20+ Minutes

JUNIT

JUnit

- Kent Beck and Erich Gamma (of Design Patterns fame) developed a unit testing framework for Java programs called **Junit**.

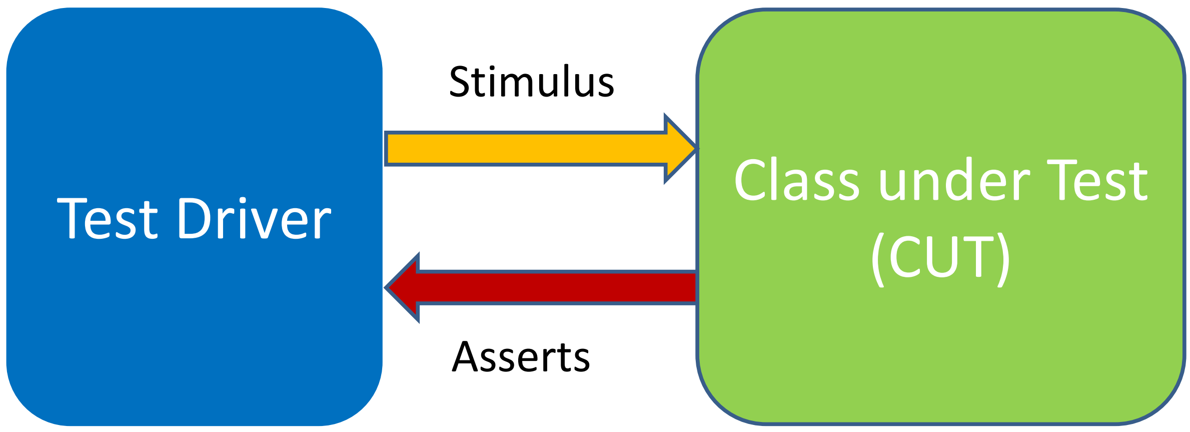
  http://www.junit.org

- JUnit 4.0 introduced annotations in the **org.junit** package for marking test code.
  - **@Test**, **@Before**, **@After**, **@BeforeClass**, **@AfterClass**, **@Ignore**, **@Test** etc.

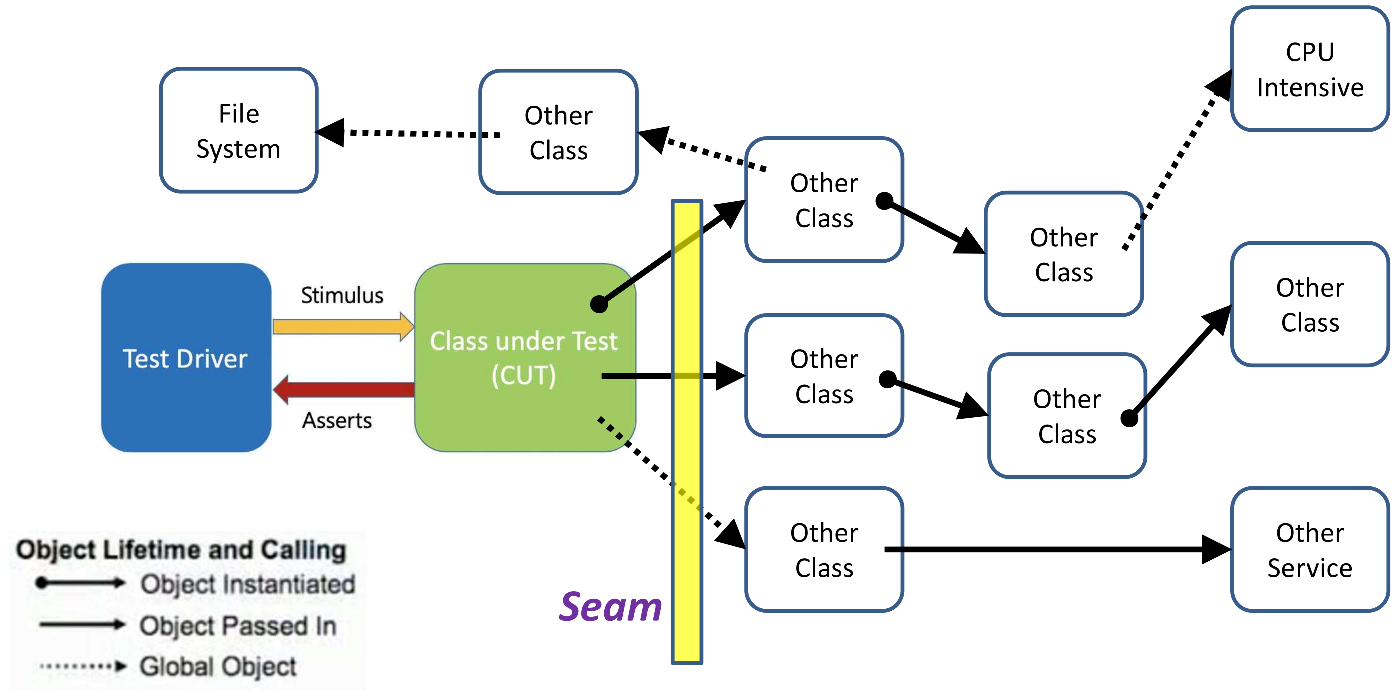
What is a unit?

- "The smallest component that it makes sense to test"
- Unit for testing depends on individual programmers or teams
- Generally, a unit means
  - class or an interface
  - a single method or function.

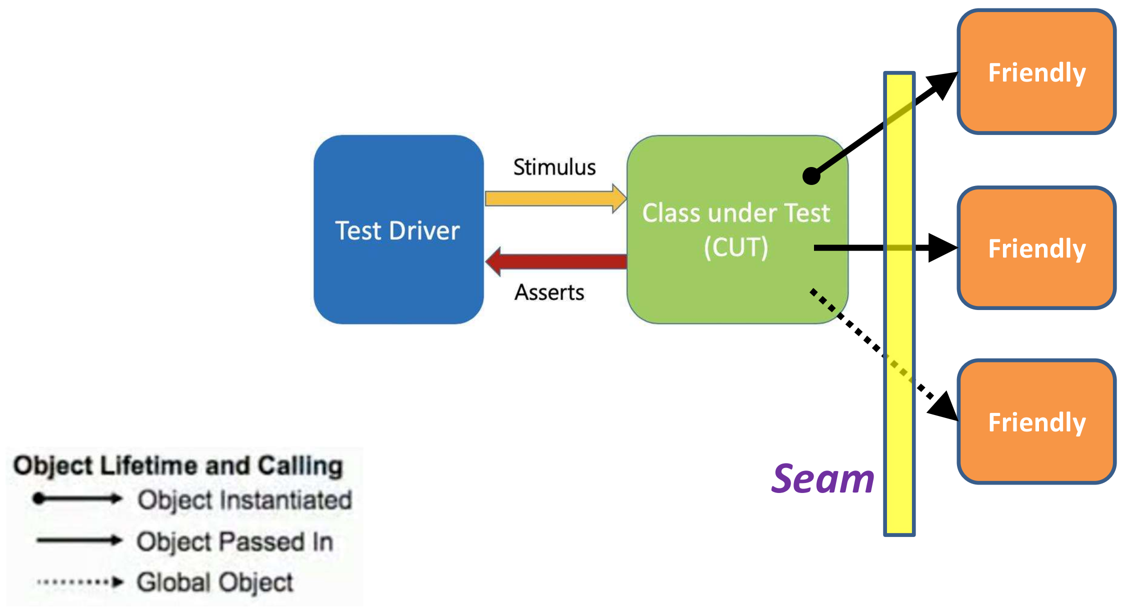
Unit Testing a Class



Unit Testing a Class



Unit Testing a Class



What is unit testing?

- Unit testing is a method that
  - instantiates a small part of our code (i.e., unit of work) and
  - verifies its behavior
  - ***independently*** from any other parts (Unit, Code etc.) of the project.
- External dependencies are managed by **Test Doubles** (Dummies/Fakes/Mocks/Stubs/Spies)

Terminology

- A **unit test** is (normally) a test of a *single* class
- A **test case** tests the response of a single method to a particular set of inputs.
- A **test fixture** is a fixed state of a set of objects used as a baseline for running tests.
  - The purpose is to ensure that there is a well known and fixed environment in which tests are run so that results are repeatable.
- A **test suite** is a collection of test cases.

Structure of a JUnit test class

- To test a class named **Foo**
- Create a test class **FooTest**

```
import static org.junit.jupiter.api.Assertions.*;;
import org.junit.jupiter.api.Test;

class FooTest {

	@Test
	void test() {
		fail("Not yet implemented");
	}

}
```

Test Cases

- Methods annotated with **@Test** are considered to be test cases:
  - Their order of execution is not specified

```
@Test
void testadd() {...}

@Test
@DisplayName(“Test for toString”)
void testToString() {...}

@Disabled
void testAnother() {...}
```

Test Fixtures

- Methods annotated with **@BeforeEach** will execute before every test case.
- Methods annotated with **@AfterEach** will execute after every test case

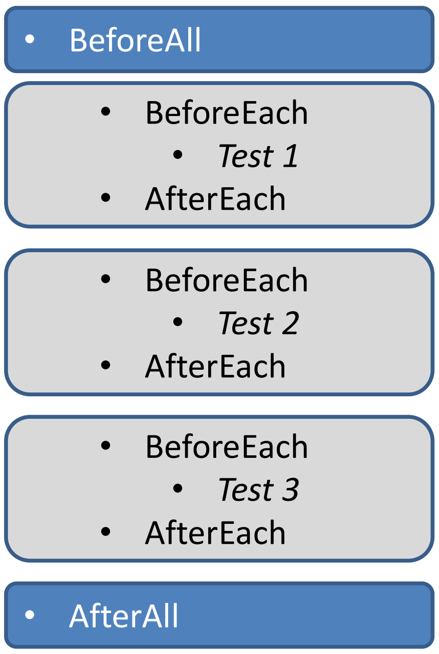
```
@BeforeEach
public void setUp() {...}

@AfterEach
public void tearDown() {...}
```

Class Test fixtures

- Methods annotated with **@BeforeAll** will execute once *before* all test cases.
- Methods annotated with **@AfterAll** will execute once *after* all test cases.
- These are useful if you need to allocate and release expensive resources once.

What JUnit does

- For *each* test case **t**:
  - JUnit executes all **@BeforeEach** methods
  - JUnit executes **t**
    - Any exceptions during its execution are logged
  - JUnit executes all **@AfterEach** methods
- Report for all test cases is presented

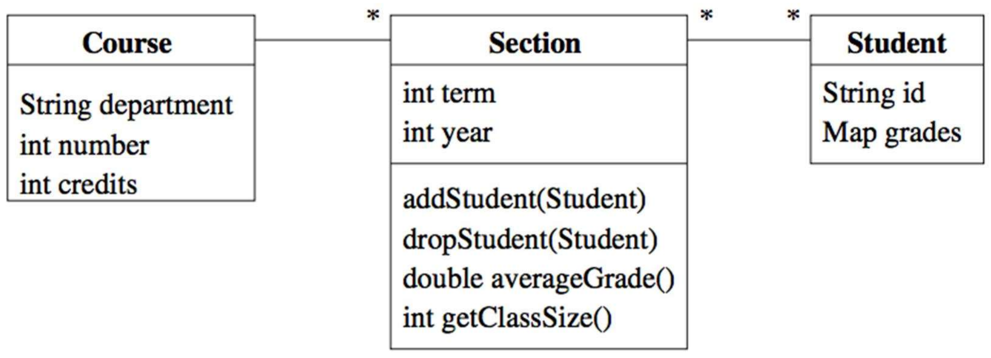


Within a test case

- Call the methods of the class being tested.
- Assert what the correct result should be with one of the provided **assert** methods.
- These steps can be repeated as many times as necessary.
- An **assert** method is a JUnit method that performs a test, and throws an **AssertionError** if the test fails.
  - JUnit catches these exceptions and shows you the results.

Example Classes

- To demonstrate writing unit tests, we are going to develop some classes for modeling **Student**s that are enrolled in a **Section** of a **Course**



Writing a simple test case

```
public class SectionTest {
  @Test
  public void testAddStudent() {

    Student student = new Student("123-45-6789");
    Course course = new Course("CS", 410, 4);
    Section section =
      new Section(course, Section.SPRING, 2001);

    section.addStudent(student);

    assertEquals(1, section.getClassSize());
  }
}
```

The left class tests that adding a **Student** increases the enrollment by one

Given

When

Then

The **assertEquals** method is imported from the **Assert** class. If its arguments are not equal, then the test fails.

Testing Error Conditions

```
@Test
void testDropStudentNotEnrolled() {
    Student student = new Student("123-45-6789");
    Course course = new Course("CS", 410, 4);
    Section section =
      new Section(course, Section.SPRING, 2001);
    assertThrows(IllegalArgumentException.class,
      () -> section.dropStudent(student));
}
```

- Making sure that your program fails in a well-understood fashion is very important.
- To test that the **dropStudent** method throws an **IllegalArgumentException**

The Assert class

- The **Assert** contains methods for validating that certain conditions are true.
  - **assertEquals**: Two entities (objects, ints, etc.) should be equal
    - (compares objects using **equals()**)
  - **assertNotNull**: A value should not be null
  - **assertSame**: Two object references should be the same
    - (compare objects using ==)
  - **assertTrue**: A boolean expression should be true
  - **assertFalse**: A boolean expression should be false
  - **fail**: The test should fail

The Assert class

- When an assertion evaluates to false, the test fails.
- Each **assert** method is overloaded to have a **String** message associated with it.

```
assertEquals(1, section.getClassSize());

assertEquals(
  1, section.getClassSize(), "Wrong number of students"
);
```

More readable assertions

JUnit provides some basic methods for validating the state of your tests (assertions), but the code and the failure messages can be hard to read

```
assertTrue(myString.contains("Hello"));
```

When the above fails, all you get is an “expected true, but got false” error message.

The **Hamcrest** assertion framework provides powerful “matchers” that provide readable assertion statements with detailed and specific failure messages:

http://hamcrest.org/JavaHamcrest

Hamcrest assertion statements

Hamcrest provides an **assertThat** method that asserts that some value “matches” a “matcher” .

Each “matcher” has a static factory method.

Matchers are composed to form complex assertions.

The matcher is syntactic sugar that aids readability.

```
import org.junit.Jupiter.api.Test;
import static org.hamcrest.Matchers.*;
import static org.hamcrest.MatcherAssert.assertThat;
class HamcrestMatchersTest {

  @Test
  void isEqualTo() {
    Integer int1 = new Integer("123");
    Integer int2 = new Integer("123");

    assertThat(int1, is(equalTo(int2)));
  }
```

Examples of Hamcrest assertions

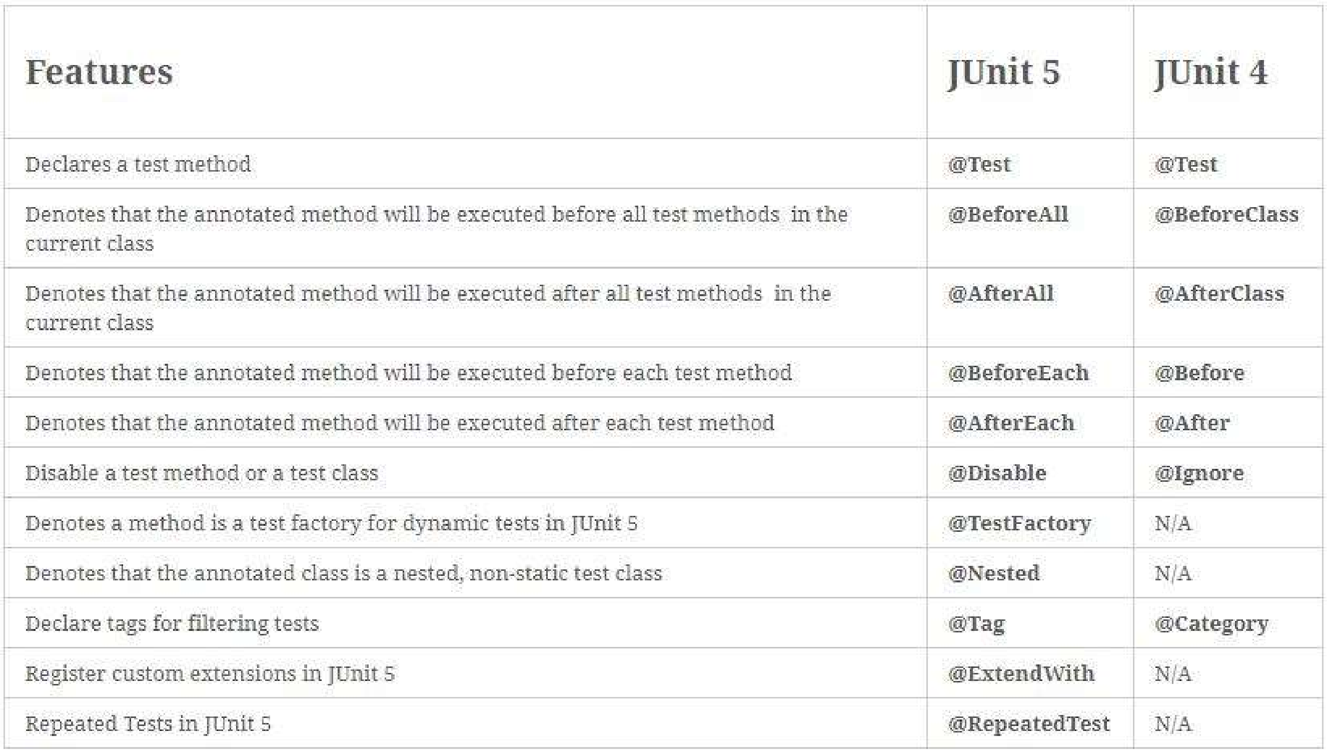
```
  @Test
  void isNullValue() {
    assertThat(null, is(nullValue()));
  }

  @Test
  void isSameInstance() {
    Object o = new Object();
    assertThat(o, is(sameInstance(o)));
  }

  @Test
  public strings() {
  String s = "Hamcrest is awesome";

    assertThat(s, startsWith("Hamcrest"));
    assertThat(s, endsWith("awesome"));
    assertThat(s, containsString("is"));
    assertThat(s, is(not(isEmptyString())));
    assertThat(s, is(equalToIgnoringCase("HAMCREST IS AWESOME")));
  }
```

Alternative to Hamcrest framework
https://assertj.github.io/doc/

AssertJ

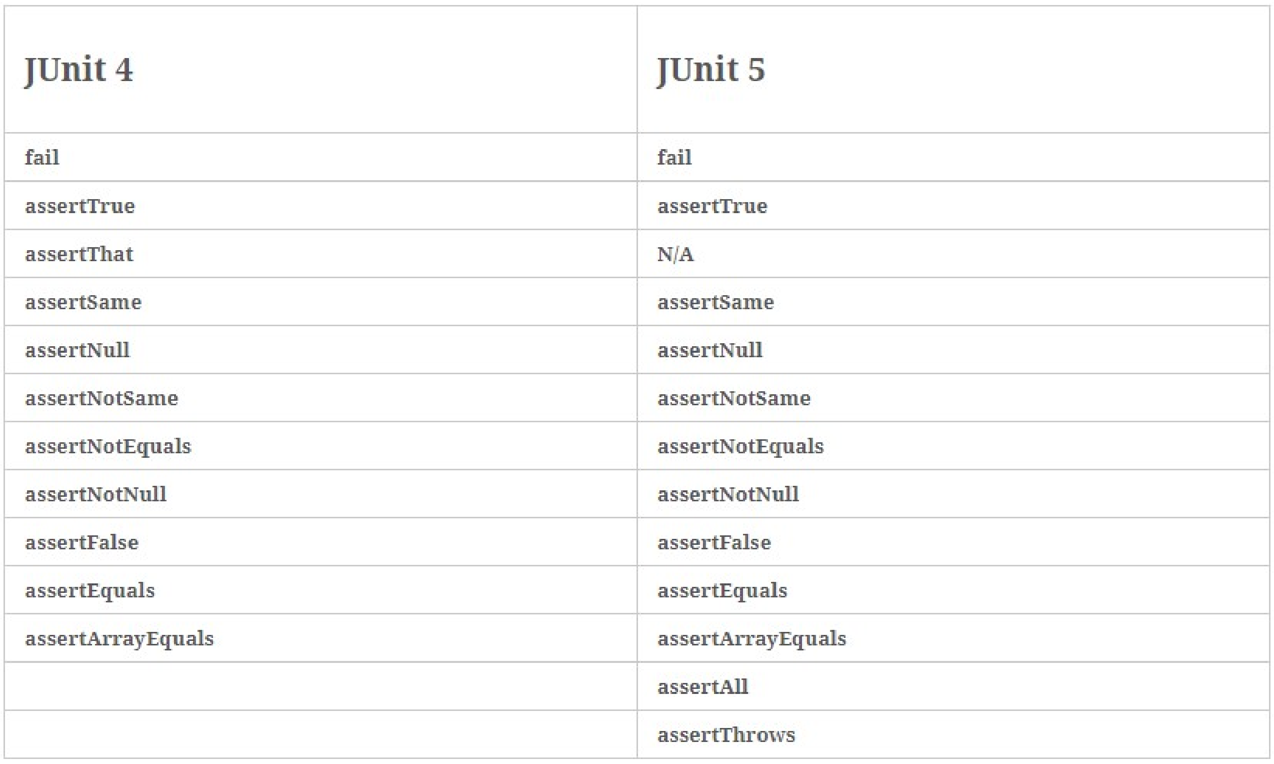
Fluent assertions for java

Annotations



Features	JUnit 5	JUnit 4
Declares a test method	@Test	@Test
Denotes that the annotated method will be executed before all test methods in the current class	@BeforeAll	@BeforeClass
Denotes that the annotated method will be executed after all test methods in the current class	@AfterAll	@AfterClass
Denotes that the annotated method will be executed before each test method	@BeforeEach	@Before
Denotes that the annotated method will be executed after each test method	@AfterEach	@After
Disable a test method or a test class	@Disable	@Ignore
Denotes a method is a test factory for dynamic tests in JUnit 5	@TestFactory	N/A
Denotes that the annotated class is a nested, non-static test class	@Nested	N/A
Declare tags for filtering tests	@Tag	@Category
Register custom extensions in JUnit 5	@ExtendWith	N/A
Repeated Tests in JUnit 5	@RepeatedTest	N/A

Assertions



JUnit 4	JUnit 5
fail	fail
assertTrue	assertTrue
assertThat	N/A
assertSame	assertSame
assertNull	assertNull
assertNotSame	assertNotSame
assertNotEquals	assertNotEquals
assertNotNull	assertNotNull
assertFalse	assertFalse
assertEquals	assertEquals
assertArrayEquals	assertArrayEquals
	assertAll
	assertThrows

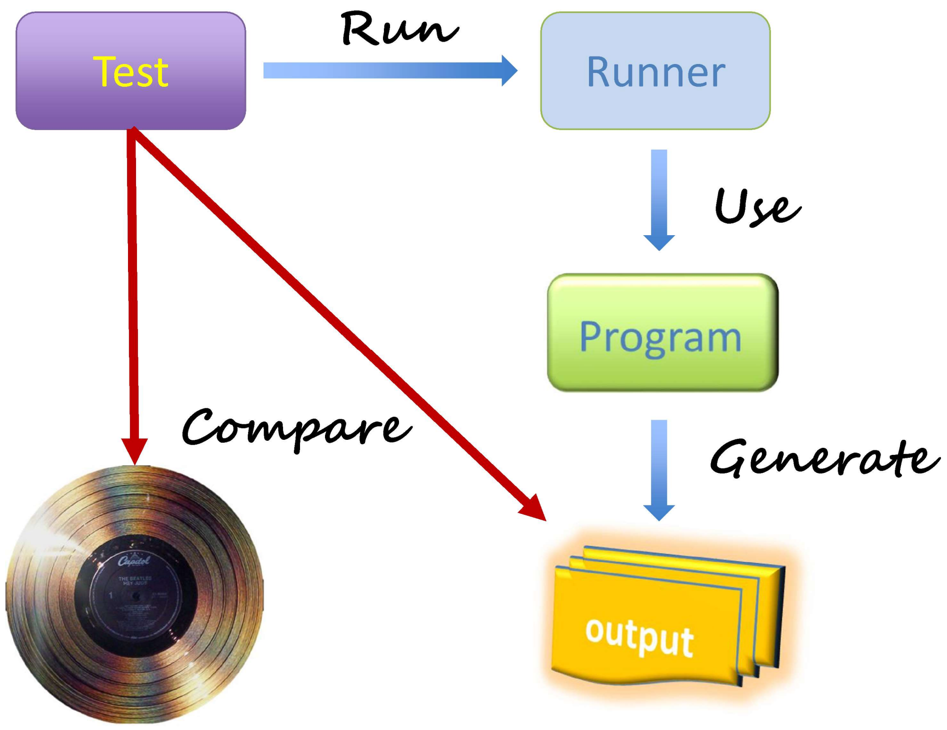
Golden Master Testing

- Golden master testing refers to capturing the result of a process, and then comparing future runs against the saved “gold master” to discover unexpected changes.
- Golden master testing is common when working with legacy code.
- Rather than trying to specify all of the logical paths through an untested module, you can feed it a varied set of inputs and turn the outputs into automatically verifying tests.
- There’s no guarantee the outputs are correct in this case, but at least you can be sure they don’t change.

How to implement golden master testing

1. Choose (or randomly generate, using a known seed) a set of inputs for your module or program.
2. Run the inputs through the system, persisting the output.
3. When testing a change, run the same inputs through the new version of the system and flag any output variation.
4. For each variation, have a human determine whether or not the change is expected and desirable. If it is, update the persisted gold master records.

Golden Master Testing Steps


Test
Run
Runner
Use
Program
Generate
Compare
output