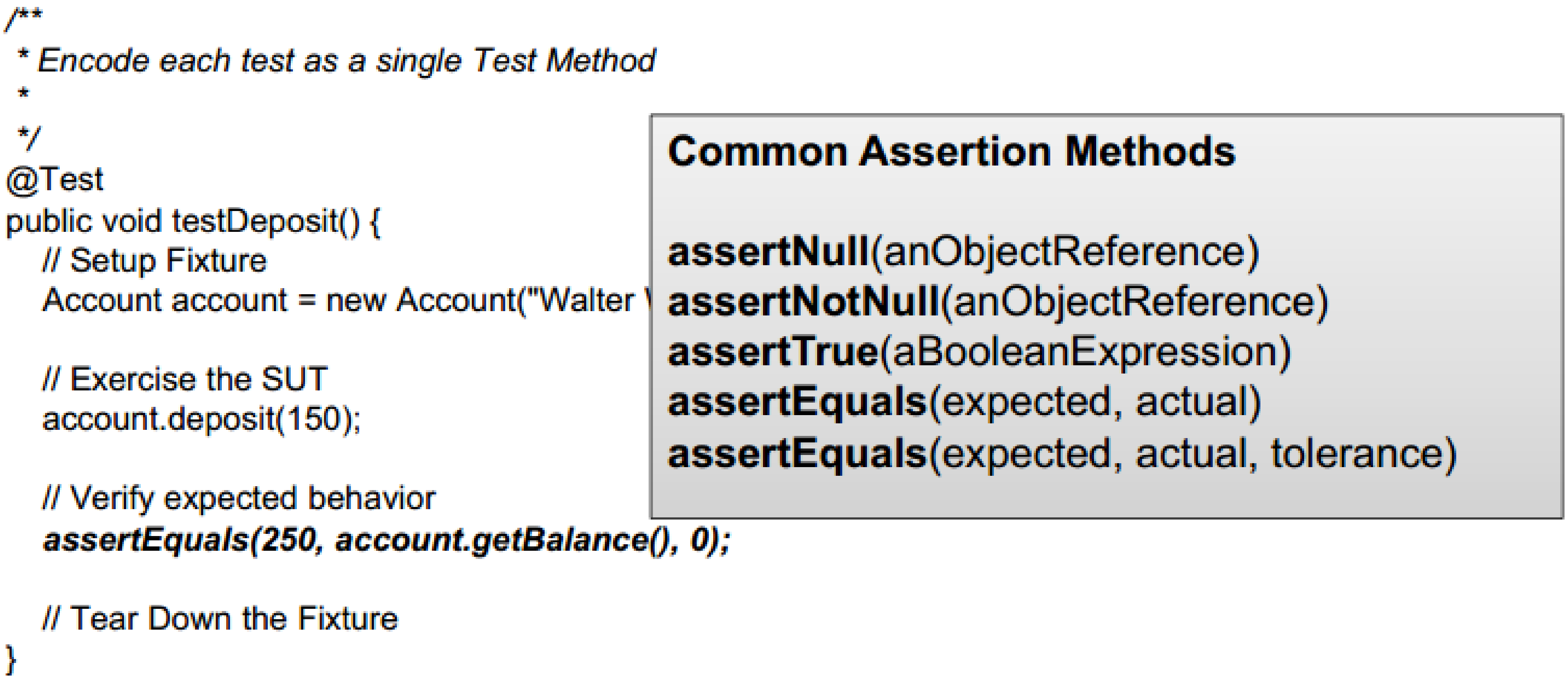
Four-Phase Test – Implicit Setup/Teardown



```
public class AccountTest {
    Account account;

    // Setup executed prior to every Test Method
    @Before public void setup() {
            account = new Account("Walter White", 100);
    }

    @Test public void testDeposit() {
            // Exercise the SUT
            account.deposit(150);

            // Verify expected behavior
            assertEquals(250, account.getBalance(), 0);
    }

    // Tear down executed after every Test Method
    @After public void tearDown() {…
```

- Code for Setup and Teardown phase is factored out of individual tests; the test runner executes the setup phase before and the teardown phase after each test method

Assertion Method

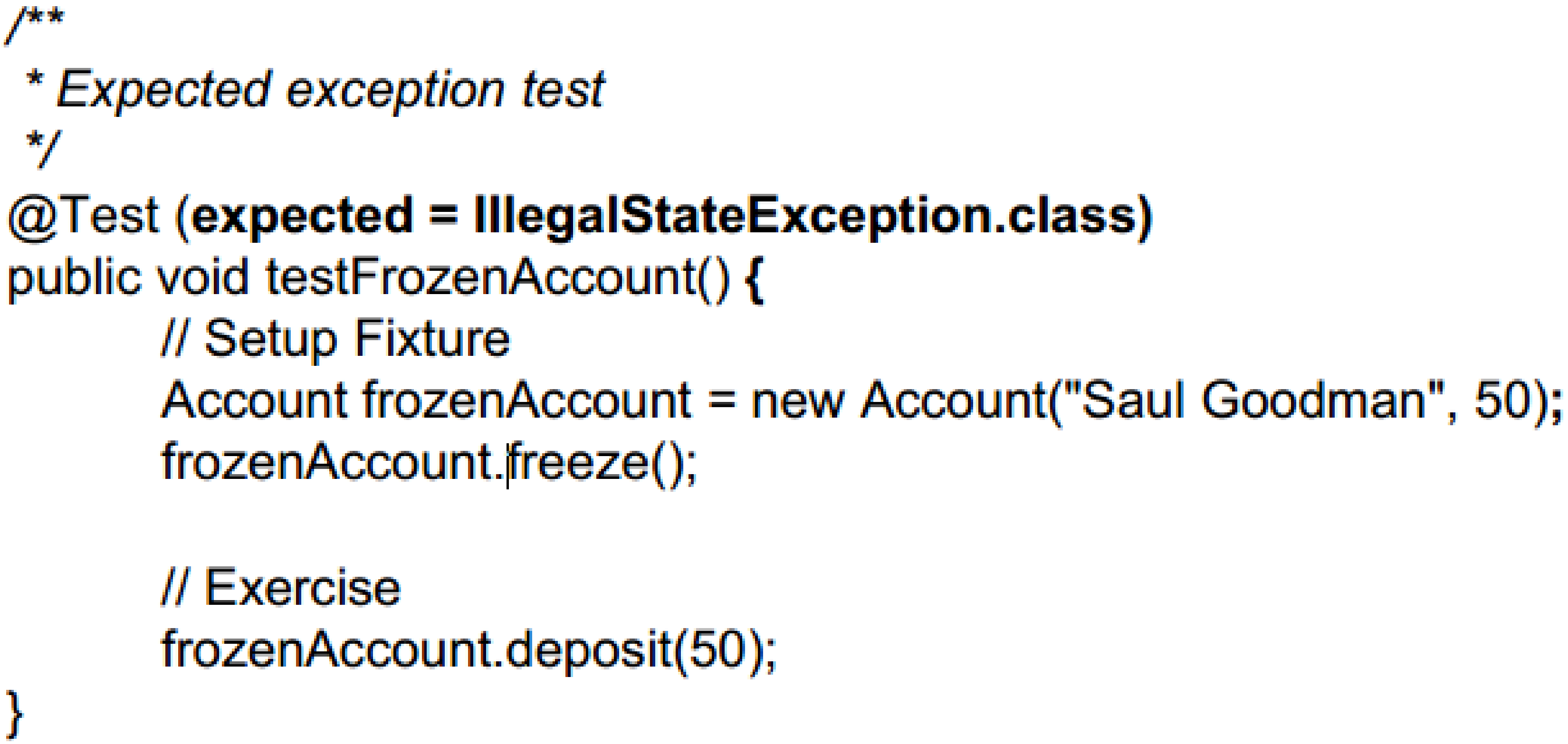


```
/**
 * Encode each test as a single Test Method
 *
 */
@Test
public void testDeposit() {
   // Setup Fixture
   Account account = new Account("Walter

   // Exercise the SUT
   account.deposit(150);

   // Verify expected behavior
   assertEquals(250, account.getBalance(), 0);

   // Tear Down the Fixture
}
```

Common Assertion Methods

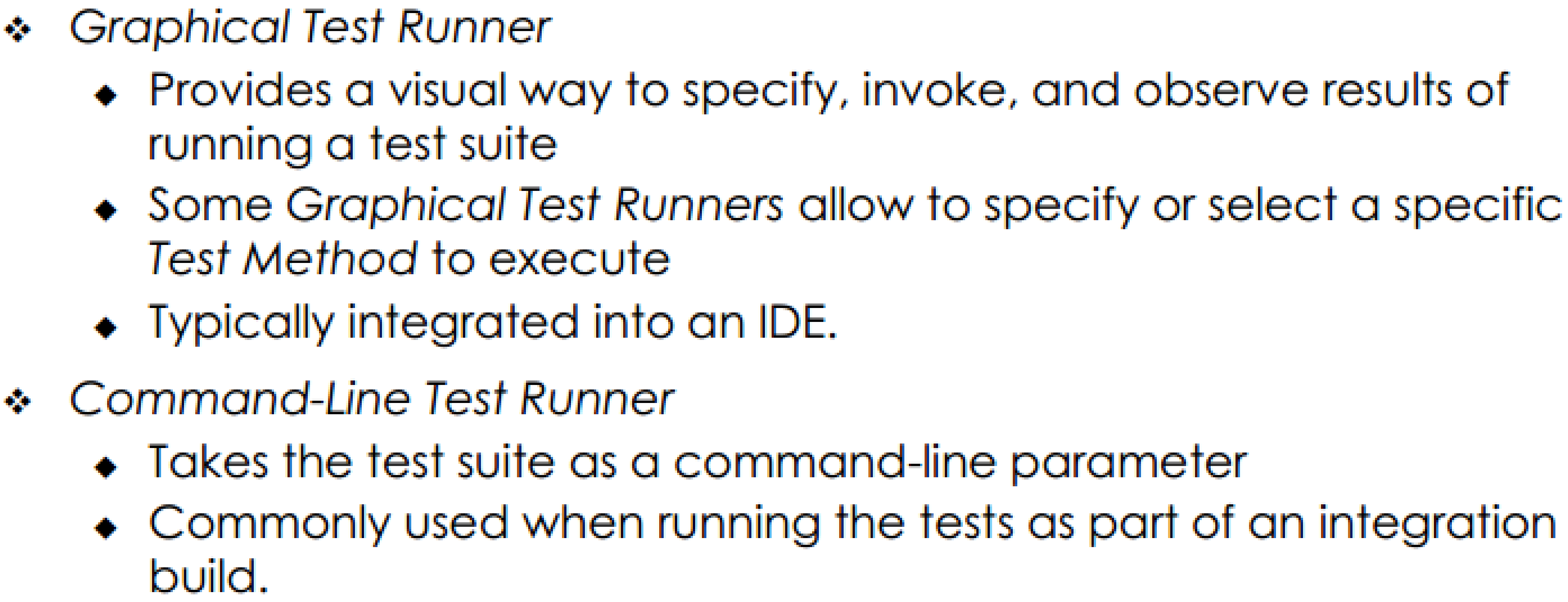
assertNull(anObjectReference)
assertNotNull(anObjectReference)
assertTrue(aBooleanExpression)
assertEquals(expected, actual)
assertEquals(expected, actual, tolerance)

Assertion Method (Expected Exception)



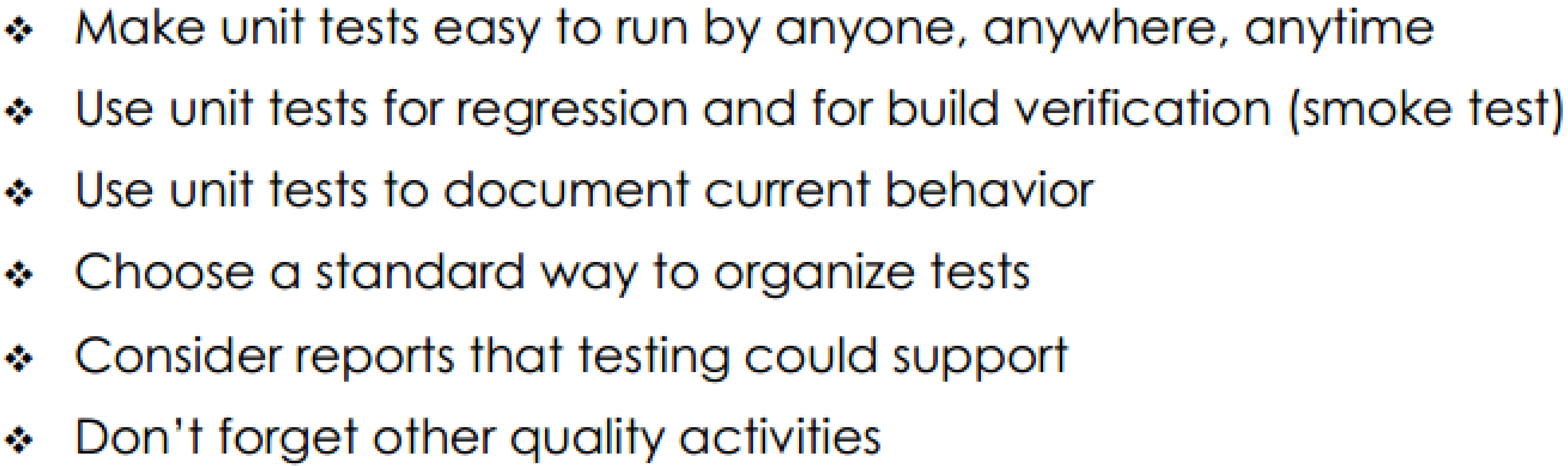
```
/**
 * Expected exception test
 */
@Test (expected = IllegalStateException.class)
public void testFrozenAccount() {
        // Setup Fixture
        Account frozenAccount = new Account("Saul Goodman", 50);
        frozenAccount.freeze();

        // Exercise
        frozenAccount.deposit(50);
}
```

Running xUnit Tests



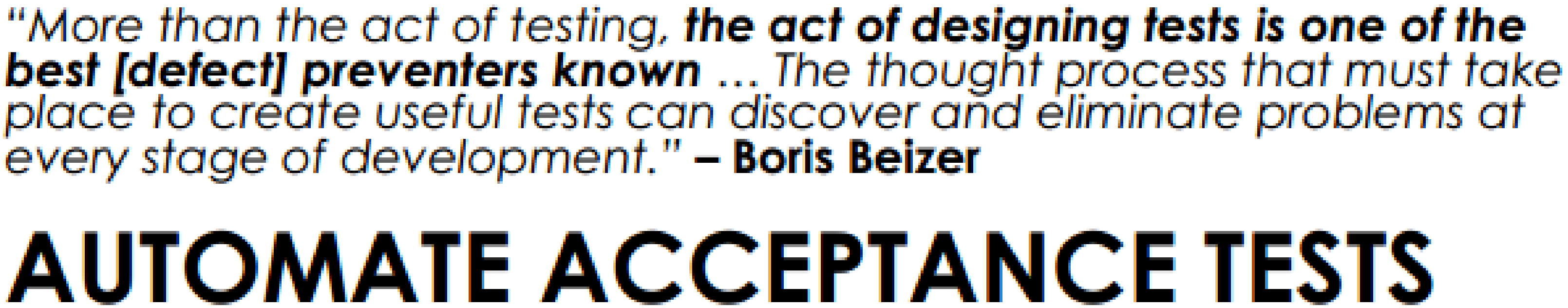
- *Graphical Test Runner*
  - Provides a visual way to specify, invoke, and observe results of running a test suite
  - Some *Graphical Test Runners* allow to specify or select a specific *Test Method* to execute
  - Typically integrated into an IDE.
- *Command-Line Test Runner*
  - Takes the test suite as a command-line parameter
  - Commonly used when running the tests as part of an integration build.

Automated Unit Test Best Practices



- Make unit tests easy to run by anyone, anywhere, anytime
- Use unit tests for regression and for build verification (smoke test)
- Use unit tests to document current behavior
- Choose a standard way to organize tests
- Consider reports that testing could support
- Don't forget other quality activities

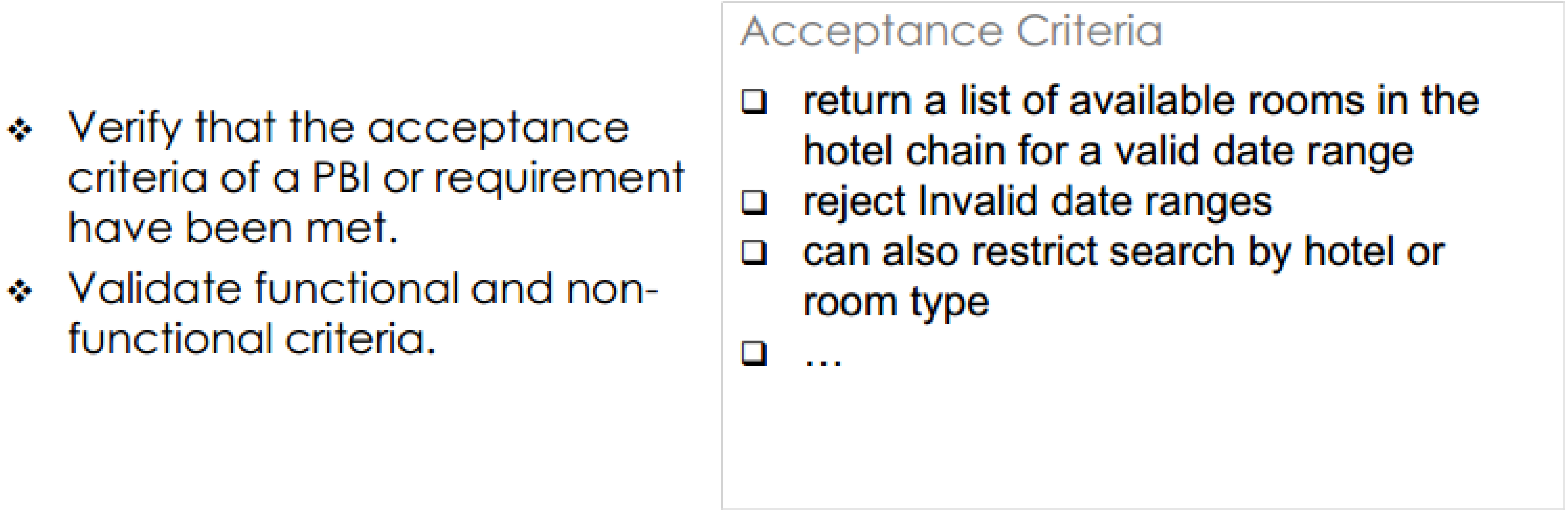
Automate Acceptance Tests



"More than the act of testing, **the act of designing tests is one of the best [defect] preventers known** … The thought process that must take place to create useful tests can discover and eliminate problems at every stage of development." **– Boris Beizer**

AUTOMATE ACCEPTANCE TESTS

The Role of Acceptance tests



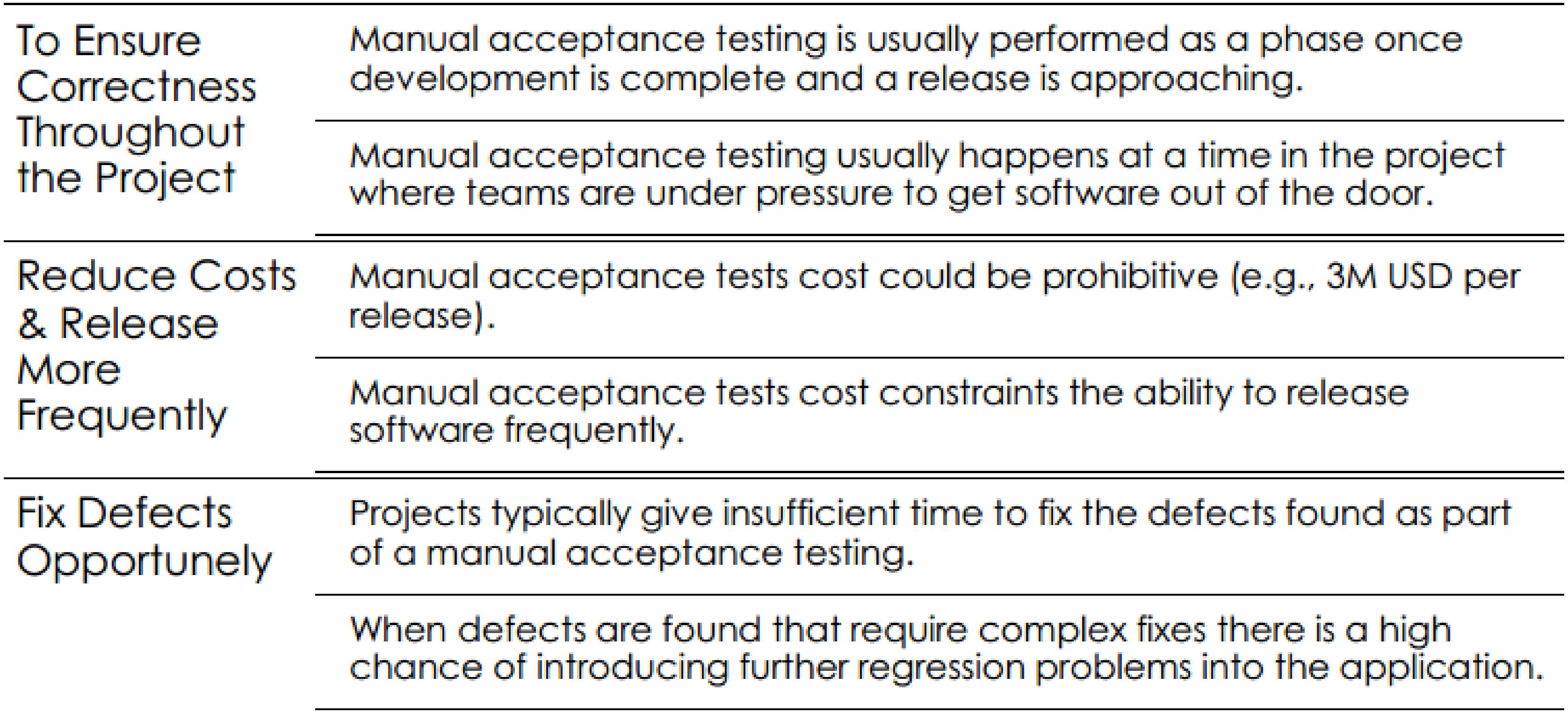
- Verify that the acceptance criteria of a PBI or requirement have been met.
- Validate functional and non-functional criteria.

Acceptance Criteria

- ❑ return a list of available rooms in the hotel chain for a valid date range
- ❑ reject Invalid date ranges
- ❑ can also restrict search by hotel or room type
- ❑ …

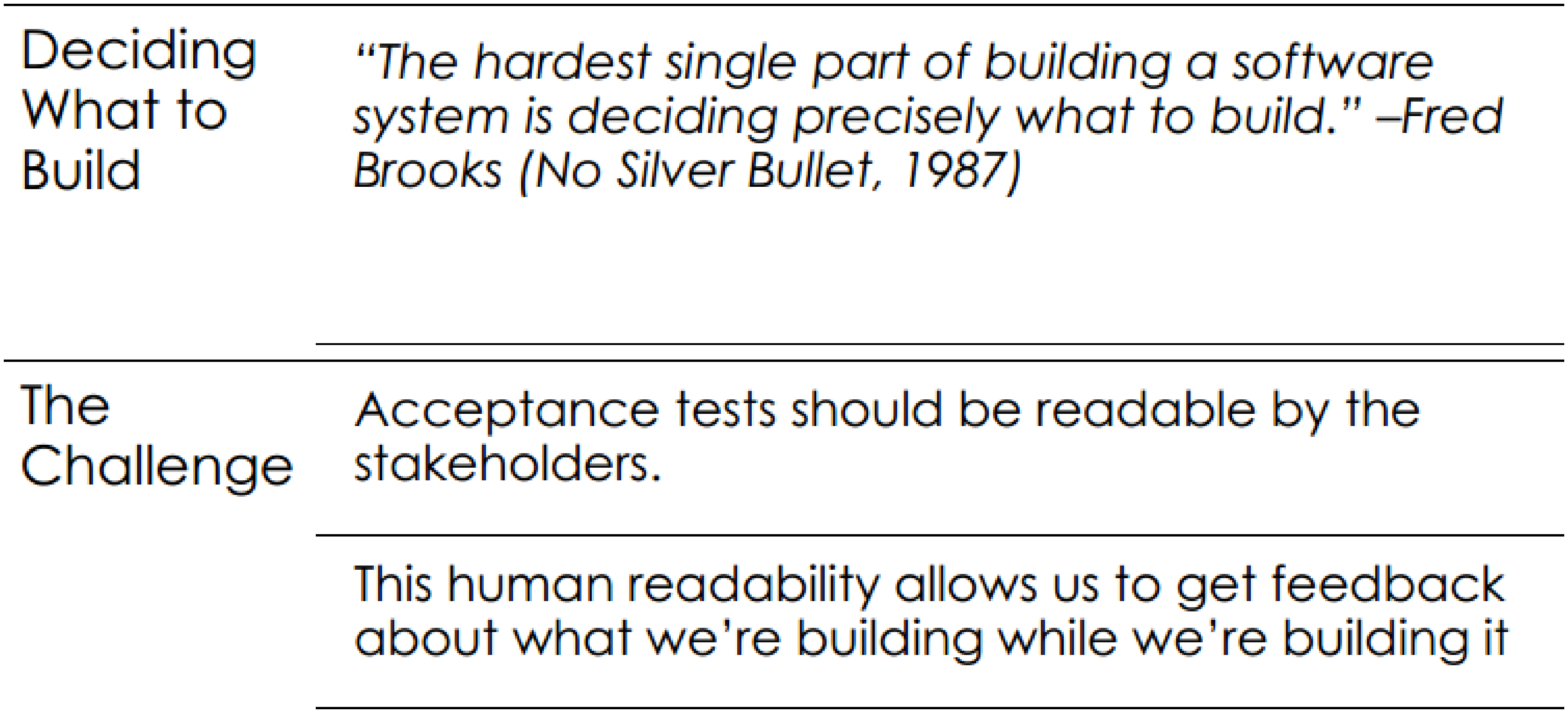
- PBI: Product Backlog Item (i.e. a user story)

Why automate Acceptance Tests



To Ensure Correctness Throughout the Project	Manual acceptance testing is usually performed as a phase once development is complete and a release is approaching.
	Manual acceptance testing usually happens at a time in the project where teams are under pressure to get software out of the door.
Reduce Costs & Release More Frequently	Manual acceptance tests cost could be prohibitive (e.g., 3M USD per release).
	Manual acceptance tests cost constraints the ability to release software frequently.
Fix Defects Opportunely	Projects typically give insufficient time to fix the defects found as part of a manual acceptance testing.
	When defects are found that require complex fixes there is a high chance of introducing further regression problems into the application.

Defining Acceptance Tests



Deciding What to Build	*"The hardest single part of building a software system is deciding precisely what to build." –Fred Brooks (No Silver Bullet, 1987)*
The Challenge	Acceptance tests should be readable by the stakeholders.
	This human readability allows us to get feedback about what we're building while we're building it

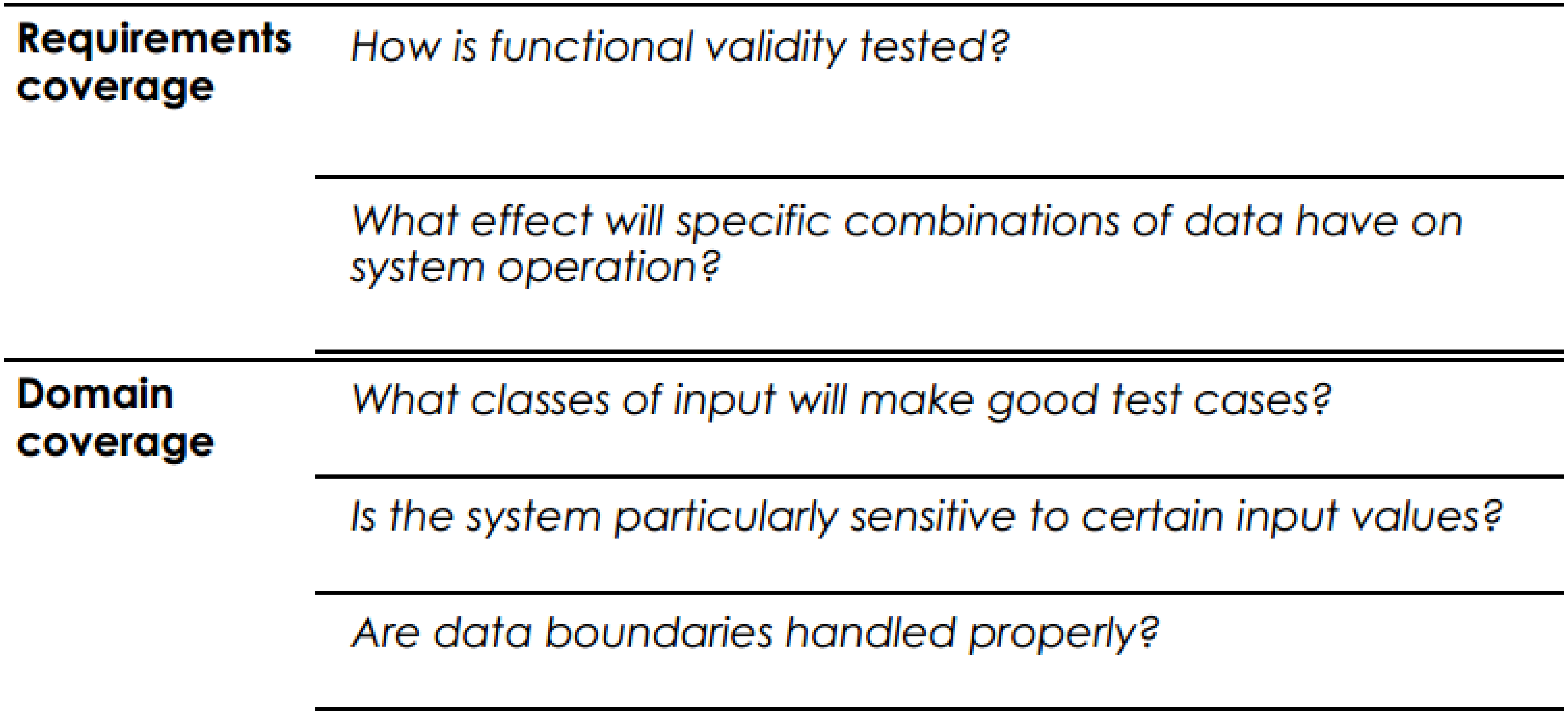
Acceptance Test is a “Black Box” test

77100
(taxableIncome)

15699
(taxOwed)

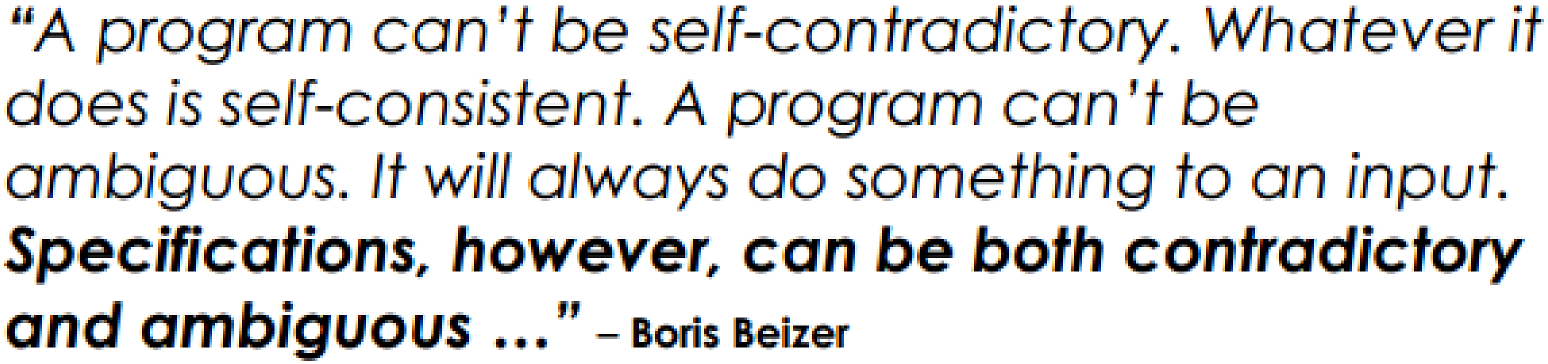
- Look at the external behavior of the software under test (SUT)
  - Ignore the internal structure of the SUT
- Attempt to find discrepancies between the SUT and its functional specifications
  - Derive test cases solely from the *specification* or the *documentation* of the SUT

Behavioral (“Black Box”) Testing Coverage



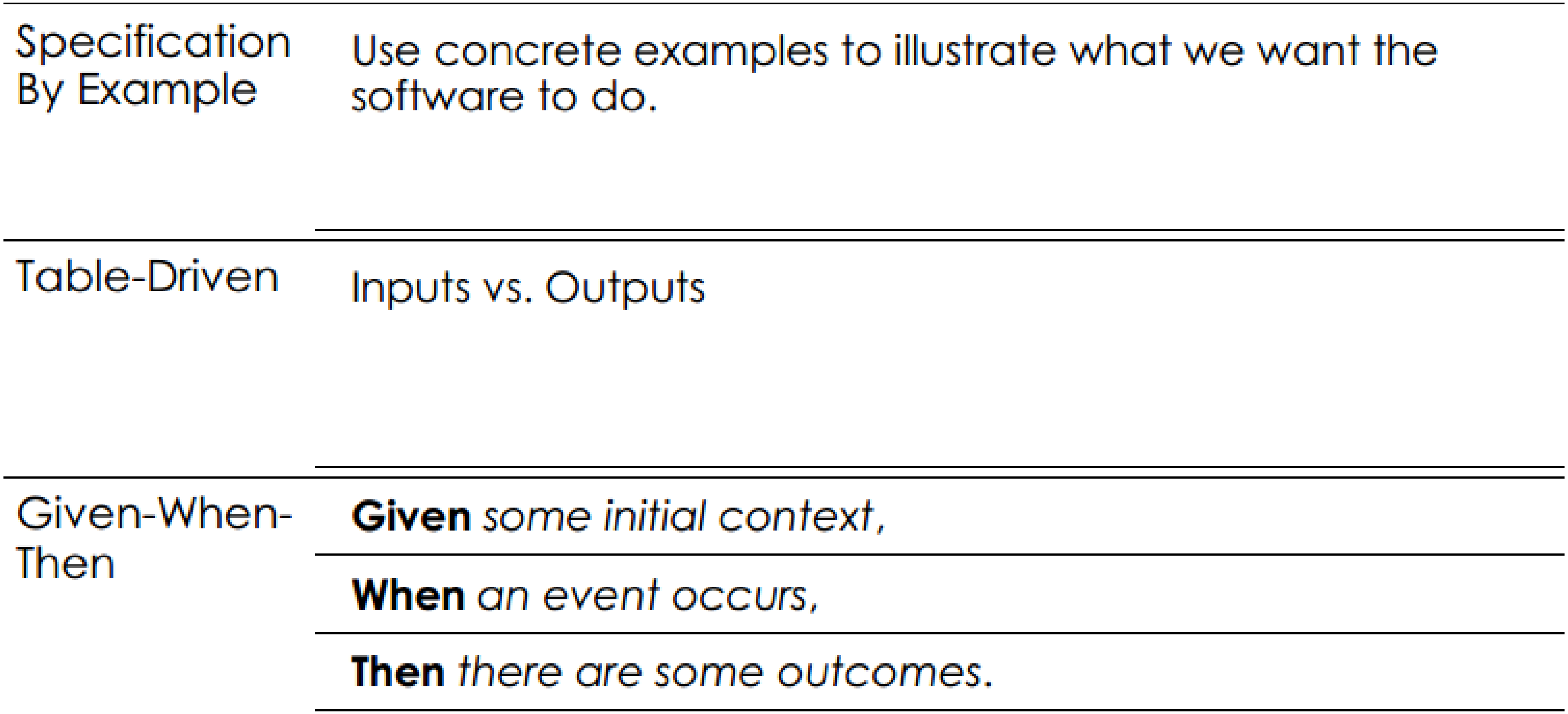
Requirements coverage	*How is functional validity tested?*
	What effect will specific combinations of data have on system operation?
Domain coverage	*What classes of input will make good test cases?*
	Is the system particularly sensitive to certain input values?
	Are data boundaries handled properly?

Functional Specifications may be lacking/problematic



"A program can't be self-contradictory. Whatever it does is self-consistent. A program can't be ambiguous. It will always do something to an input. **Specifications, however, can be both contradictory and ambiguous …"** – **Boris Beizer**

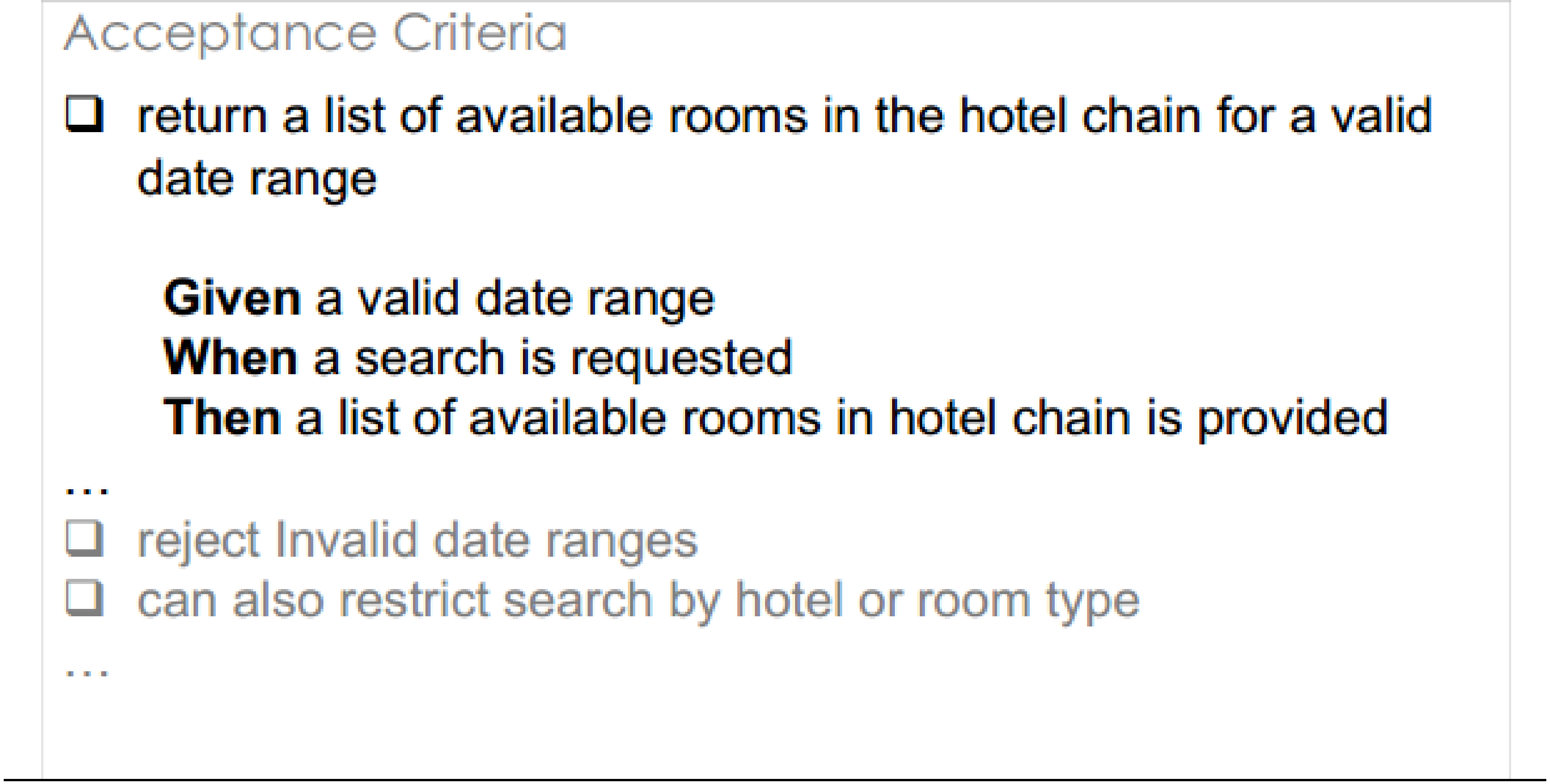
Test Acceptance Criteria with Specific Examples



Specification By Example	Use concrete examples to illustrate what we want the software to do.
Table-Driven	Inputs vs. Outputs
Given-When-Then	**Given** *some initial context,*
	When *an event occurs,*
	Then *there are some outcomes.*

- Specification by example: specific but not complete
- Testing based on examples:
  samples behavior, no general assurance of correct behavior

Given-When-Then Format



Acceptance Criteria

- ❑ return a list of available rooms in the hotel chain for a valid date range

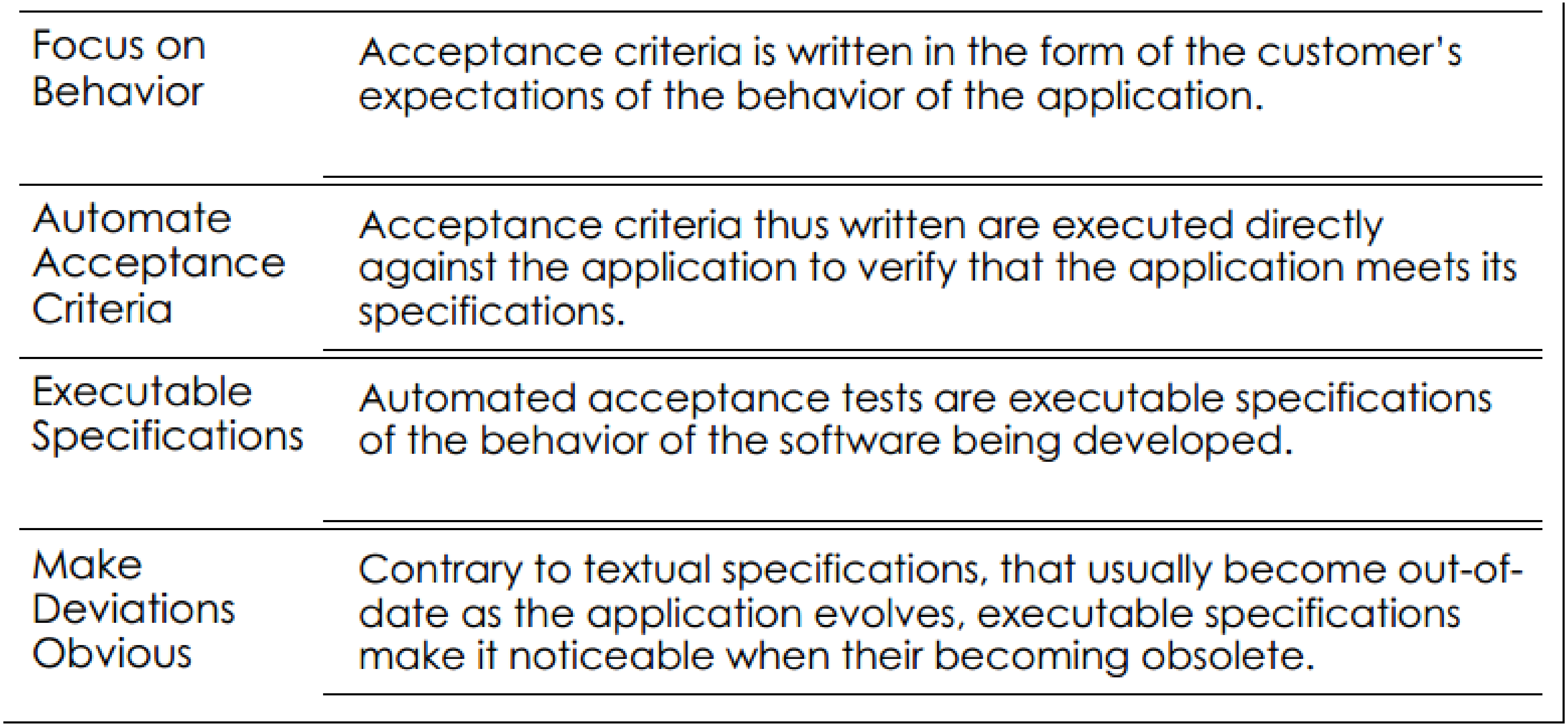
  Given a valid date range
  When a search is requested
  Then a list of available rooms in hotel chain is provided

…

- ❑ reject Invalid date ranges
- ❑ can also restrict search by hotel or room type

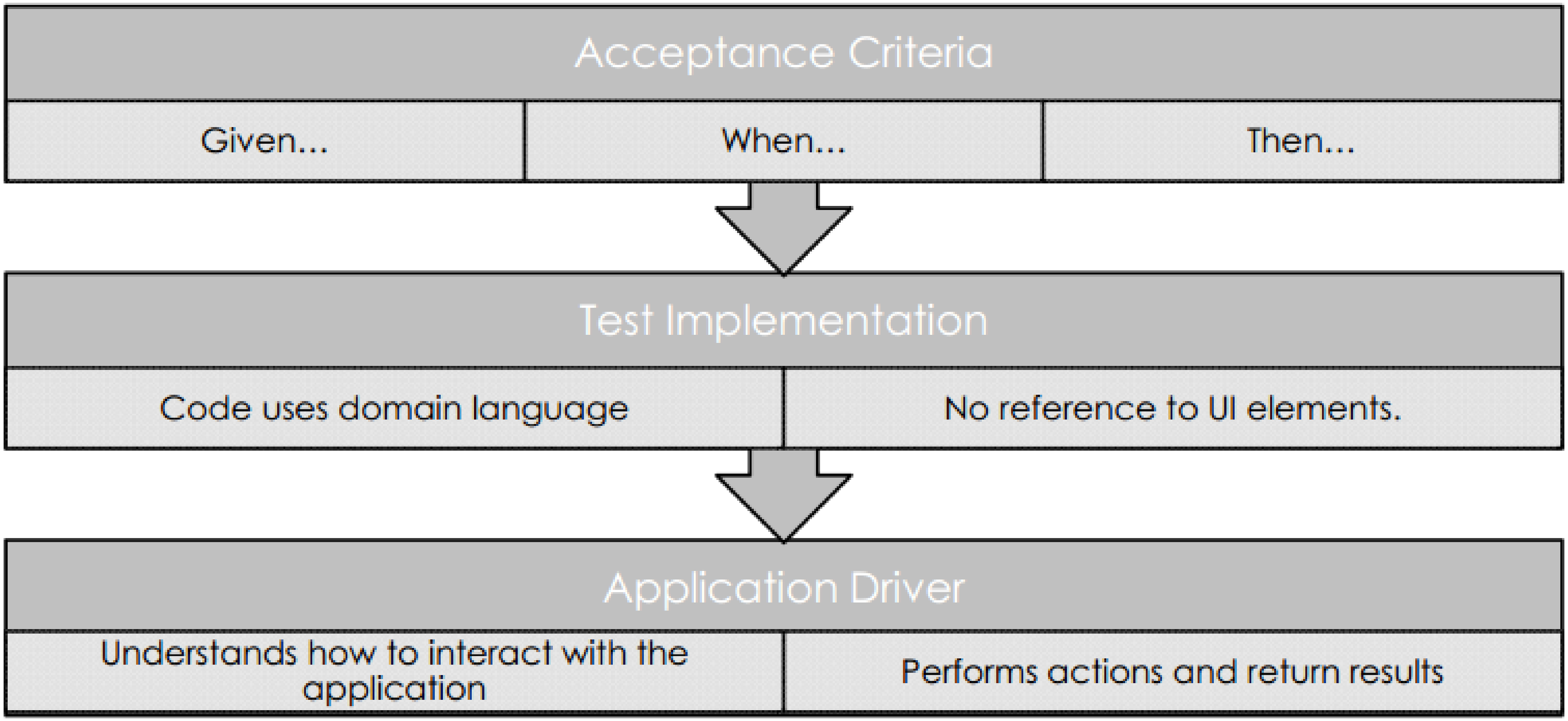
…

Behavior Driven Development



Focus on Behavior	Acceptance criteria is written in the form of the customer's expectations of the behavior of the application.
Automate Acceptance Criteria	Acceptance criteria thus written are executed directly against the application to verify that the application meets its specifications.
Executable Specifications	Automated acceptance tests are executable specifications of the behavior of the software being developed.
Make Deviations Obvious	Contrary to textual specifications, that usually become out-of-date as the application evolves, executable specifications make it noticeable when their becoming obsolete.

Components of Acceptance Test Automation



Acceptance Criteria

Given... | When... | Then...

Test Implementation

Code uses domain language | No reference to UI elements.

Application Driver

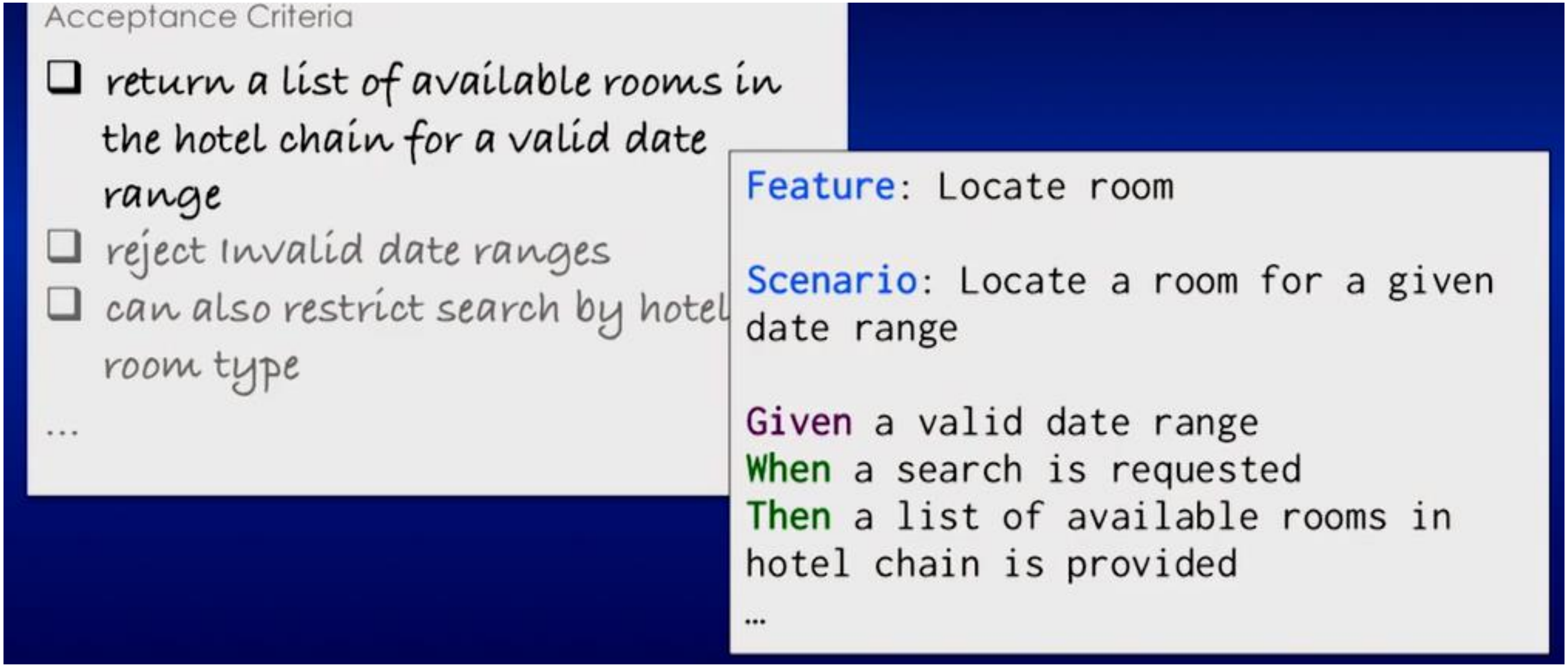
Understands how to interact with the application | Performs actions and return results

Acceptance Test Frameworks

- FitNesse.org
  - http://www.fitnesse.org/
  - User guide: http://www.fitnesse.org/FitNesse.UserGuide
  - FIT: framework for integration testing
    - http://www.fitnesse.org/FitNesse.UserGuide.WritingAcceptanceTests.FitFramework
  - Framework for table-driven acceptance testing

- Cucumber.io
  - Given-when-then format
- Specflow.org
  - Cucumber for .Net

Acceptance Criteria as Executable Specifications



Acceptance Criteria

- ☐ return a list of available rooms in the hotel chain for a valid date range
- ☐ reject invalid date ranges
- ☐ can also restrict search by hotel room type

...

```
Feature: Locate room

Scenario: Locate a room for a given
date range

Given a valid date range
When a search is requested
Then a list of available rooms in
hotel chain is provided
...
```

Cucumber Example

Feature Make a reservation

Feature identified by short phrase

As a reservation maker
I want to reserve a room at a hotel
In order to ensure a place to stay

Feature defined by user story

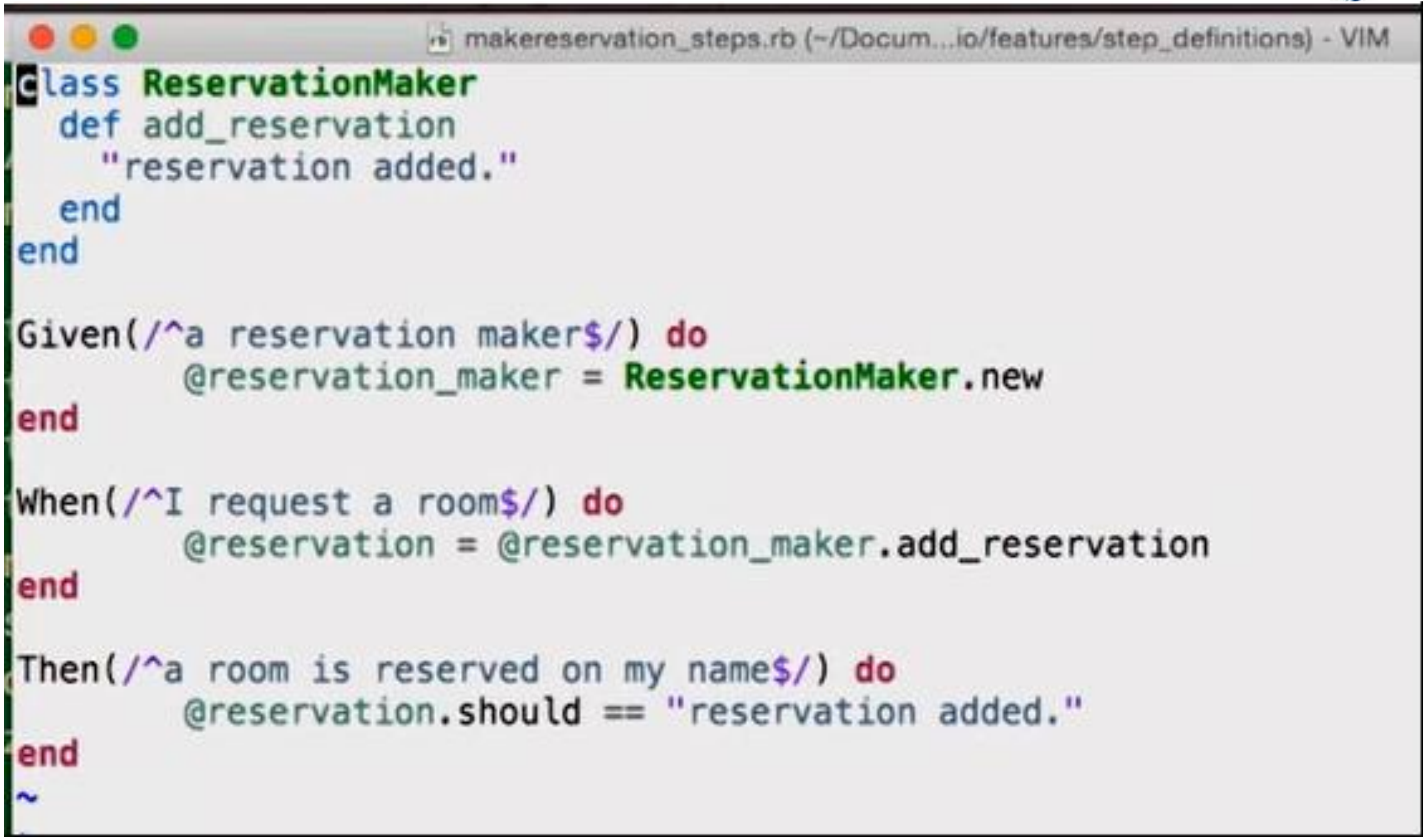
Scenario Room is available

Scenario: one particular way a user story could play out;
Corresponds to acceptance criterion

Given a reservation maker
When I request a room
Then a room is requested in my name

Scenario definition:
Given: test fixture
When: SUT execution
Then: Outcome verification

Cucumber Example (2)



makereservation_steps.rb (~/Docum...io/features/step_definitions) - VIM

```
class ReservationMaker
  def add_reservation
    "reservation added."
  end
end

Given(/^a reservation maker$/) do
        @reservation_maker = ReservationMaker.new
end

When(/^I request a room$/) do
        @reservation = @reservation_maker.add_reservation
end

Then(/^a room is reserved on my name$/) do
        @reservation.should == "reservation added."
end
~
```