MODULE ON

Automated state-based testing

RISE Research Institutes of Sweden

Building blocks the capsule

Introduction

Unit tests with JUNIT

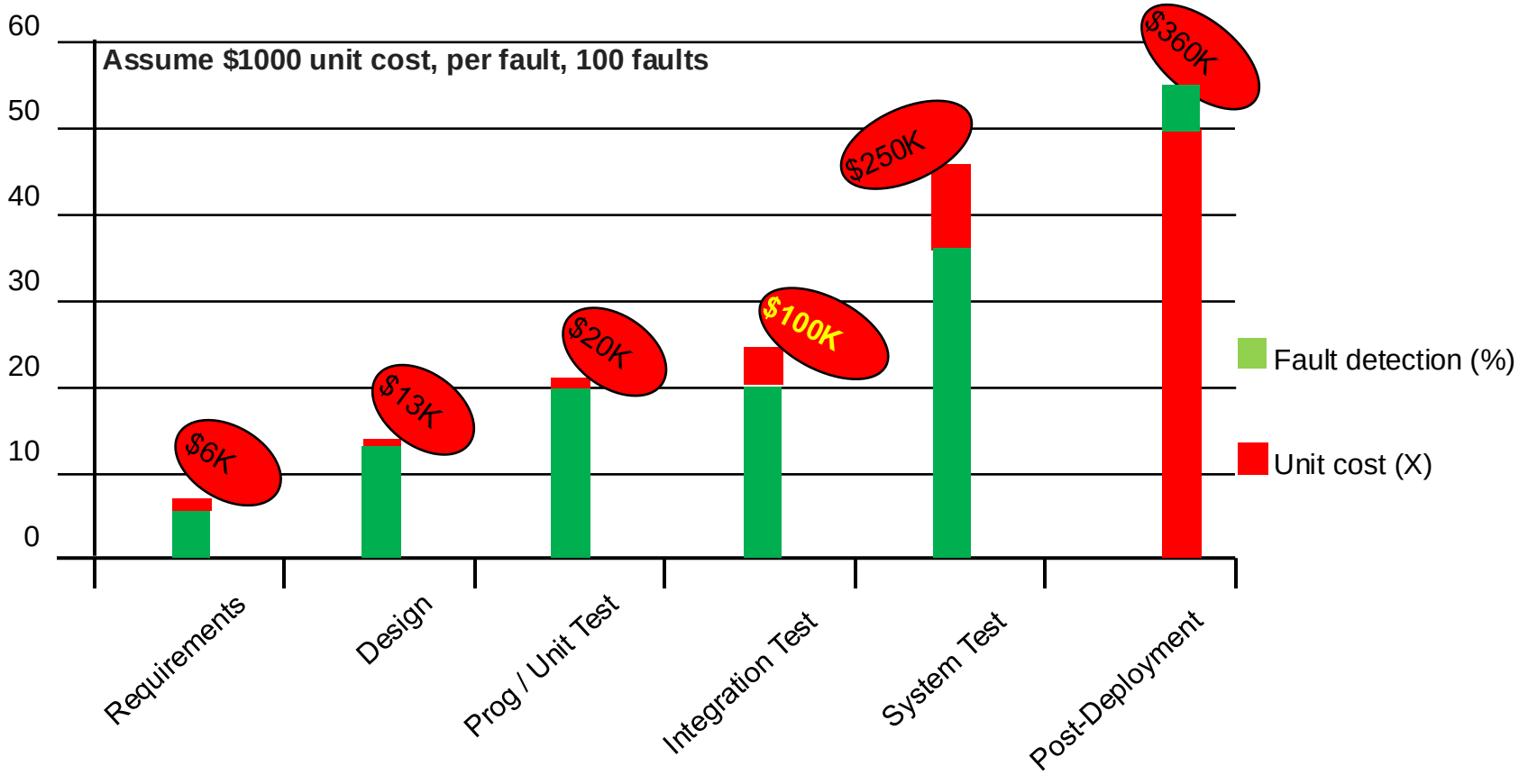
Activity 1: Test ThreePlayerGame

State Models, code instrumentation and test generation

Activity 2: Review generated tests

Test Evaluation

Software Testing – Introduction (Motivation)

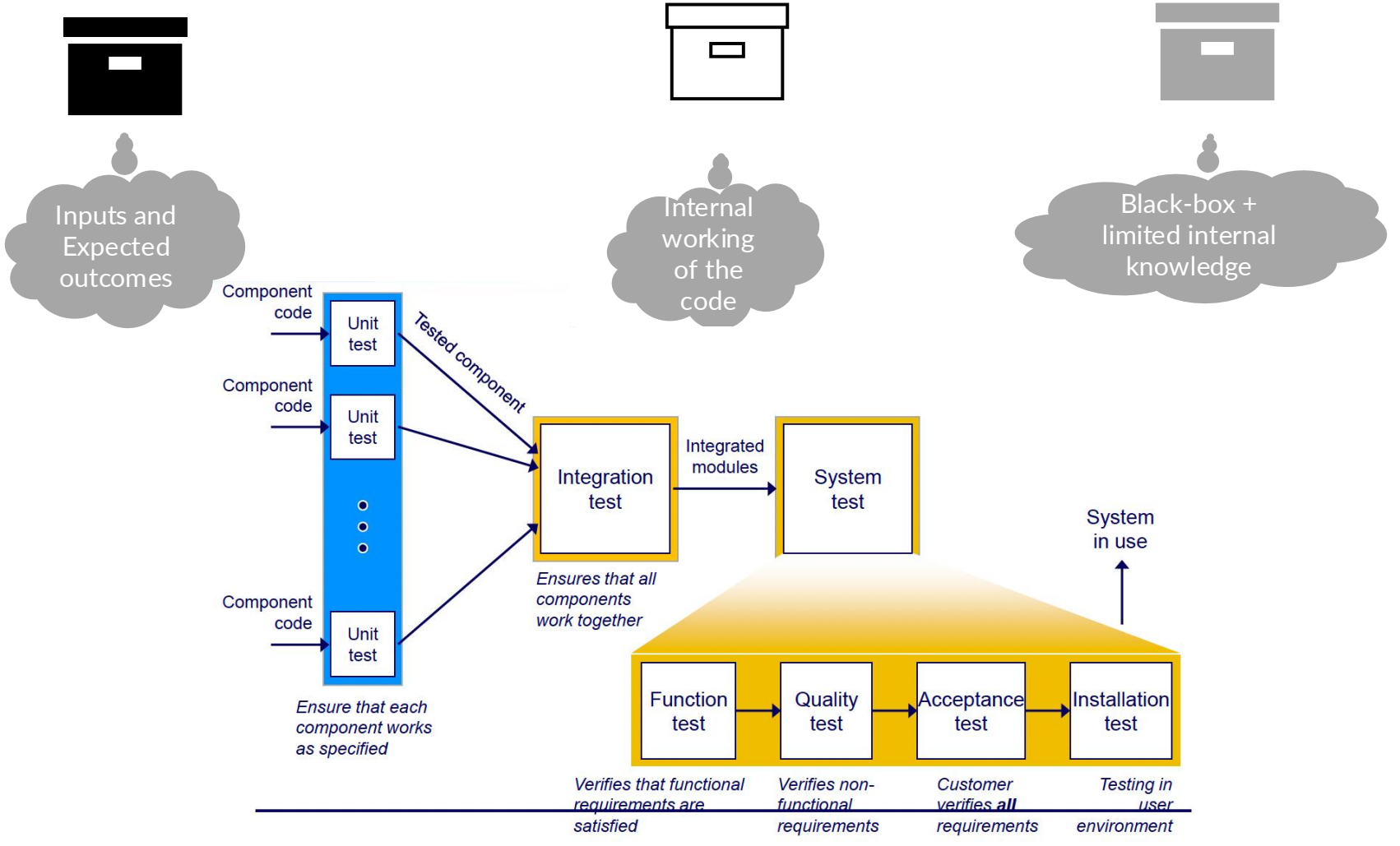


Software Engineering Institute; Carnegie Mellon University; Handbook CMU/SEI-96-HB-002

Software Testing – Introduction

"Testing is the process of establishing confidence that a program or system does what it is supposed to" - Hetzel, 1973

Nature of test design approaches



Example test

Arrange	Act	Assert

Scenario	Test name	Pre-conditions	Steps	Data	Expected Results	Actual Results	Pass/Fail
Check sum	Check sum for 2 positive numbers	N/A	- Initiate calculator – pass two numbers to sum()	2 and 2	4	4.0	Fail

```
public class Calculator {
  public float sum(float a, float b)
    return a+b;
}
```

Example test in code

```
import SUT.Calculator;

public class UnitTests {

        Calculator sut;
        //Check sum for 2 positive numbers
        public boolean checkSum() {
                sut= new Calculator();
                float r=sut.sum(2,2);
                if(r==4)
                        return true;
                return false;
        }
}
```

Arrange

Act

Assert

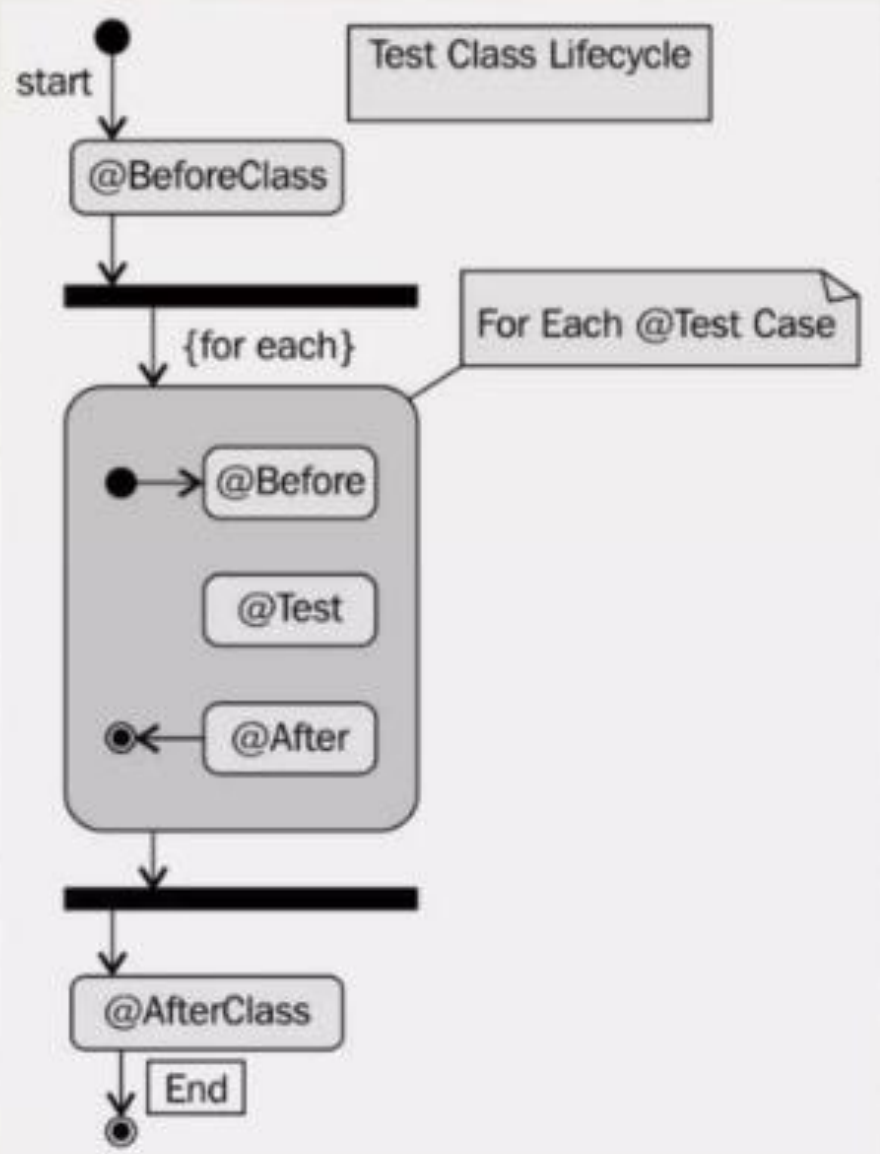
JUnit basics

Background on xUnit

- **First xUnit framework developed:** for Smalltalk in in 90s by Kent Beck
- **Why?:** To make it easy to arrange, act and assert. With additional perks on results visualization and repeatable cross-platform tests.
- **Junit:**
  - **Annotations and assertion for better test management:** @Test, @Test(timeout=1000), @Before, assertTrue()…
  - **Test Runner, and Test Suite** to manage test dependency and group tests
  - **Better visualization** (coverage and Junit test Results)

RI.
SE

Junit annotations flow



RI.
SE

Junit Assertions

Boolean

- assertEquals(bool expected, bool actual)
- assertTrue/False(bool condition)
- assertNull/assertNotNull(Object obj)

Reference

- assertSame/assertNotSame(ref1, ref2)

Array

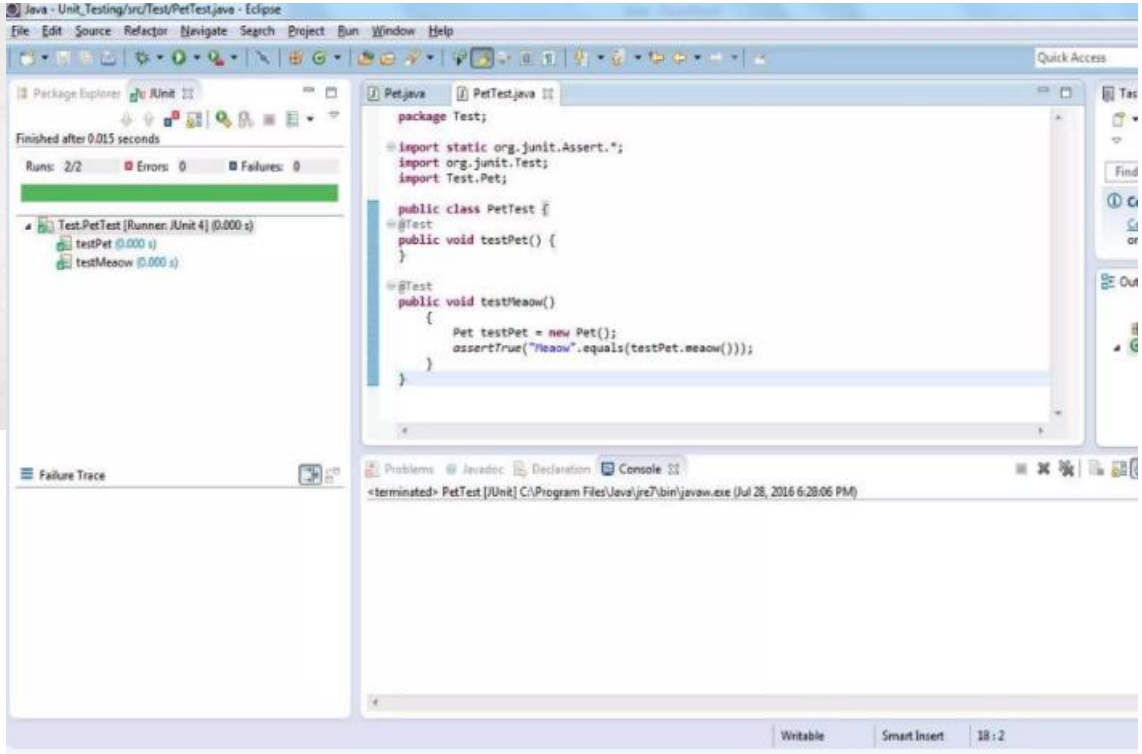
- assertArrayEquals(array1, array2)

RI.
SE

Example @Test and Assert

```java
package Test;

public class Pet {
public Pet() {}

public String meaow() {
return "Meaow";
}
}
```

```java
package Test;

import static org.junit.Assert.*;
import org.junit.Test;
import Test.Pet;

public class PetTest {
@Test
public void testPet() {
}

@Test
public void testMeaow()
{
Pet testPet = new Pet();
assertTrue("Meaow".equals(testPet.meaow()));
}
}
```



Example Test suite

```
package Test;

import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import org.junit.runners.Suite.SuiteClasses;

@RunWith(Suite.class)
@SuiteClasses({ PetTest.class, TestCase.class, TestCase1.class,
TestClassMain.class, TestSuite.class, UnitTest.class })
public class AllTests {

}
```

Testing Exceptions

Java - Unit_Testing/src/TestSamples/ExceptionCase.java - Eclipse

File Edit Source Refactor Navigate Search Project Run Window Help

Package Explorer | JUnit

Finished after 0.01 seconds

Runs: 1/1 Errors: 0 Failures: 0

TestSamples.ExceptionCase [Runner: JUnit 4] (0.000 s)

outOfBounds (0.000 s)

ExceptionCase.java

```
package TestSamples;

import java.util.ArrayList;

import org.junit.Test;

public class ExceptionCase {
@Test(expected = IndexOutOfBoundsException.class)
public void outOfBounds() {
new ArrayList<Object>().get(1);
}
}
```

System under test (SUT)

```java
package niSUT;

public class niTwoPlayerGame {
        private int p2_points;
        private int p1_points;
        public int server; // server variable for P1, P2 and P3 as

        public niTwoPlayerGame() ...
        }
        public void p1_Start() ...
        }
        public void p1_WinsVolley() ...
        }
        private void p1_AddPoint() ...
        }
        public boolean p1_IsWinner() ...
        }
        public boolean p1_IsServer() ...
        }
        public int p1_Score() ...
        }
        public void p2_Start() ...
        }
        public void p2_WinsVolley() ...
        }
        private void p2_AddPoint() ...
        }
        public boolean p2_IsWinner() ...
        }
        public boolean p2_IsServer() ...
        }
        public int p2_Score() ...
        }

        public void simulateVolley()
        {
                System.out.println("simulateVolley() Called");
        }
}
```

SUT: 3 Player Volley Game Back-end

- **Serving:** The game *starts with a serve by one of the players*. The *server* is determined by a random draw or agreement among the players.
- **Gameplay:** Players take turns on serving the ball, given that they *win a volley*.
- **Scoring:** A point is scored when a player wins the volley. The score is maintained for all of the three players involved. The first player to reach a predetermined number of points (21) *wins the game*.

RI.
SE

```java
        public void p1_Start()
        {
                /*P1 Serves first*/
                /* not implemented method simulateVolley() should be called here */
                server=1; // p1 is server

        }
        public void p1_WinsVolley()
        {
                /*P1 ends the volley*/
                /* not implemented method simulateVolley() should be called here */
                server=1; // p1 is server
                p1_AddPoint();

        }
        private void p1_AddPoint()
        {
                /*Adds 1 to the P1's score*/
                p1_points++;
        }
        public boolean p1_IsWinner()
        {
                /*True if P1's score is 21*/
                if(p1_points>20)
                {
                        return true;
                }
                return false;
        }
        public boolean p1_IsServer()
        {
                /*True if P1 is server*/
                if(server==1)
                        return true;
                return false;
        }
}
```

```java
package niSUT;

public class niThreePlayerGame extends niTwoPlayerGame {
        private int p3_points;
        public niThreePlayerGame()
        {
                /*Constructor*/
        }
        public void p3_Start()
        {
                /*P3 Serves First*/
                /* not implemented method simulateVolley() should be called here */
                server=3; // p3 is server
        }
        public void p3_WinsVolley()
        {
                /*P3 ends the volley*/
                /* not implemented method simulateVolley() should be called here */
                server=3; // p3 is server
                p3_AddPoint();
        }
        private void p3_AddPoint() ...
        }
        public boolean p3_IsWinner() ...
        }
        public boolean p3_IsServer() ...
        }
        public int p3_Score()
        {
                /*Returns P3's Score*/
                return p3_points;
        }
}
```

Example test

```
3   import static org.junit.Assert.*;
4
5   import org.junit.Test;
6
7   import niSUT.niThreePlayerGame;
8
9   public class UnitTests {
10
11          niThreePlayerGame sut;
12          //example test case that tests server tracking for server p1
13          @Test
14          public void testServer1() {
15                  sut= new niThreePlayerGame();
16                  sut.p1_Start();
17                  assertEquals(1,sut.server);
18          }
19  }
20
```

- **Serving:** The game *starts with a serve by one of the players*. The *server* is determined by a random draw or agreement among the players.

- Write and submit Junit4 unit tests for the ThreePlayerGame class and its inherited methods.

- **Serving:** The game *starts with a serve by one of the players*. The *server* is determined by a random draw or agreement among the players.
- **Gameplay:** Players take turns on serving the ball, given that they *win a volley*.
- **Scoring:** A point is scored when a player wins the volley. The score is maintained for all of the three players involved. The first player to reach a predetermined number of points (21) *wins the game*.

Activity 1

TEST CASE GENERATOR

SUT1: THREPLAYERGAME
SUT1: UNIT TESTS
SUT1: STATE MODEL
SUT1: INSTRUMENTED THREPLAYERGAME
SUT1: GENERATE TEST
SUT1: EVALUATE SUT

UnitTests.java

```
package niSUT.tests;

import static org.junit.Assert.*;

import org.junit.Test;

import niSUT.niThreePlayerGame;

public class UnitTests {

        niThreePlayerGame sut;
        //example test case that tests server tracking for server p1
        @Test(timeout = 1000)
        public void unitTest0() {
                sut= new niThreePlayerGame();
                sut.p1_Start();
                assertEquals(1,1);
        }
}
```

Console

```
JUnit version 4.13.2
.
Time: 0.009

OK (1 test)
```

Run your unit tests in the Code Playground

State-based testing (Motivation)

ARDUPILOT

egunak95 Aleksandr 1 Aug '22

Share

r participating in solving my problem. I thought that in the new firmware of the controller, the developers excluded DISARM in flight, I read about this on other forums. There is a way to mechanically lock the switch, for this you need to disassemble the transmitter and change the switch. For example, these are the switches:

aliexpress.ru

Тумблер с замком - купить недорого | AliExpress 2

Тумблер с замком купить по выгодной цене на АлиЭкспресс. ➤ Скидки, купоны, промокоды. Отзывы реальных покупателей. ✓ Мы ускорили доставку по РФ. Тумблер с замком - большой выбор на сайте и в новом приложении AliExpress Россия.

d the
hang

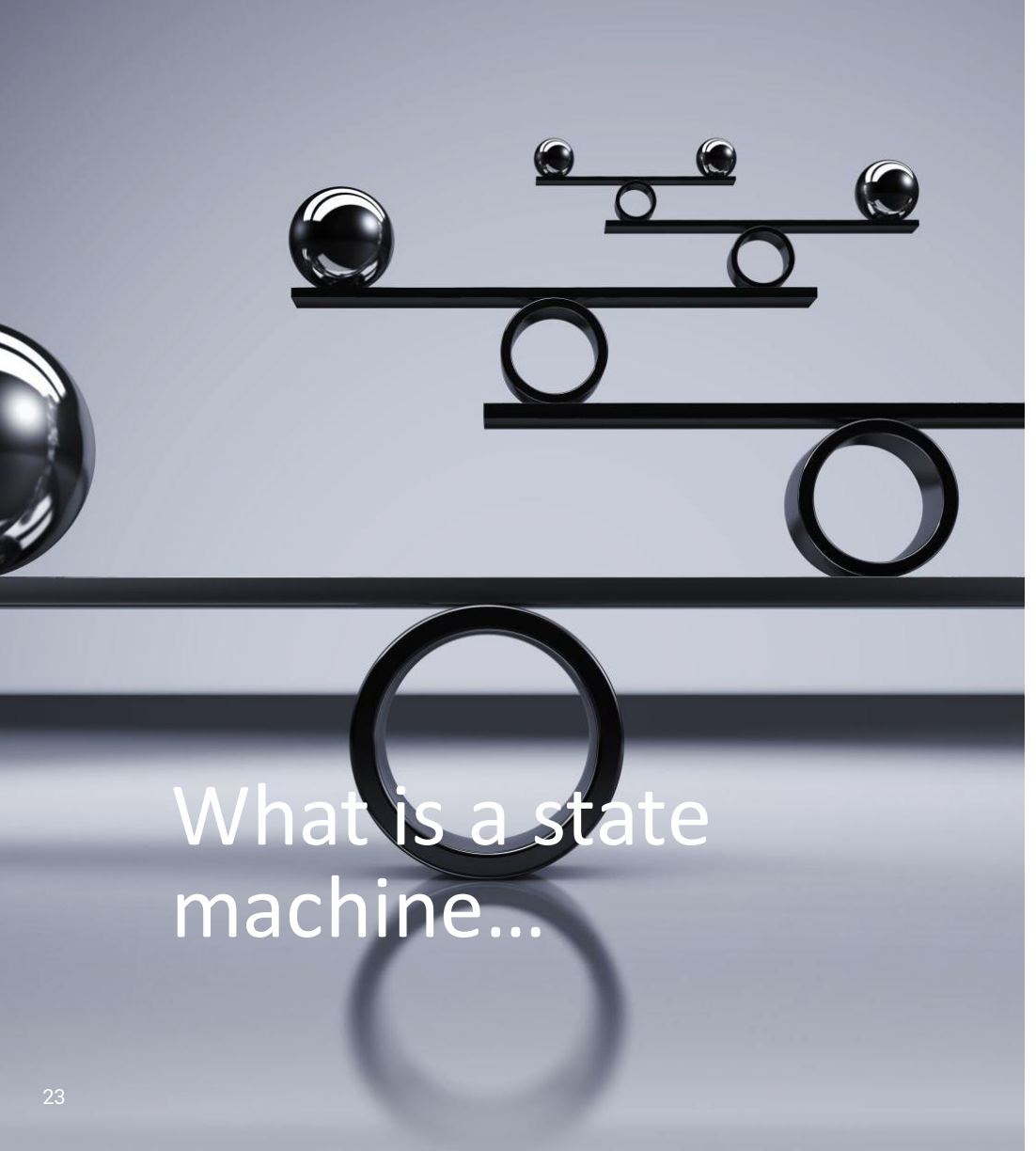
State-based modelling & testing

RI.
SE

Motivation

- We are interested in testing the behaviour of many different types of systems, including event-driven software systems
- Interaction with GUI systems can follow a large number of paths
- State machines can model event-driven behaviour
- If we can express the system under test as a state machine, we can generate test cases for its behaviour

RI.
SE



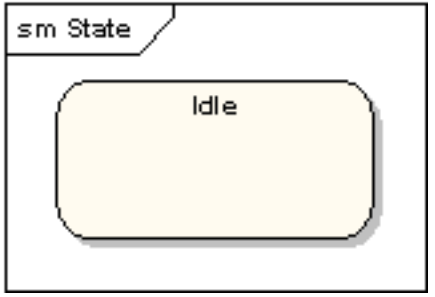
What is a state machine…

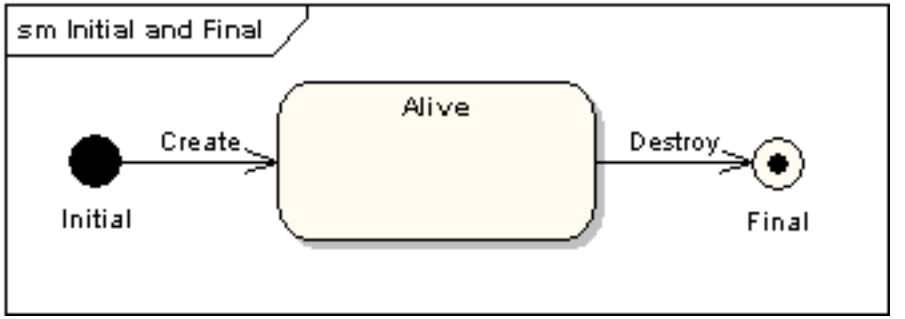
- A system whose output is determined by both current state and past input
- Previous inputs are represented in the current state
- State-based behaviour
  - **Identical inputs are not always accepted**
    - Depends upon the state
- **When accepted, they may produce different outputs**
  - Depends upon the state

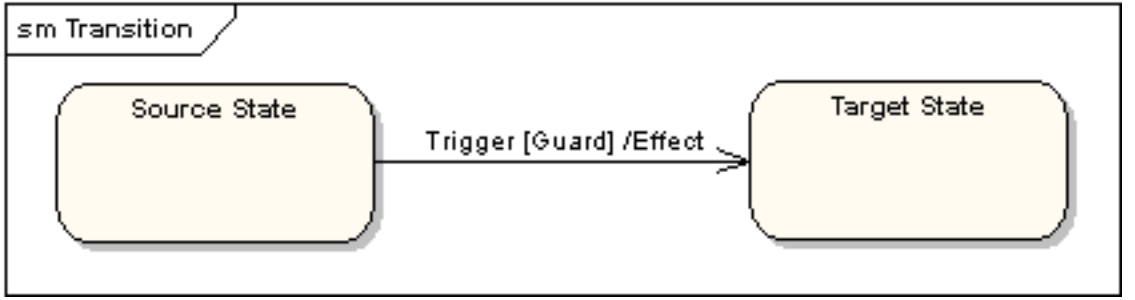
Building blocks of a state machine

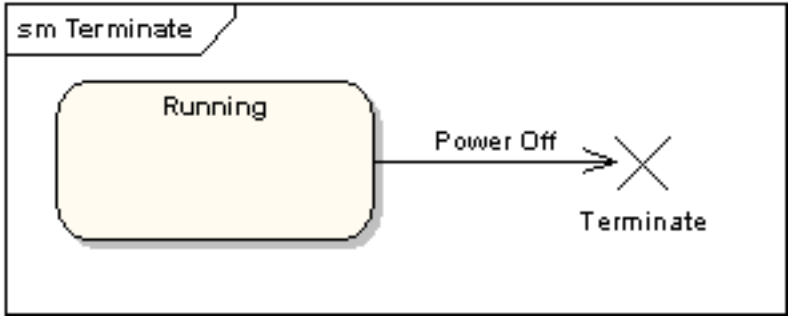
State	An abstraction that summarizes past inputs, and determines behaviour on subsequent inputs
Transition	An allowable two-state sequence. Caused by an event
Event	An input or a time interval
Action	The output that follows an event

State machines

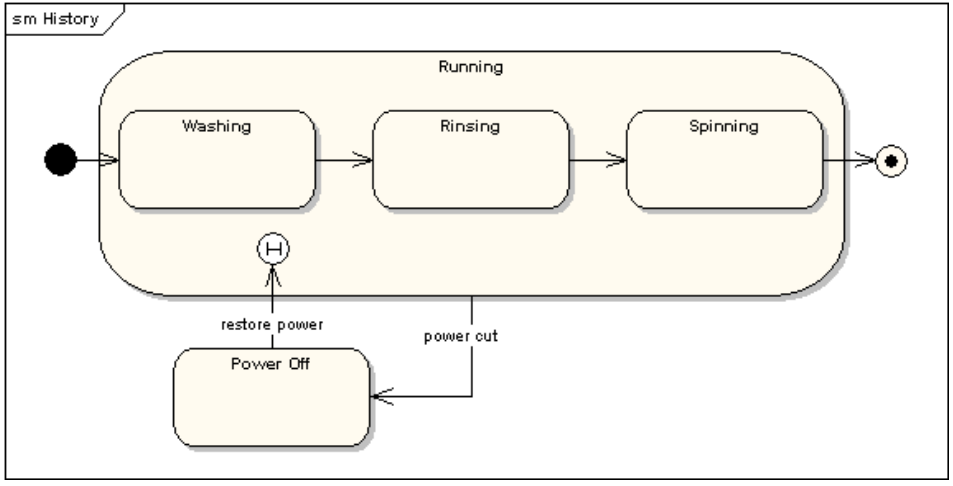


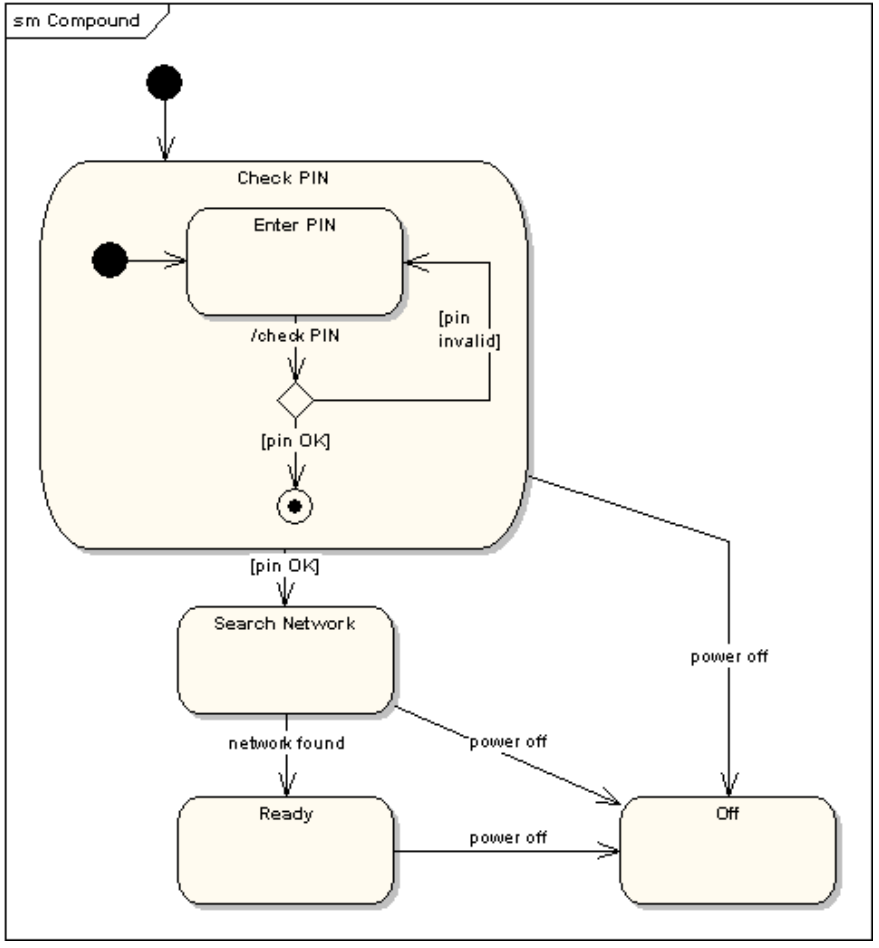






State machines



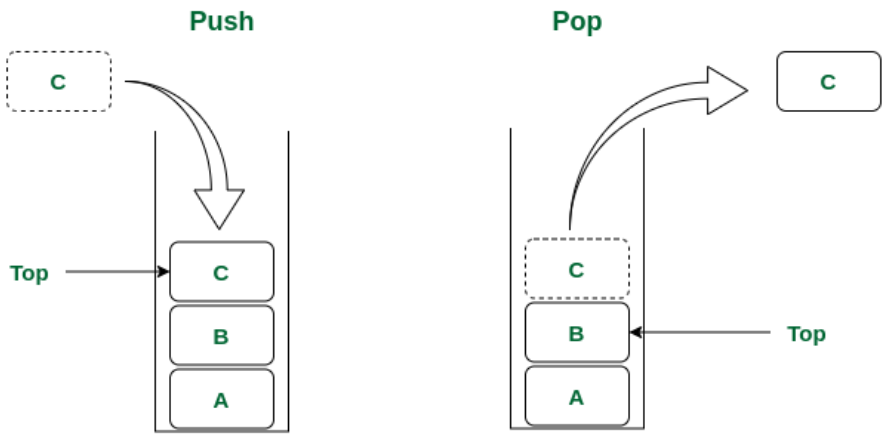


State machine – a behavioral model

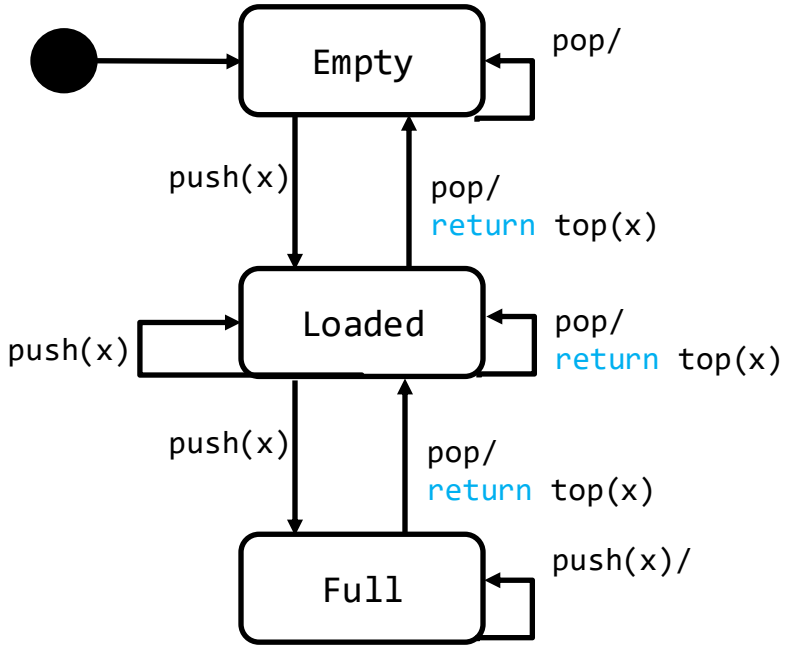
1. Begin in the **initial state**
2. Wait for an event
3. An event comes in
    1. If not accepted in the current state, **ignore**
    2. If accepted, a transition fires, output is produced (if any), the **resultant state** of the transition becomes the current state
4. Repeat from step 2 unless the current state is the **final state**

State Transition Diagram

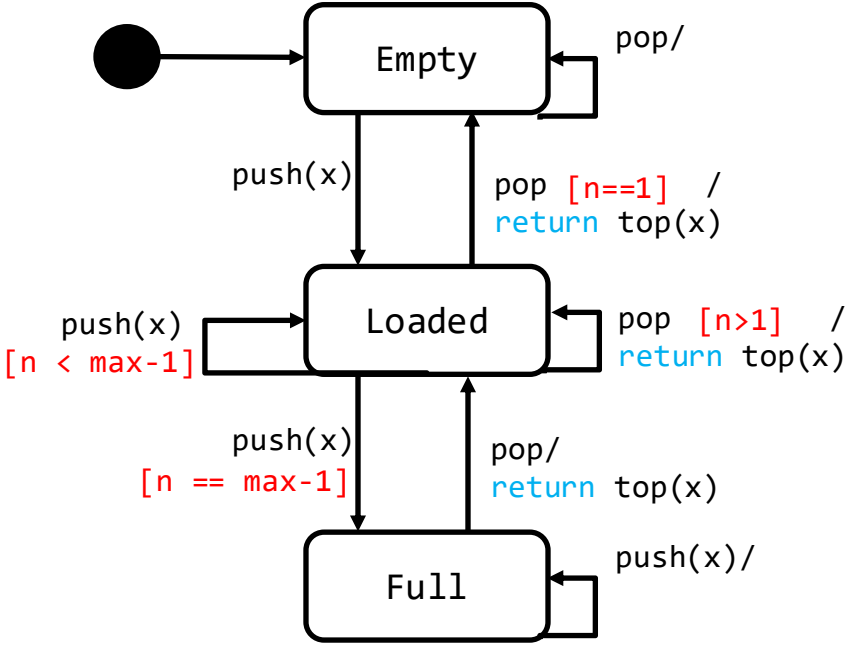
- This model is ambiguous, e.g. there are two possible reactions to push and pop in the Loaded state
- Guards can be added to transitions
- A guard is a predicate associated with the event
- A guarded transition cannot fire unless the guard predicate evaluates to true

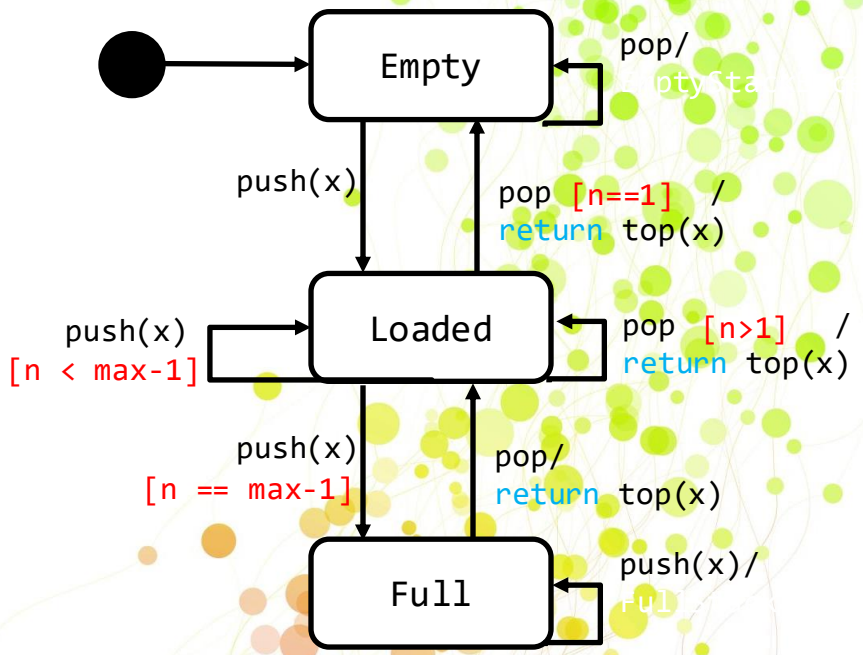


Stack Data Structure



State Transition Diagram with guards

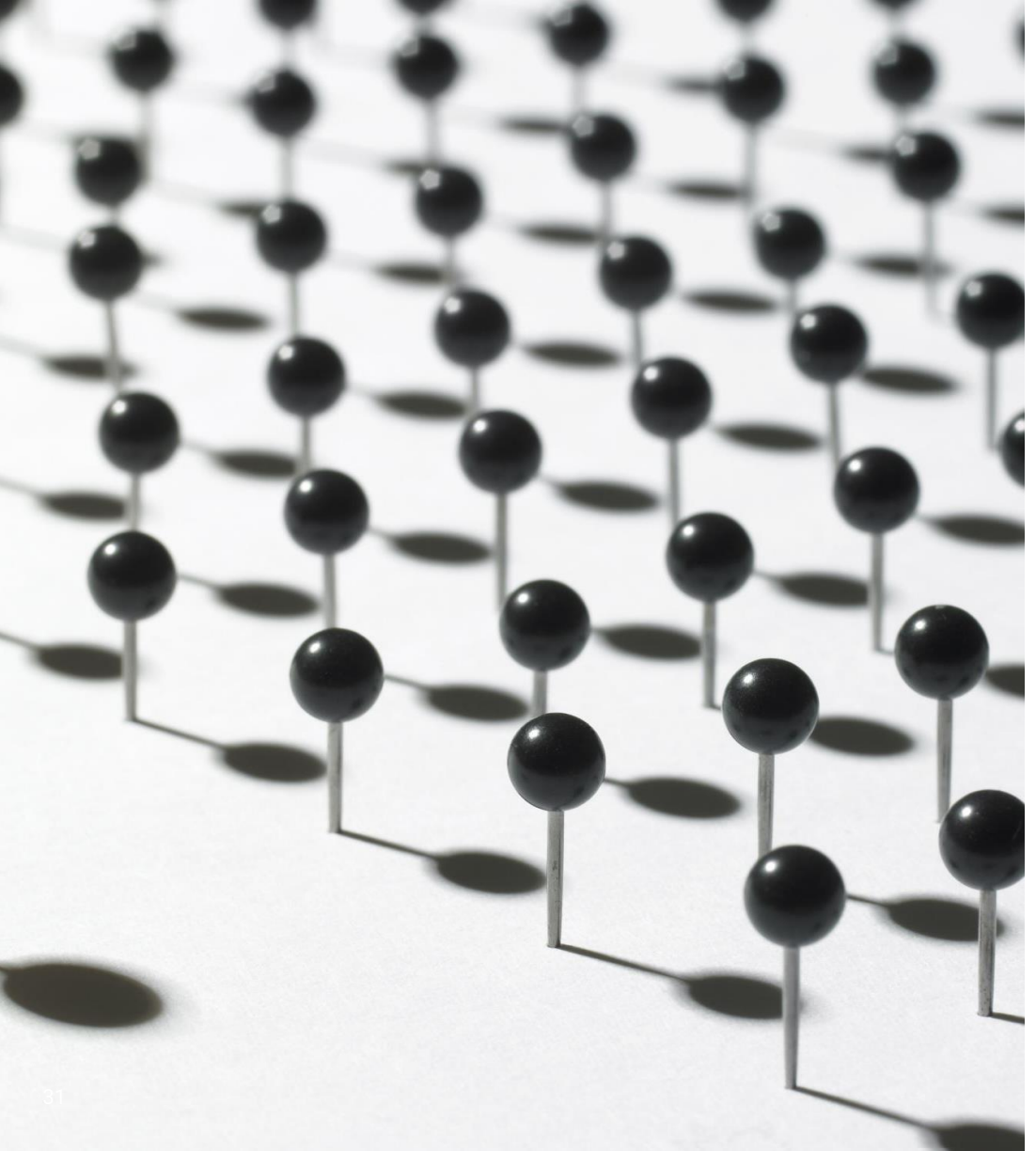




Coverage

- Exhaustive
- All Transitions/state
  - Every transition executed at least once
- N-wise Transition
  - All n-transition sequences
- All round trip paths
- All guards
  - Exercise guards in negation and in satisfaction
- Sneak paths (corrupt states)
  - Exercise all illegal paths

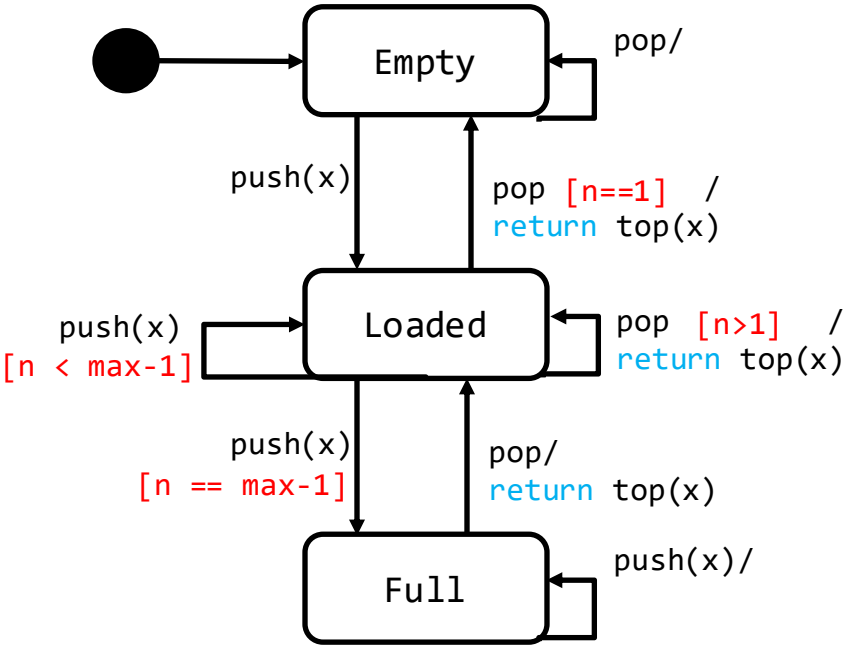
RI.
SE



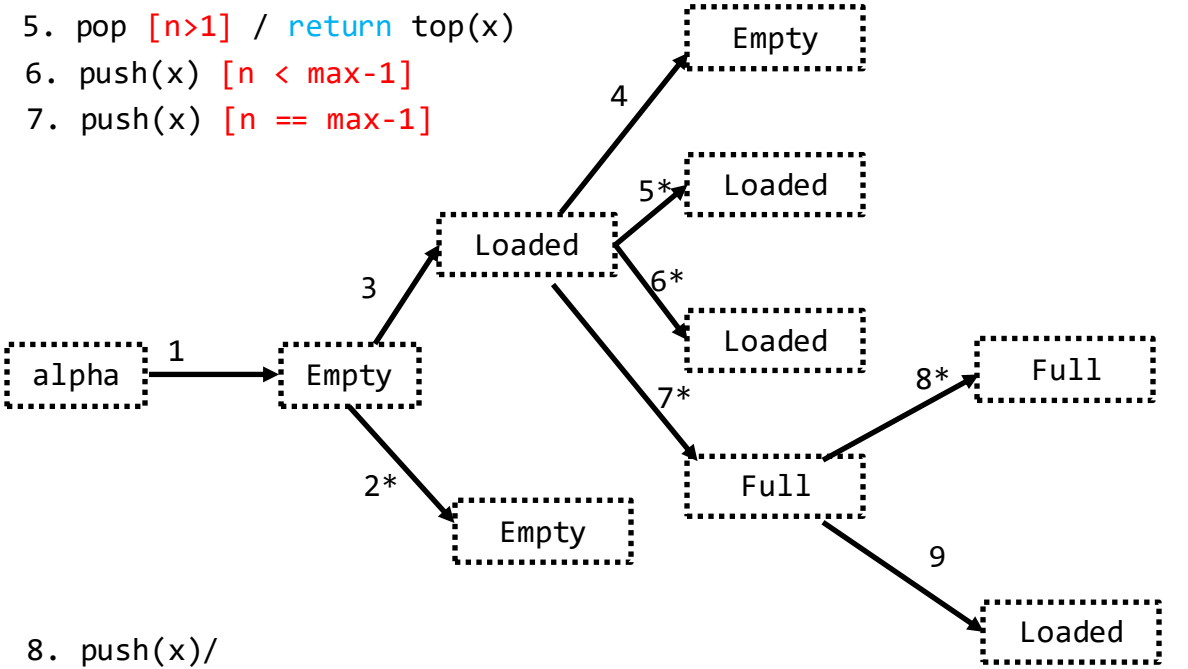
Coverage: N+ Strategy

- N+ Strategy
  - Start at α
  - Follow transition path
  - Stop if ω or visited
  - Three loop iterations
  - Assumes state observer
  - Try all sneak paths

N+ Strategy



1. ()
2. pop/
3. push(x)
4. pop [n==1] / return top(x)
5. pop [n>1] / return top(x)
6. push(x) [n < max-1]
7. push(x) [n == max-1]



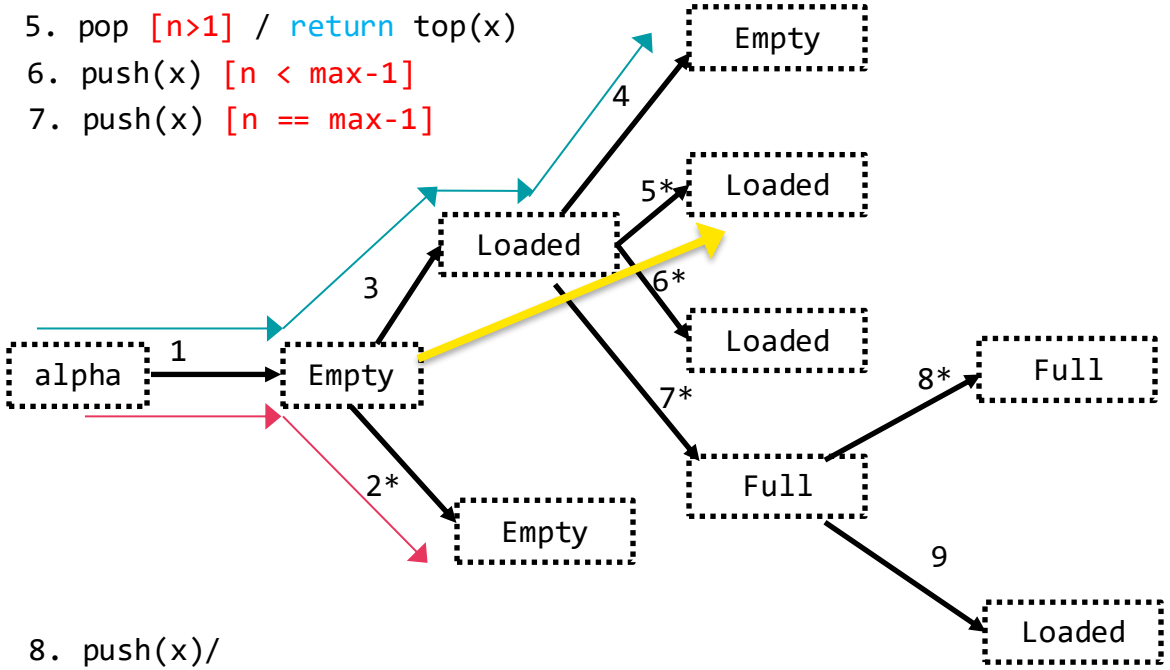
8. push(x)/
9. pop/return top(x)

N+ Conformance Tests

1. ()
Check we are in "Empty" State
2.pop/

Check we are in "Empty" State
~()
1. ()
Check we are in "Empty" State
3. push(x)
Check we are in "Loaded" State
4. pop [n==1] / return top(x)
Check we are in "Empty" State
3. push(x)
Check we are in "Loaded" State
GUARD: [n>1] ?
3. push(x)
5. pop [n>1] / return top(x)
Check we are in "Loaded" State
~()

1. ()
2.pop/
3. push(x)
4. pop [n==1] / return top(x)
5. pop [n>1] / return top(x)
6. push(x) [n < max-1]
7. push(x) [n == max-1]



8. push(x)/
9. pop/return top(x)

Manual review
(True to life)

N+ Conformance Test (in JUnit)

1. ()
Check we are in "Empty" State
3. push(x)
Check we are in "Loaded" State
4. pop [n==1] / return top(x)
Check we are in "Empty" State

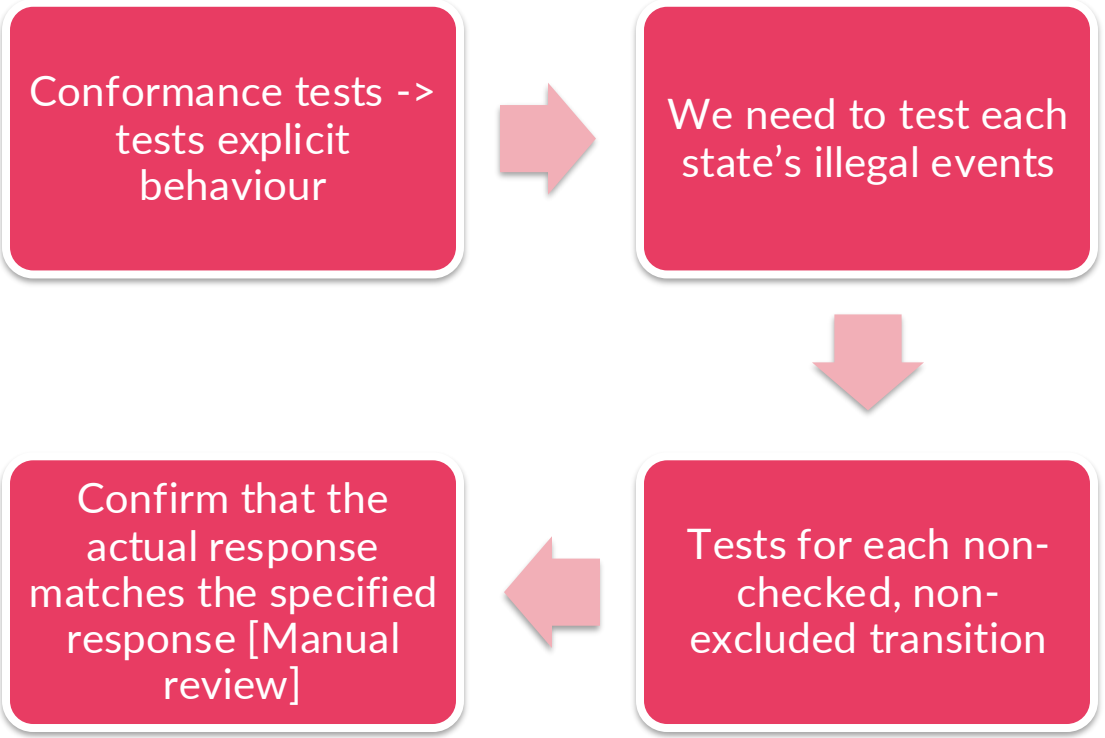
Size : 5

10

Abstract

```
import SUT.Stack;

public class UnitTests {

        Stack sut;
        int x;
        int max;
        @Test
        public void testPath01() {
                max=5;
                sut= new Stack(max);
                x=10;
                sut.push(x);
                assertTrue(sut.isLoaded());
                //Check [n==1]
                //while(sut.getLength()!=1)
                        //sut.push(x)
                sut.pop();
                assertTrue(sut.isEmpty());
        }
}
```

Sneak paths


Conformance tests -> tests explicit behaviour
We need to test each state's illegal events
Tests for each non-checked, non-excluded transition
Confirm that the actual response matches the specified response [Manual review]

N+ Sneak Paths Tests

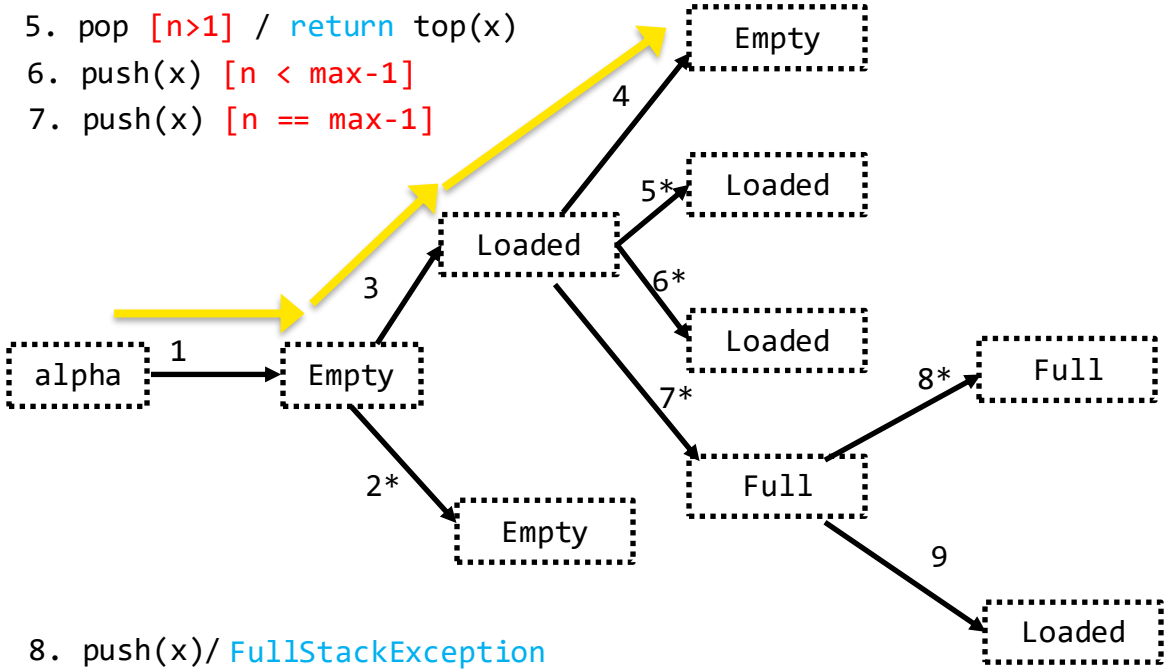
1. ()
"Empty"
[has child]
Do everything but "3 & 2"
Exceptions but still in "Empty"

3. push(x)
"Loaded"
[has child]
Do everything but "4, 5, 6, & 7"
Exceptions but still in "Loaded"

4. pop [n==1] / return top(x)
[has child]
STOP

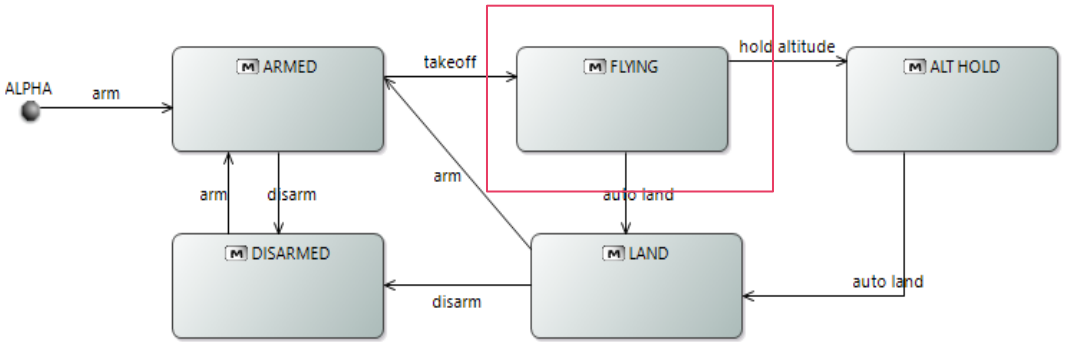
Manual review
(True to life)

1. ()
2. pop/ EmptyStackException
3. push(x)
4. pop [n==1] / return top(x)
5. pop [n>1] / return top(x)
6. push(x) [n < max-1]
7. push(x) [n == max-1]



8. push(x)/ FullStackException
9. pop/return top(x)

N+ Sneak Paths Tests (Another example)



Function	Post
disarm	Armed=False

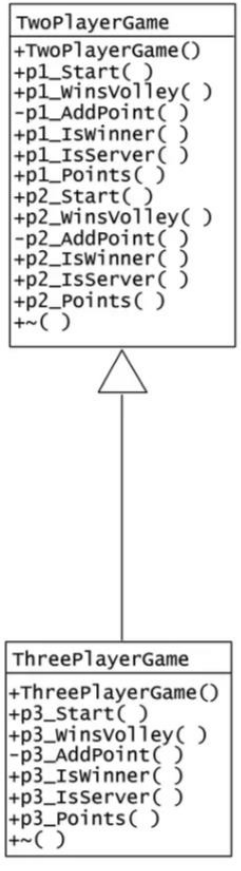
Test Setup	Sequence
arm, takeoff	disarm.InvokedInterface

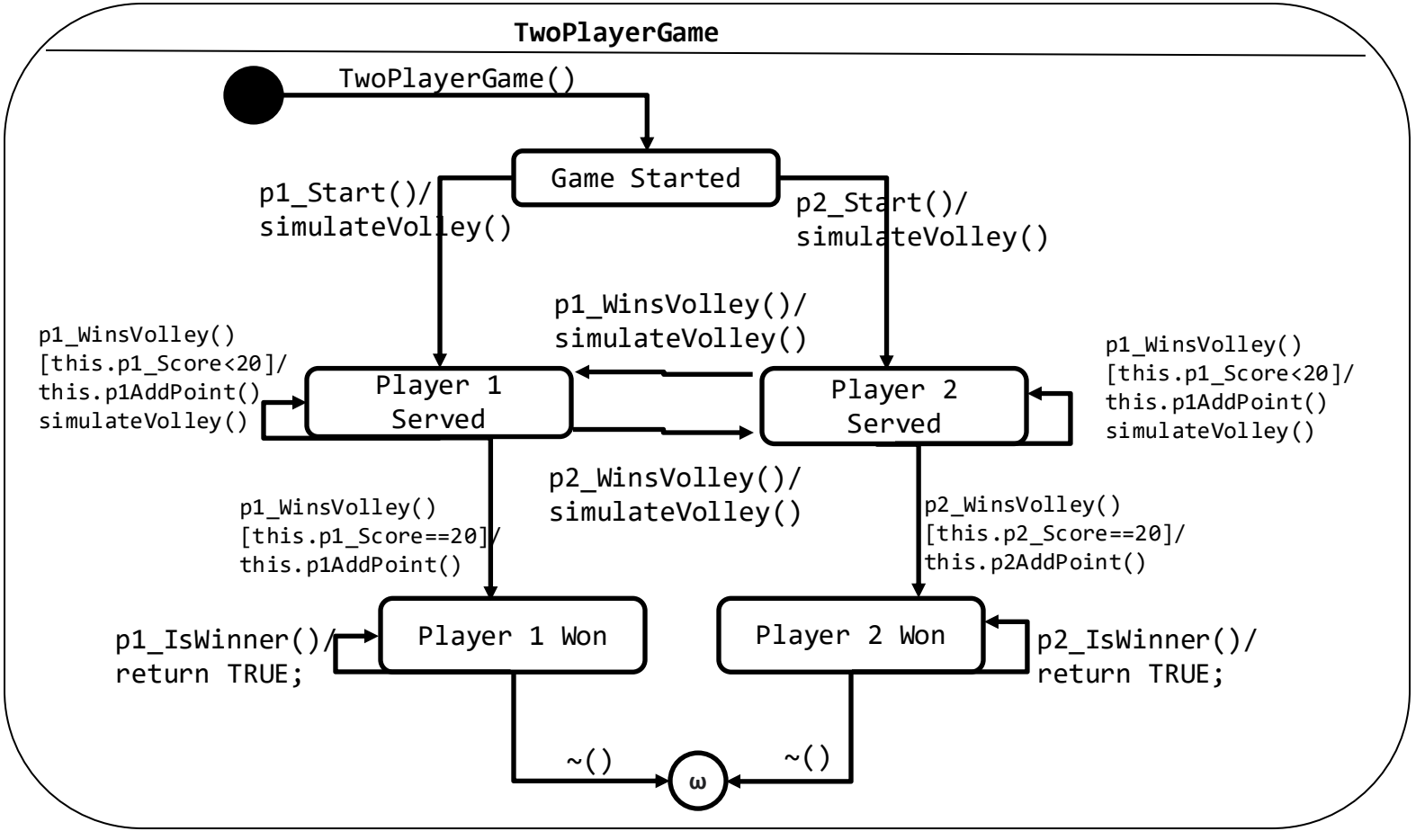
Duration	Oracle
disarm.duration (2)	FALSE(Armed=False)

Should not be able to disarm mid-air

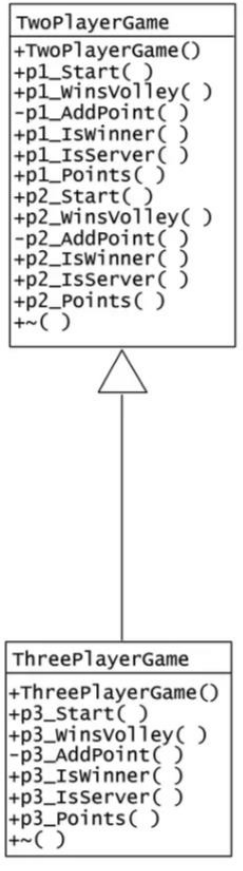
State-based testing of SUT

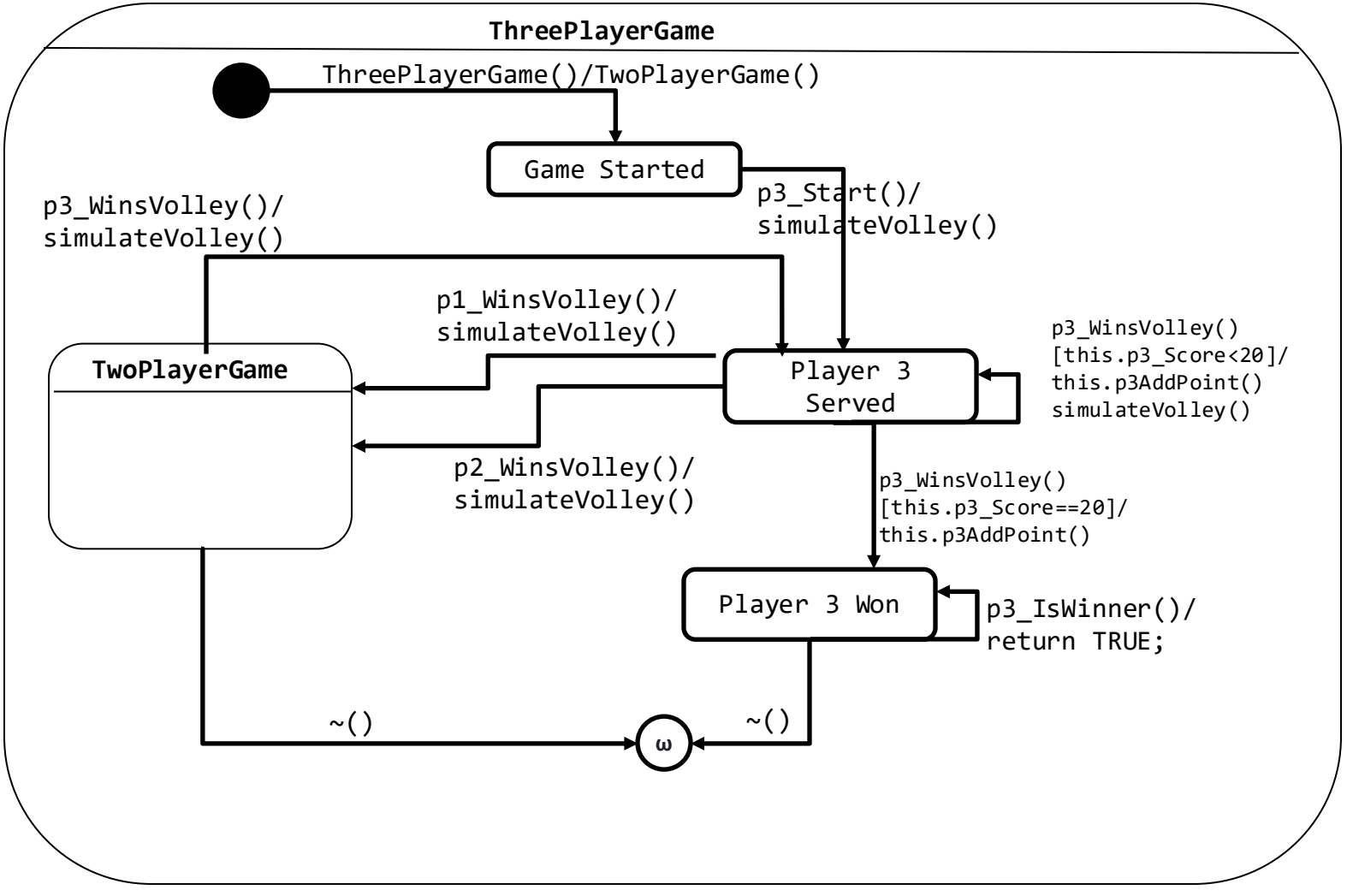
Modelling SUT



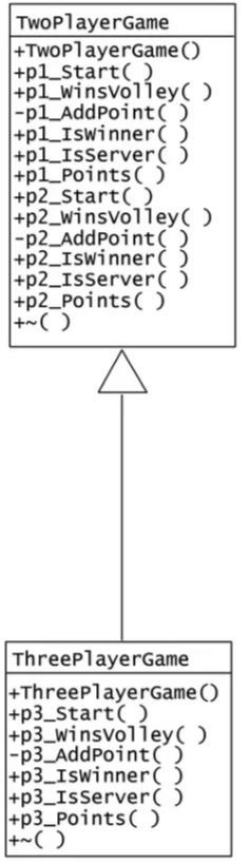


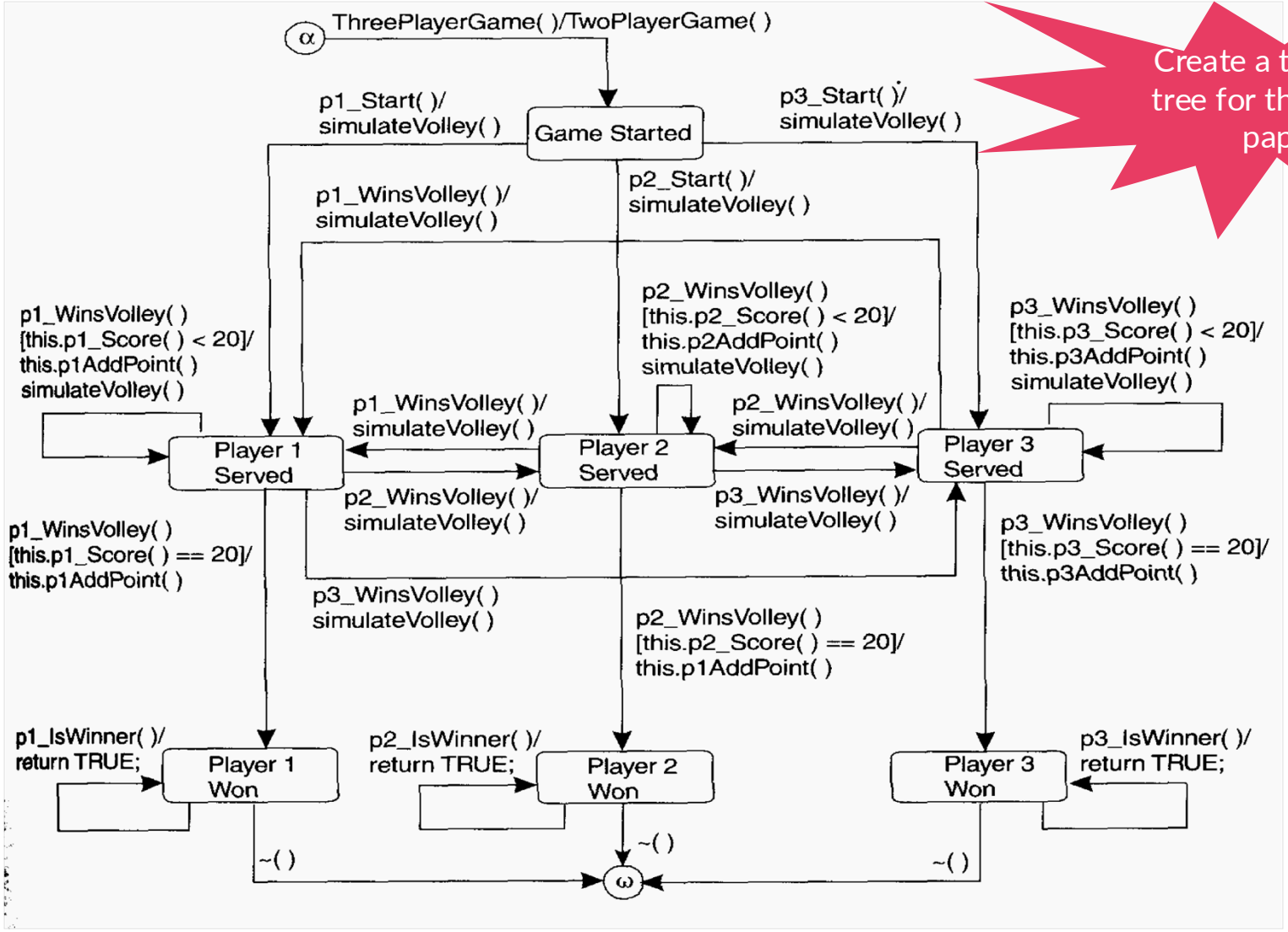
Modelling SUT





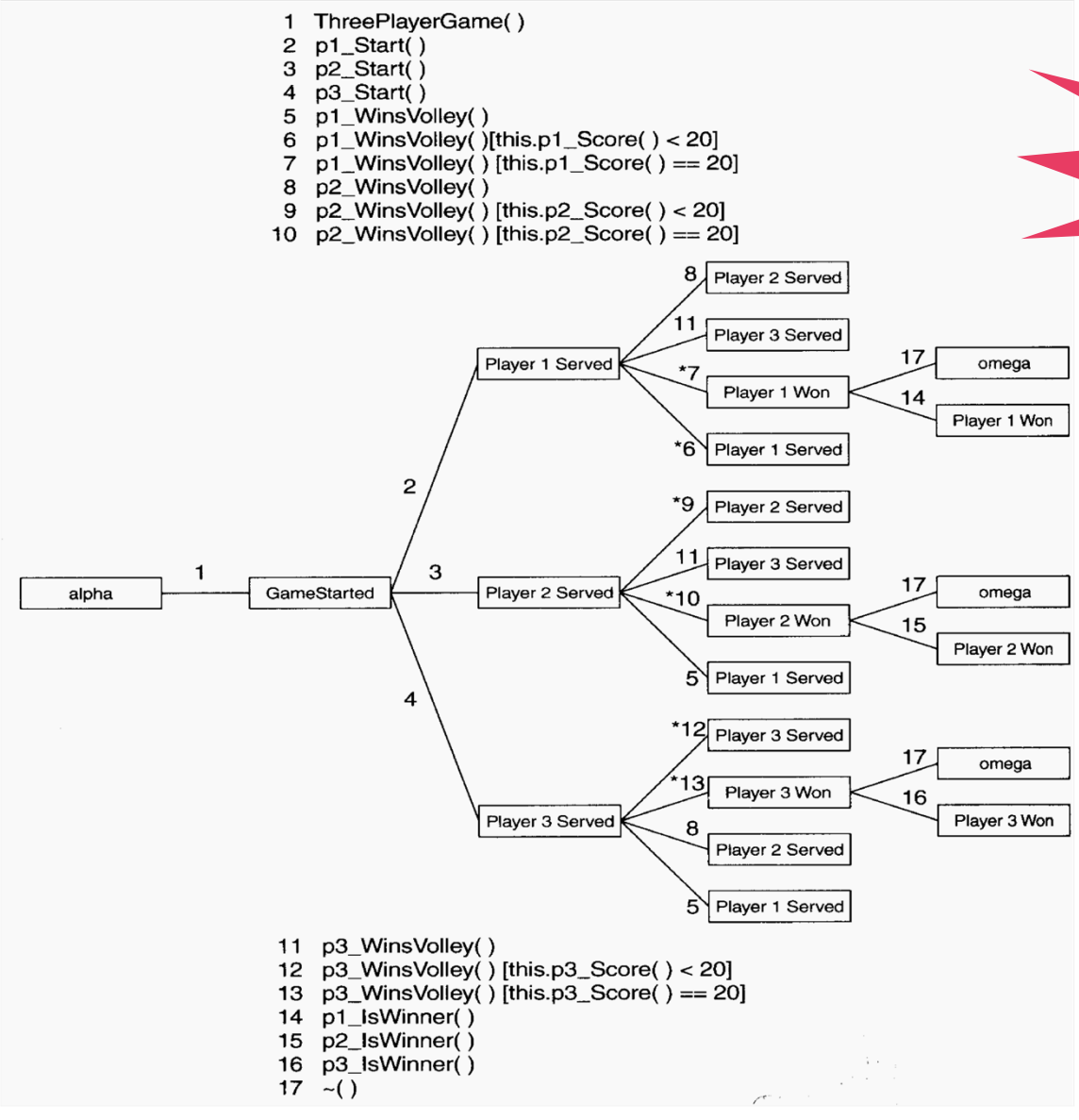
Test Ready Model





Create a transition tree for this SM on paper

Tree



Does it look the same?

SUT Instrumentation for Automated Test Generation

```java
/* @instrumentation */
public String state;
public TwoPlayerGame()
{
        /*Constructor*/
        /* @instrumentation */
        state="Game Started";
}
public void p1_Start()
{
        /*P1 Serves first*/
        /* not implemented method simulateVolley() s
        server=1; // p1 is server
        /* @instrumentation */
        state="Player 1 Served";
}
```

```java
/*
 * State Reporter
 * Code instrumentation
 * Not the part of SUT
 * Code marked as @instrumentation is
 * placed and was not part of the SUR
 */
/* @instrumentation */
public String stateReporter()
{
        return this.state;
}
/* @instrumentation */
public void dtor() // as we don't have destructors in java
{
        state="T";
}
```

Activity 2

- Generate suites for All Transition (conformance), and Sneak Path test suites in Code Playground
- Enhance, review and submit the generated JUnit4 test coverage.

Tip: Be creative with SneakPath tests!

Thank you!

Muhammad Abbas
Senior Forskare
Smart Industrial Automation

muhammad.abbas@ri.se

Jean Malm
Adjunkt
CSE

jean.malm@mdu.se

https://github.com/a66as/StateBasedTestCaseGeneration

RI.
SE