Software Testing, STE
22nd - 26th October 2012
ASSIGNMENT

Tim Pizey
Wellcome Trust Centre for Human Genetics
Oxford University
`Tim.Pizey@gmail.com`

Page count 38

March 9, 2013

[Page intentionally left blank]

Contents

Part I
JDSL Test Management Plan - HI Build

1 Business Strategy

This *Test Plan* is guided by the *Business Plan* which is to market a **High Integrity Build**, where *high integrity* means tested to the highest standards we can devise. The main **risk** to the project is not that it is not completed, as it could currently be sold, but a risk of reputational damage should any failure in a customer's project be correctly attributed to our product. The guiding principle is that there must be no compromise on quality, every effort must be made to discover defects and all defects must be fixed.

The longer term strategy is then to make further releases, at the same quality levels, with updates to align the product with current versions of the Java^{TM}Virtual Machine, specifically updates to use Java^{TM}Collections (Java 1.4) and Java^{TM}Generics (Java 1.5). The tests developed will ensure that there are no regressions due to these upgrades.

The project deliverables are the library class files packaged as a jar, the documentation in javadoc format, and the test sources and class files similarly packaged. The library source files are not part of the saleable product but will be delivered to the business at the project close as a tagged `git` repository.

2 Approach

The project will be managed as a tailored PrinCE2^{TM}project, divided into stages with defined products as the outputs from each stage. This document serves both as the **Project Initiation Document** and the **Project Plan**. The release of a version will be managed as the PrinCE2 process **Managing a Stage Boundary**.

This is not a Test Driven Development, as the code is already in place, nor is it an Agile development, as the product definition is fixed. The Test Development stage will use the simple Kanban board [Trello].

The project will be managed within a closed [Github] repository. Defects discovered will be managed using the [Github Issue Tracker] ([Jira] is considered too heavy weight).

The Continuous Integration server used to run the tests will be bought in as a service from [Cloud Bees] and configured to run the tests against multiple versions of the JVM on multiple operating systems with differing amounts of memory.

3 Organisation

The project will require the following roles, though in fact more than one role may be performed by one individual

Test Manager Responsible for the planning, managing and controlling tests. Responsible for assuring the business that the project has defined the highest quality standards and that the product meets them. Uses [Trello] and [Github Issue Tracker] to maintain a [Burndown Chart].

Configuration Engineer Responsible for setup of Continuous Integration machine and its collection of slave servers. Also responsible for releasing and tagging versions for delivery at each stage boundary.

Test Engineer Responsible for design and implementation of tests, reporting to the Test Manager through pulling work from the [Trello] board.

Software Engineer Responsible for correcting defects, updating the status of issues in the [Github Issue Tracker].

4 Progress and Controls

At the end of each week the **Test Engineer** will create a **Checkpoint Report** giving the number of defects raised, number outstanding, number of classes tested, number of test methods written for each class, number of classes yet to be tested and an assessment as to when the next version of the code and tests should be created by the Configuration Engineer.

4.1 Communication Strategy

The project team will communicate by three mailing lists:

Commits Any commit to the git repository automatically generates a commit message to this list.

Builds failures Any build failures from Jenkins are mailed here. It is expected that the the committer responsible for the failure fixes it and explains the cause and actions taken to fix.

General This is the main project *agora* where all project information dissemination and discussion occurs.

Emails off-list are discouraged.

The Test Engineer will use the [Trello] board to move a class from **To Do** to **Doing** and **Done**, adding a note of the number of tests written to test the class.

To avoid excessive iterations, and excessive versions, there will be close communication between the Test Engineer and the Software Engineer, using the [Github Issue Tracker]. The testers will test against the HEAD revision, which the Software Engineer will also commit to.

The Project Manager will prepare an *End Stage Report* for the business at the end of each stage.

4.2 Cost Control

The project is budgeted to cost $300,000 to the first shipment, of which $240,000 is labour costs, which results in a budget of 4800 hours (2.86 person years) labour.

The labour budget is allocated between the stages, with the majority, 3600 hours, being allocated to the Functional Test Writing stage.

4.2.1 Estimate of Project Scale

The scale of the project is estimated by using method counting as an estimate of [function points]. Note that there will be scope to reuse tests where one interface extends another, as some tests can also be inherited.

Using an estimate of 3 tests per method and 4 tests per argument enables us to arrive at a figure of 959 tests for the project, which in turn gives a burn down rate of 3.75 hours per test.

Simple			**Tests**		
Interface	**Methods**	**Arguments**	Method	Argument	Total
Container	0	0	0	0	0
Deque	8	2	24	8	32
Dictionary	0	0	0	0	0
PriorityQueue	0	0	0	0	0
Queue	5	1	15	4	19
RankedSequence	4	6	12	24	36
SimpleContainer	2	0	6	0	6
SimpleDictionary	7	6	21	24	45
SimplePriorityQueue	4	2	12	8	20
Stack	5	1	15	4	19
Simple: 10	**48**	**18**	105	72	177
Core					
BinaryTree	5	7	15	28	43
BinaryTreeBased	1	0	3	0	3
BookSequence	3	4	9	16	25
CircularSequence	9	13	27	52	79
Comparator	6	11	18	44	62
ComparatorBased	2	1	6	4	10
Container	2	0	6	0	6
Edge	0	0	0	0	0
Graph	9	15	27	60	87
InspectableBinaryTree	3	3	9	12	21
InspectableGraph	20	15	60	60	120
InspectableTree	7	6	21	24	45
KeyBasedContainer	8	10	24	40	64
Locator	4	0	12	0	12
Position	2	0	6	0	6
PositionalContainer	3	4	9	16	25
PositionalSequence	13	11	39	44	83
PriorityQueue	5	2	15	8	23
RestructurableBinaryTree	1	1	3	4	7
Sequence	4	5	12	20	32
Tree	3	5	9	20	29
Vertex	0	0	0	0	0
Core: 22	**110**	**113**	**330**	**452**	**782**
Total: 32	**158**	**131**	**435**	**524**	**959**

4.2.2 Cost Monitoring

Budgets are monitored using a [Burndown Chart], which is updated with actual figures for numbers of tests written per class. The initial budget of 3600 hours requires a burndown rate of 3.75 hours per test.

Given the uncertainties of the estimate only 3000 hours of the budget will initially be released, with an expected burndown rate of 3.1 hours per test.

5 Defect Management

Defects, identified by the creation of failing tests by the Test Engineer, are entered into the [Github Issue Tracker]. Their lifecycle is managed by the Software Engineer and the Test Manager using an agreed set of statuses.

6 Regression Testing

As all tests will always be run within a Continuous Integration testing framework this could be called Continuous Regression Testing.

7 Configuraton Management

The project has two types of product: documents (**management products**) and code.

7.1 Document Versioning

Each document will have the following configuration information in the footer:

Title	Version	Publication Date

7.2 Artefact Versions and Repository Tags

The code artefacts consist in source files and compiled files bundled in a jar file. The source files will also be bundled in a jar, with the same name but with the additional qualifier **-sources**. The test source files will live in a separate sub directory of the project tree. The test sources and class files will be bundled into a single jar file, with the same name but with the additional qualifier **-tests**. All jar files will have accompanying hashes, with the same name but extensions **.md5** and **.sha1**, these are produced by [Maven 3].

Both documents and source files will be held in a `git` repository, which will be tagged at each milestone in the project with a tag of the following format:

JDSL	Version	Qualifier

The version tagging of the source files and the name of the binary artefact (jar) file will be controlled using [Maven 3] and the [Maven Release Plugin].

The **Version** is the next issue version, which will start at 0.1 and progress through to the first issue at version 1.0. The **Qualifier** can be one of

- **empty** no qualifier is used for a release version

- **RCn** where **n** is the Release Candidate Number
- **SNAPSHOT** to indicate work in progress. This is the tag state of the source tree under which all changes are made.

The release tagging procedure, performed by the [Maven Release Plugin], is to remove the qualifier **SNAPSHOT**, commit the pom.xml, tag the tree at the **Version** then increment the version number and add the **SNAPSHOT** qualifier to the pom.xml

The version within the pom.xml determines the name of the binary artefact, the jar file. The deliverable class jar file will have the following names, in the same order. Additional Release Candidates can be added.

- **jdsl-0.1-SNAPSHOT.jar**
- **jdsl-0.1-RC1.jar**
- **jdsl-0.1.jar**
- **jdsl-1.0-SNAPSHOT.jar**
- **jdsl-1.0-RC1.jar**
- **jdsl-1.0.jar**
- **jdsl-1.1-SNAPSHOT.jar**

The release procedure will be performed at the end of each stage.

The use of [Maven 3] fixes some of the trivial issues with previous build tools such as `Ant` and `make`, as all dependencies must be in jars, all jars must be versioned and all Maven plugins used must be versioned. This goes a long way to ensuring that a Maven3 build is repeatable, from machine to machine and from operating system to operating system.

8 Test Environment

The tests should be run on all target hardware and operating systems, however initial testing will be run locally by the Test Engineer, upon commit they will be run on the Jenkins setup.

Jenkins can be configured to run the same test suite against multiple Java Virtual Machines.

This configuration can then be duplicated across a farm of slaves running different operating systems. Current versions of Java^{TM}are able to simulate earlier versions and so the tests will be run with simulated versions 1.3, 1.4, 1.5, 1.6 and 1.7 on the Oracle JVM, the IBM JVM and OpenJDK. This gives fifteen versions of the tests to run on each slave.

The slaves can be configured with differing amounts of memory. For each operating system we will run two configurations: a slave with a small memory, 512mb and one with a typical memory, 4gb.

The operating systems to test against will be Ubuntu 12.04, RedHat Enterprise Linux 6, Windows Server 2012 and OSX 10.8.2.

Further combinations of operating system, JDK and memory can be added with ease, however the above configuration will generate 60 test environments; it maybe that this number is considered not to generate further assurance, so this it will be approached slowly and evaluated for benefit!

Slaves will only run released, tagged, versions. The HEAD version will be run in the top level of Jenkins, reporting any failures as a normal Continuous Integration service.

The tests will be run with JUnit version 3.8.1 as this will run on all versions of Java$^{\text{TM}}$and imposes a discipline on test class and test method names which ensures inter-operation with both the IDE (Eclipse Juno), Maven Surefire plugin and the Continuous Integration server Jenkins.

9 Project Product Description

Title	Description	Stage
Test Plan	This document	Initiation
Git Repository	Repository created with [Maven 3] directory structure	Start up
Issue tracker	Comes with [Github]	Start up
Kanban board	Boards created for each stage at [Trello]	Start up
Baseline code	Java sources for interface definitions and implementations	Adoption
Static Reports	Initial static analysis reports to feed into development	Adoption
	For each class	
Analysis	Comment on congruence between implementation types and semantic types	Test Development
Failure Modes	How the accepted class may yet fail.	Test Development
Constructor tests	Tests for any non-zero argument constructors.	Test Development
Functional Tests	Tests for each method	Test Development
Bench Mark Tests	Speed tests for future ensuring no speed regressions in future versions	Test Development
	For each version release	
Defect Report	Defects to be corrected	Test Running
Tagged code	Defects fixed	SW Development
Acceptance Report	Including **Lessons Learnt**	Closure

10 Stages

The project will be managed in stages, the end of each stage will be defined by the delivery of defined outputs (**products**).

10.1 Initiation

The initiation stage will deliver this document and ensure that the hardware and manpower identified in it have been authorised.

10.2 Start up

The Test Manager will purchase a private [Github] account. The Configuration Engineer will set this up and import the initial code, creating with the [Maven 3] directory structure and a minimal, initial, Project Object Model **pom.xml**.

The [Github Issue Tracker] comes with [Github] but does need to be initially configured with some project specific information, which will be setup by the Test Manager.

The Test Manager will setup the Kanban boards for each stage at [Trello].

10.3 Adoption

The stage consists in the adoption of the existing software. The software was last updated in September 2005, and was arguably old fashioned in its layout and build then.

The *High Integrity Build* will use current best practice vis a Maven 3 [Maven 3] build configuration to establish a repeatable build process. This repeatable build to be placed under continuous integration using the [Jenkins] Continuous Integration build server. All sources will be kept in a Git [Git] source control repository.

Source files will follow the [Java^{TM}Naming Conventions].

The adoption will include adding a [Maven 3] Project Object Model (pom.xml) which will enable the project to be built using [Maven 3] and will configure the testing and quality plugins.

10.3.1 Baseline Reports

The Baseline Build includes no automated tests.

On top of the baseline the following reporting tools will be used:

[**Cobertura**] A test coverage tool which annotates source code lines with the number of times they have been executed and conditionals with whether all branches have been executed. Will initially show no coverage.

[**Findbugs**] Static code analysis to detect use of a catalogue of anti-patterns.

[**Checkstyle**] Ensures coding style is adhered to. Requires configuration to our coding style.

[**PMD**] A source code analysis tool.

[**?**] A source code analysis tool which detects *cut and paste* repeated code.

[**JDepend**] A static analyser with code quality metrics based upon coupling and cyclical references.

These reports will be aggregated using [Maven Dashboard], though [Sonar] may be used instead.

Base lines of all these reports will be archived for monitoring change.

Output of these reports will be shared with Software Engineer to rectify defects, these defects will be recorded as a single defect, to avoid needless duplication.

10.4 Functional Test Development

The basis of test creation will be static analysis of the code using an understanding of the correct, advertised, behaviour of the classes under test. The Test Engineer will select a particular class, choosing classes which do not extend others first, moving it to **Doing** in [Trello], and write tests as the outcome of performing the following tasks. The Trello ticket will already have contain a note of the number of methods of the class.

10.4.1 Analysis of Argument Types

Method parameters will be analysed using *equivalence partitioning* to form a *Boundary Value Analysis*, this will be used to test that values at, above and below each boundary are tested. Additionally `null` arguments will also be tested.

A common mistake in Java programs is to give in to the temptation to use the built in primitive types, such as Integer, or indeed String, rather than define a specific type for your needs. This can result in the range of the type being a super-set of the range actually required. For example using an Integer, with a range of -2^{31} to $2^{31} - 1$, when what is actually wanted is a Position with a range 0 to $2^{31} - 1$. Inspection will reveal these usages and tests will be written to exercise them, though a recommendation to refactor will also be raised as an issue.

10.4.2 State Transition Analysis

Each class that can be viewed as a State Machine will have its transitions enumerated, and each transition tested.

10.4.3 Thread Safety Analysis

Particular care must be taken in the design and testing of code for use in a multi threaded environment. Problems with thread safety derive from shared mutable state. In Java^{TM}the problem is not only ensuring that different threads do not overwrite each other's changes but to ensure that changes made in one thread are visible to other threads. The tools available to ensure thread safety are the `synchronized` keyword, which may be applied to a method or a block, and the `volatile` keyword which can be applied to a variable to ensure that the most recently written version of a shared variable is read by any thread. It is particularly important to realise that the `++` operator is not atomic in Java^{TM}.

The best way to avoid thread safety issues is not to share mutable state [Bloch].

The first step then is to establish by inspection what the shared mutable state is. If it cannot be eliminated by making the data immutable or because the desired functionality relies upon it being shared tests need to be designed to expose possible thread clashes. There are frameworks such as [GroboUtils] which can be used to exercise a class or multithreaded tests can be written by hand.

10.4.4 Memory Leak Analysis

There are some known anti-patterns in Java^{TM}, such as synchronising a finaliser, which can be detected by inspection. They are so well known that they are incorporated in the rule sets within [Findbugs] and other anti-pattern recognisers.

10.4.5 Failure Modes Analysis

How the class might fail, outside of its normal exception handling, with especial attention to whether the class can exhaust memory itself and how it fails when memory has been exhausted by some other cause.

10.4.6 Security Analysis

The main security threat to Java^{TM}systems is from a poisoned library. The reading of the code, and the test coverage, will ensure that no inappropriate security breaching code is included in the library. The addition of a checksum to accompany the library ensures that it is not tampered with after being published.

10.5 Defect Correction

Whilst the correction of defects is undertaken by a different role, the Software Engineer, this stage should be undertaken in parallel with the Functional Test Writing stage. The software Engineer will be driven by the [Github Issue Tracker], changing the status of issues and committing changes.

10.6 Performance Tests

Whilst there are no explicit performance requirements we need to guard against introducing speed degradations, or performance regressions, with changes. To this end we will use [Caliper-CI] benchmark tests against which future builds can be compared.

10.7 Project Closure

10.7.1 Deliverables

Acceptance Report A report detailing the version number (tag) of the accepted tree, the number of tests run, the time taken for tests run. Any qualifications to the acceptance.

Lessons Learnt Recommendations for the next iteration of the project.

Jar files Delivered to sales for release.

git repository archived or handed over to maintenance.

10.7.2 Acceptance Criteria

All tests to be written in JUnit, all to pass. Should any tests not be passing then the Test Manager will list them, with an explanation, in an **Outstanding Defects Report**.

Part II
Test Report

1 Test Environment

The tests should be run on all target hardware and operating systems, given below are the versions actually run on due to limitations of available testing setup :

Processor	OS	Memory	Java^{TM}Version
Intel Core 2 Duo	OSX 10.6.8	4 GB	HotSpot(TM) 64-Bit version 1.6.0_37
Intel i7	Ubuntu 12.0.4	8 GB	OpenJDK 64-Bit version 1.7.0_09

Current versions of Java^{TM}are able to simulate earlier versions and so the tests have been run with simulated Java^{TM}versions 1.3, 1.4, 1.5, 1.6 and 1.7.

The tests were run with JUnit version 3.8.1.

2 Additional Test Machinery

The defects in the SUT prevent the full tests being run successfully, as **JUnit** fails at the first failing assertion in a test. These defects would normally be fixed, to enable the next defect to be detected. For the purposes of this example a new class `FixedStackArray` is supplied on which all the tests run to completion, in a real world development the fixes would be applied directly to the SUT.

Some machinery is introduced to enable the tests to be run on both classes, exposing another design smell: the method `atPosition(int p)` is not specified in the interface `jdslcomp.simple.api.Stack` so we cannot test cleanly to the `Stack` interface.

3 Test Design

The tests are designed using a combination of categorisation, analysis and inspection .

3.1 Method Categories

Creation	Transforming	Non-transforming
Default constructor	push	size
Sized constructor	pop	isEmpty
		top
		atPosition
		toString

Table 1: Method Categories

3.2 Analysis of Argument Types

The semantic type of the `size` argument to the constructor is **Positive Integer Greater Than Zero** which ranges from `1` to `Infinity` however it is implemented using the Java^{TM}primitive `int` which ranges from `Integer.MIN_VALUE` to `Integer.MAX_VALUE` ie it includes negative integers and zero and it has an upper bound. We need to test that the SUT correctly handles all cases where the range of the implementation does not coincide with the range of the semantic type.

Similarly the `position` argument has the same semantic type, **positive Integer Greater Than Zero**, but is implemented as `int` so the same checks need to be made. In addition we know that the SUT is implemented using a java `Object` array which uses a zero based index so we need to test for a mis-match between the `position` in the abstract `Stack` and the concrete `index` of the array.

3.3 State Transition Analysis

The state model in Figure 1 shows the effect on the System Under Test (SUT) of the two state transforming methods `push` and `pop`, the creation and non-transforming methods are excluded for clarity.

Edge	**Operation**	**Before**	**After**	**Transition**	**Return**	**testName**
r1	pop	S1	S1	no	Exception	testPopFromEmpty
r2	push	S1	S2	yes	void	testPushToEmpty
r3	pop	S2	S1	yes	value	testPopEmptying
r4	push	S2	S2	no	void	testPushCentral
r5	pop	S2	S2	no	value	testPopCentral
r6	push	S2	S3	yes	void	testPushFilling
r7	pop	S3	S2	yes	value	testPopFromFull
r8	push	S3	S3	no	Exception	testPushToFull

Table 2: State Table

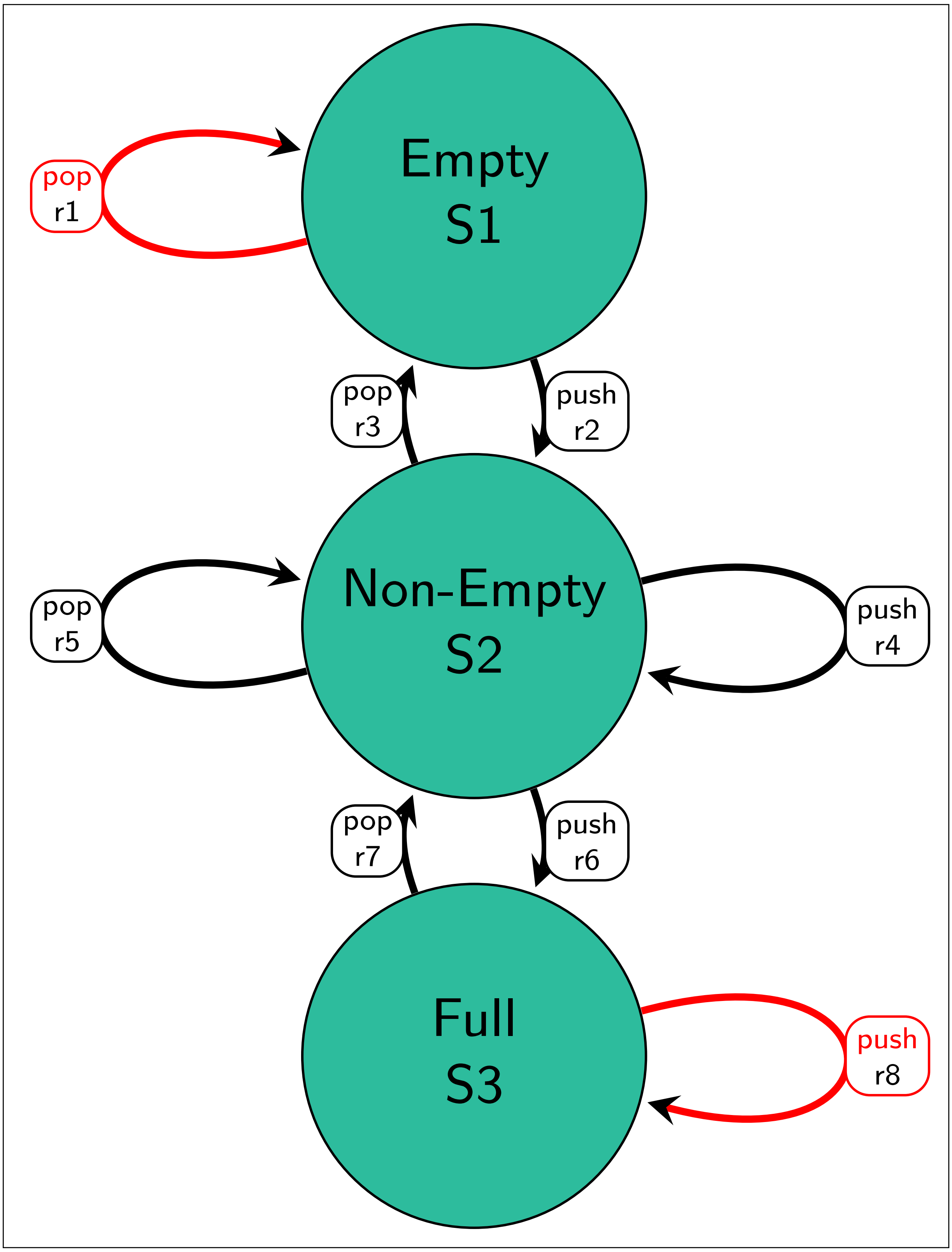


Figure 1: Stack States, exception transitions are coloured in red

3.4 Thread Safety Analysis

To use the `ArrayStack` in a multi-threaded setting it is necessary to `synchronize` on the stack. It is not sufficient to `synchronize` the `push` and `pop` methods as shown in `testUnsafeThreadedAccess()`.

3.5 Memory Leak Analysis

[Findbugs] does not report any anti-patterns.

3.6 Failure Modes Analysis

Two modes of failure have been highlighted: memory exhaustion, where the initial size of the stack is greater than the available memory, exposed by `testMaxConstructor()`, the correct exception is thrown.

The second failure, highlighted in `testUnsafeThreadedAccess()`, arises from misuse: failure to `synchronize` upon the stack object in a multithreaded environment will lead to thread clashes and unexpected results. This is not a defect and cannot be changed but should be documented.

3.7 Security Analysis

No security issues are considered at this level.

4 Tests

Listing 1: Tests

```
package ArrayStack;

import jdslcomp.simple.api.Stack;
import jdslcomp.simple.api.StackEmptyException;
import jdslcomp.simple.api.StackFullException;
import jdslcomp.simple.api.StackOutOfScopeException;
import junit.framework.TestCase;

/**
 * Tests for the ArrayStack.
 *
 * @author timp
 */
public abstract class ArrayStackSpec extends TestCase {

  /** Our ArrayStack has an additional method, over those in
 *  the interface Stack so we need some machinery.
 */
public interface InspectableStack extends Stack {
  Object atPosition(int position);
}

public class Sut implements InspectableStack {
  Stack stack;

  public Sut(Stack s) {
    stack = s;
  }

  @Override
  public int size() {
    return stack.size();
  }

  @Override
  public boolean isEmpty() {
    return stack.isEmpty();
  }

  @Override
  public Object top() throws StackEmptyException {
    return stack.top();
  }

  @Override
  public void push(Object element) {
    stack.push(element);
  }

  @Override
  public Object pop() throws StackEmptyException {
    return stack.pop();
  }

  /**
   * Machinery to work around method not included
   * in Stack interface.
   */
  @Override
  public Object atPosition(int position) {
    if (stack instanceof ArrayStack) {
      return ((ArrayStack) stack).atPosition(position);
    } else if (stack instanceof FixedArrayStack) {
      return ((FixedArrayStack) stack).atPosition(position);
    } else
      throw new RuntimeException("Unexpected␣class␣"
          + stack.getClass().getName());
  }

  @Override
  public String toString() {
    return stack.toString();
  }
}

// Creation Tests
public void testNegativeSizeConstructor() {
  try {
    getSizedStack(-1);
  } catch (NegativeArraySizeException e) {
    fail(failMessage(e)); // bug
  } catch (IllegalArgumentException e) {
    e = null; // expected
  }
}

public void testZeroSizedConstructor() {
  @SuppressWarnings("unused")
  Stack zero = null;
  try {
    zero = getSizedStack(0);
    fail("Should␣have␣bombed␣with␣IllegalArgumentException");
  } catch (IllegalArgumentException e) {
    e = null; // expected
  }
}

public void testOneConstructor() {
  Sut s = getSizedStack(1);
  exerciseSizedStack(s, 1);
}

public void testThreeConstructor() {
  Sut s = getSizedStack(3);
  exerciseSizedStack(s, 3);
}

public void testCapacityConstructor() {
  Sut s = getSizedStack(ArrayStack.CAPACITY);
  exerciseSizedStack(s, ArrayStack.CAPACITY);
}

public void testMaxConstructor() {
  try {
    getSizedStack(Integer.MAX_VALUE);
    fail("Should␣have␣bombed");
  } catch (OutOfMemoryError e) {
    e = null; // expected
  }
}

public void testDefaultConstructor() {
  Sut s = getDefaultStack();
  exerciseSizedStack(s, ArrayStack.CAPACITY);
}

private void exerciseSizedStack(Sut s, int size) {
  assertTrue(s.isEmpty());
  assertEquals("", s.toString());
  try {
    s.top();
  } catch (StackEmptyException e) {
    e = null;
  }
  StringBuffer expected = new StringBuffer();
  for (int i = 0; i < size; i++) {
    s.push(new Integer(i));
    assertFalse(s.isEmpty());
    assertEquals(new Integer(i), s.top());
    assertEquals(s.top(), s.atPosition(i + 1));
    try {
      s.atPosition(i + 2);
    } catch (StackOutOfScopeException e) {
      e = null;
    }
    try {
      s.atPosition(0);
    } catch (IllegalArgumentException e) {
      e = null;
    } catch (ArrayIndexOutOfBoundsException e) {
      fail(failMessage(e)); // bug
    }
    expected.append(i + "␣");
    assertEquals(expected.toString().trim(), s.toString());
  }
  try {
    s.push("breaker");
  } catch (ArrayIndexOutOfBoundsException e) {
    fail("Should␣have␣bombed␣with␣StackFullException");
  } catch (StackFullException e) {
    e = null; // expected
  }

  for (int i = 0; i < size; i++) {
    s.pop();
  }
  try {
    s.pop();
  } catch (ArrayIndexOutOfBoundsException e) {
    fail("Should␣have␣bombed␣with␣StackEmptyException");
  } catch (StackEmptyException e) {
    e = null; // expected
  }
}

// Argument Tests

public void testPushNull() throws Exception {
  Sut s = getDefaultStack();
  s.push(null);
  assertEquals(1, s.size());
  assertNull(s.top());
  assertNull(s.pop());
  assertEquals(0, s.size());
}

// Fails with ArrayIndexOutOfBounds
public void testBadPosition() throws Exception {
  Sut s = getDefaultStack();
  try {
    s.atPosition(-1);
  } catch (IllegalArgumentException e) {
    e = null; //expected
  }
  try {
    s.atPosition(0);
  } catch (IllegalArgumentException e) {
    e = null; //expected
  }
  try {
    s.atPosition(1);
  } catch (StackOutOfScopeException e) {
    e = null; //expected
  }
}

// State Transition Tests

public void testPopFromEmpty() {
  Sut s = getDefaultStack();
  try {
    s.pop();
  } catch (StackEmptyException e) {
    e = null;
  }
}

public void testPushToEmpty() {
  Sut s = getDefaultStack();
  s.push("1");
  assertEquals("1", s.toString());
  assertEquals("1", s.top());
  assertEquals(1, s.size());
  assertFalse(s.isEmpty());
  assertEquals("1", s.atPosition(1));
}

public void testPopEmptying() {
  Sut s = getDefaultStack();
  s.push("1");
  s.pop();
  assertEquals("", s.toString());
  try {
    s.top();
  } catch (StackEmptyException e) {
    e = null;
  }
  assertEquals(0, s.size());
  assertTrue(s.isEmpty());
}

public void testPushCentral() {
  Sut s = getDefaultStack();
  s.push("1");
  s.push("2");
  assertEquals("1␣2", s.toString());
  assertEquals("2", s.top());
  assertEquals(2, s.size());
  assertFalse(s.isEmpty());
  assertEquals("1", s.atPosition(1));
  assertEquals("2", s.atPosition(2));
}

public void testPopCentral() {
  Sut s = getDefaultStack();
  s.push("1");
  s.push("2");
  s.pop();
  assertEquals("1", s.toString());
  assertEquals("1", s.top());
  assertEquals(1, s.size());
  assertFalse(s.isEmpty());
  assertEquals("1", s.atPosition(1));
}

public void testPushFilling() {
  Sut s = getSizedStack(1);
  s.push("1");
  assertEquals("1", s.toString());
  assertEquals("1", s.top());
  assertEquals(1, s.size());
  assertFalse(s.isEmpty());
  assertEquals("1", s.atPosition(1));
}

public void testPopFromFull() {
  Sut s = getSizedStack(1);
  s.push("1");
  s.pop();
  assertEquals("", s.toString());
  try {
    s.top();
  } catch (StackEmptyException e) {
    e = null;
  }
  assertEquals(0, s.size());
  assertTrue(s.isEmpty());
}

public void testPushToFull() {
  Sut s = getSizedStack(1);
  s.push("1");
  try {
    s.push("2");
  } catch (StackFullException e) {
    e = null;
  }
  assertEquals("1", s.toString());
  try {
    s.top();
  } catch (StackEmptyException e) {
    e = null;
  }
  assertEquals(1, s.size());
  assertFalse(s.isEmpty());
}

// Thread Safety Tests

Sut as = getSizedStack(200);
static boolean finished;
static Error firstError;

public void testThreadedAccess() throws Exception {
  finished = false;
  firstError = null;
  for (int i = 0; i < 100; i++)
    new PushPopper().start();
  while(!finished)
   Thread.sleep(10);
  if (firstError != null)
    throw firstError;
}
// Test fails. It is not really a test, more an
```

```
// illustration that incorrect usage will fail.
// Should be moved to its own file and
// excluded from Continuous Integration.
public void testUnsafeThreadedAccess() throws Exception {
  finished = false;
  firstError = null;
  for (int i = 0; i < 100; i++)
    new UnsafePushPopper().start();
  while(!finished)
   Thread.sleep(10);
  if (firstError != null)
    throw firstError;
}

/** Thread safe, synchronized on stack. */
private class PushPopper extends Thread {
  public void run() {
    for (int i = 0; i < 100; i++) {
      synchronized (as) {
        as.push("A");
        try {
          assertEquals("A", as.pop());
          assertTrue(valid(as));
        } catch (Throwable t) {
          firstError = new Error("Failed␣on␣iteration␣" + i, t);
          finished = true;
          Thread.currentThread().notifyAll();
          throw firstError;
        }
      }
    }
    finished = true;
  }
}


/** Thread unsafe, not synchronized on stack. */
private class UnsafePushPopper extends Thread {
  public void run() {
    for (int i = 0; i < 2000; i++) {
      as.push("A");
      try {
        assertEquals("A", as.pop());
        assertTrue(valid(as));
        } catch (Throwable t) {
          firstError = new Error("Failed␣on␣iteration␣" + i, t);
          finished = true;
          throw firstError;
        }
      }
      finished = true;
    }
  }

  abstract Sut getDefaultStack();

  abstract Sut getSizedStack(int size);

  private String failMessage(Exception e) {
    return e.getClass().getName()
        + (e.getMessage() == null ? "" : ":" + e.getMessage());
  }

  public boolean valid(Sut s) {
    for (int i = 0; i < s.size(); i++) {
      if (s.atPosition(i + 1) == null)
        return false;
    }
    return true;
  }
}
```

Listing 2: ArrayStackTest

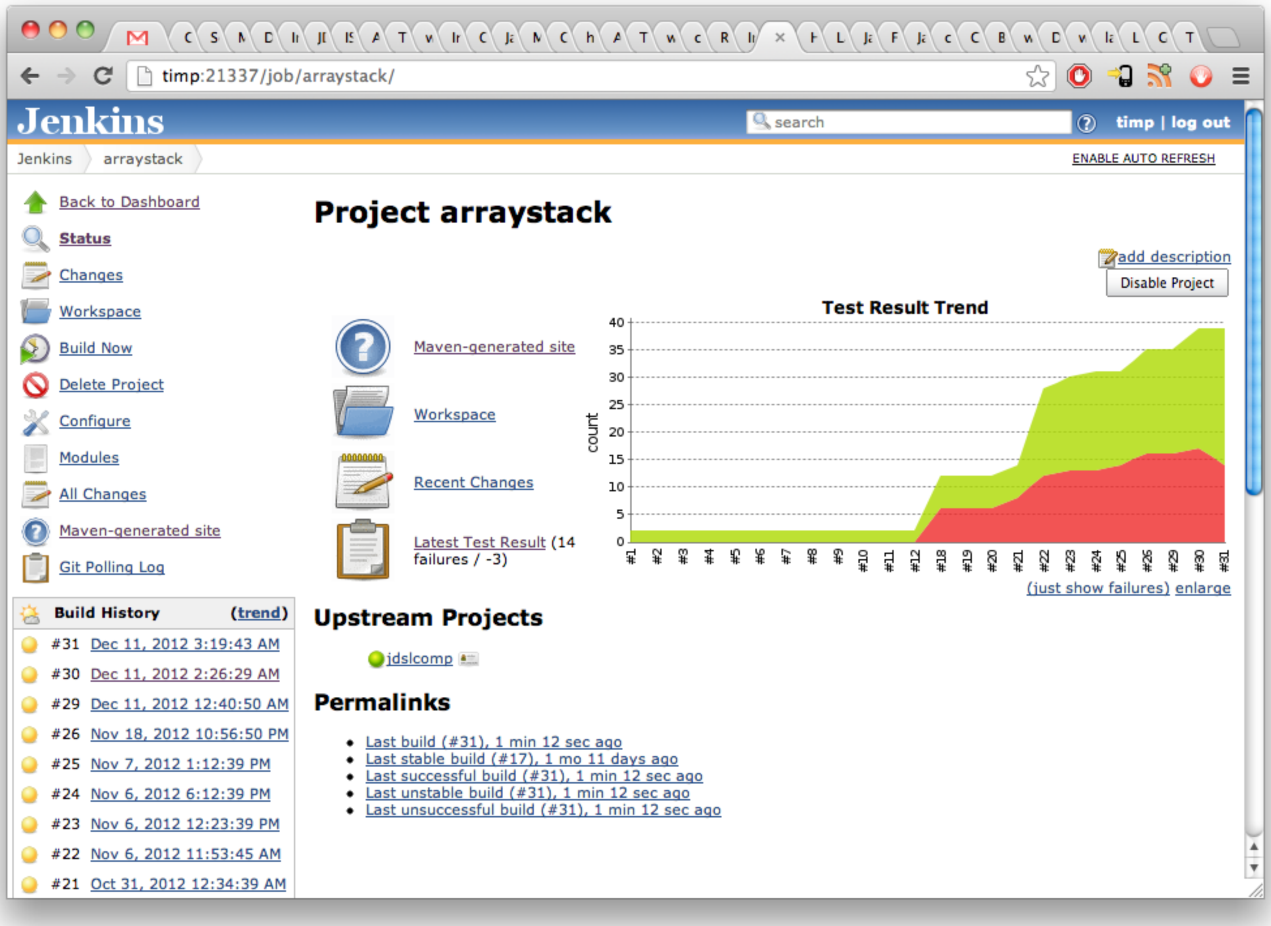
```
package ArrayStack;

public class ArrayStackTest extends ArrayStackSpec {

  @Override
  Sut getDefaultStack() {
    return new Sut(new ArrayStack());
  }

  @Override
  Sut getSizedStack(int size) {
    return new Sut(new ArrayStack(size));
  }

}
```

Listing 3: FixedArrayStackTest

```java
package ArrayStack;

public class FixedArrayStackTest extends ArrayStackSpec {

  @Override
  Sut getDefaultStack() {
    return new Sut(new FixedArrayStack());
  }

  @Override
  Sut getSizedStack(int size) {
    return new Sut(new FixedArrayStack(size));
  }

}
```

5 Results

5.1 Jenkins Results



timp:21337/job/arraystack/

Jenkins | search | timp | log out

Jenkins › arraystack | ENABLE AUTO REFRESH

- Back to Dashboard
- Status
- Changes
- Workspace
- Build Now
- Delete Project
- Configure
- Modules
- All Changes
- Maven-generated site
- Git Polling Log

Project arraystack

add description | Disable Project

- Maven-generated site
- Workspace
- Recent Changes
- Latest Test Result (14 failures / -3)

Build History	(trend)
#31	Dec 11, 2012 3:19:43 AM
#30	Dec 11, 2012 2:26:29 AM
#29	Dec 11, 2012 12:40:50 AM
#26	Nov 18, 2012 10:56:50 PM
#25	Nov 7, 2012 1:12:39 PM
#24	Nov 6, 2012 6:12:39 PM
#23	Nov 6, 2012 12:23:39 PM
#22	Nov 6, 2012 11:53:45 AM
#21	Oct 31, 2012 12:34:39 AM

Upstream Projects

jdslcomp

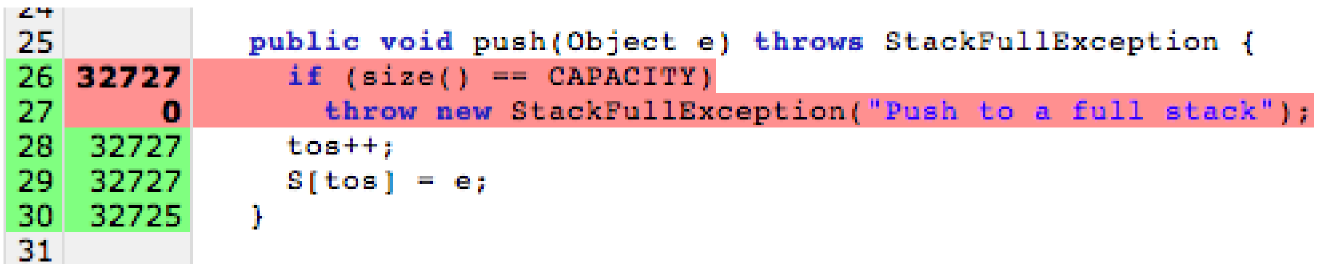
Permalinks

- Last build (#31), 1 min 12 sec ago
- Last stable build (#17), 1 mo 11 days ago
- Last successful build (#31), 1 min 12 sec ago
- Last unstable build (#31), 1 min 12 sec ago
- Last unsuccessful build (#31), 1 min 12 sec ago

5.2 Static Analysis Reports

5.2.1 Cobertura

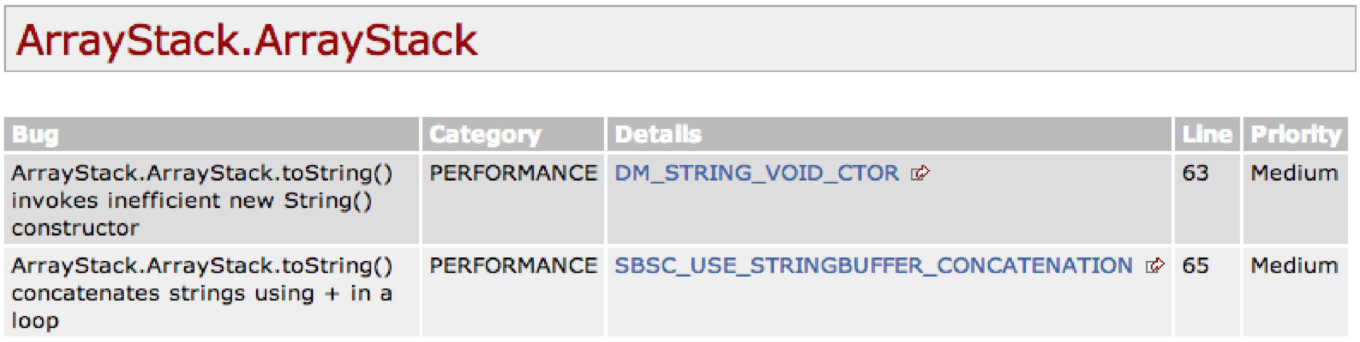
Cobertura shows an uncovered line in push(Object o) caused by defect AS02.



```
            public void push(Object e) throws StackFullException {
32727         if (size() == CAPACITY)
    0           throw new StackFullException("Push to a full stack");
32727         tos++;
32727         S[tos] = e;
32725       }
```

5.2.2 Findbugs

Findbugs discovers two performance issues which can be safely ignored.



ArrayStack.ArrayStack

Bug	Category	Details	Line	Priority
ArrayStack.ArrayStack.toString() invokes inefficient new String() constructor	PERFORMANCE	DM_STRING_VOID_CTOR	63	Medium
ArrayStack.ArrayStack.toString() concatenates strings using + in a loop	PERFORMANCE	SBSC_USE_STRINGBUFFER_CONCATENATION	65	Medium

5.2.3 Checkstyle

Checkstyle finds 40 style issues, from the trivial "File does not end with a newline", through design pattern checks "Method 'isEmpty' is not designed for extension - needs to be abstract, final or empty.".

Checkstyle needs to be configured carefully but can be useful, each issue is worth considering, even if to positively configure Checkstyle not to perform that check.

5.2.4 PMD

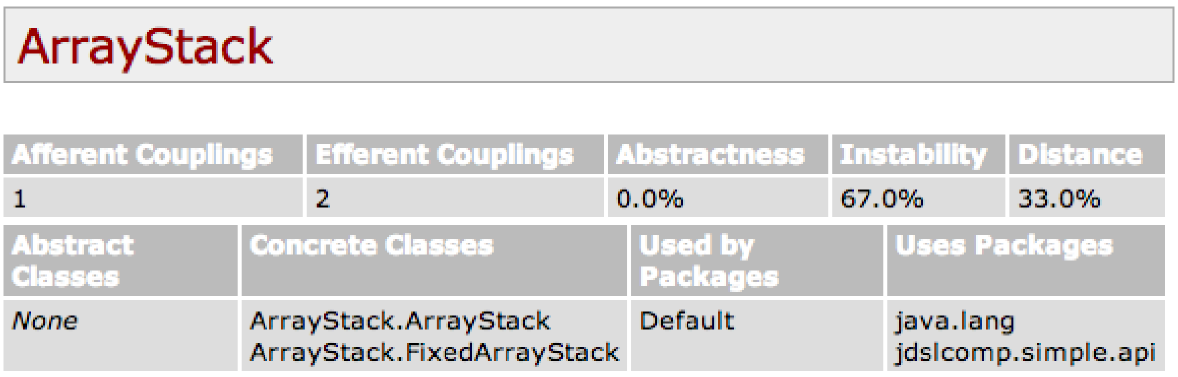
PMD found no issues.

5.2.5 CPD

Not surprisingly picked up the similarities between ArrayStack.java and FixedArayStack.java.

5.2.6 JDepend

JDepend produces a summary of its quality metrics, and an explanation of the code quality model it uses.



ArrayStack

Afferent Couplings	Efferent Couplings	Abstractness	Instability	Distance
1	2	0.0%	67.0%	33.0%

Abstract Classes	Concrete Classes	Used by Packages	Uses Packages
None	ArrayStack.ArrayStack ArrayStack.FixedArrayStack	Default	java.lang jdslcomp.simple.api

5.3 Surefire Results

ArrayStackTest

Test	Status	Reason	Time
testNegativeSizeConstructor	Failed	Threw NegativeArraySizeException	0.004
testZeroSizedConstructor	Failed	Should have bombed	0
testOneConstructor	Failed	Attempt to go beyond top of stack	0.004
testThreeConstructor	Failed	Attempt to go beyond top of stack	0
testCapacityConstructor	Failed	Attempt to go beyond top of stack	0
testMaxConstructor	Passed		0
testDefaultConstructor	Failed	Attempt to go beyond top of stack	0
testPushNull	Passed		0
testBadPosition	Failed	ArrayIndexOutOfBounds -2	0
testPopFromEmpty	Passed		0
testPushToEmpty	Failed	expected:<1> but was:<>	0
testPopEmptying	Passed		0
testPushCentral	Failed	expected:<1 2> but was:<2 >	0.001
testPopCentral	Failed	expected:<1> but was:<>	0
testPushFilling	Failed	expected:<1> but was:<>	0
testPopFromFull	Passed		0
testPushToFull	Failed	ArrayIndexOutOfBounds	0.001
testThreadedAccess	Passed		0.043
testUnsafeThreadedAccess	Failed	Failed on iteration 0	0.111

FixedArrayStackTest

Test	Status	Reason	Time
testNegativeSizeConstructor	Passed		0
testZeroSizedConstructor	Passed		0
testOneConstructor	Passed		0
testThreeConstructor	Passed		0
testCapacityConstructor	Passed		3.098
testMaxConstructor	Passed		0.042
testDefaultConstructor	Passed		2.849
testPushNull	Passed		0
testBadPosition	Passed		0
testPopFromEmpty	Passed		0.001
testPushToEmpty	Passed		0
testPopEmptying	Passed		0
testPushCentral	Passed		0
testPopCentral	Passed		0
testPushFilling	Passed		0
testPopFromFull	Passed		0
testPushToFull	Passed		0
testThreadedAccess	Passed		0.039
testUnsafeThreadedAccess	Failed	Failed on iteration 116	0.117

5.4 Differences between ArrayStack and FixedArrayStack

Listing 4: AS01, AS02. Capacity bug fix; parameter range check

```
20,21c20,24
<   public ArrayStack(int cap) {
<     capacity = CAPACITY;
---
>   public FixedArrayStack(int cap) {
>     if (cap < 1)
>       throw new IllegalArgumentException(
>           "A␣stack␣must␣be␣at␣least␣one␣element␣big.");
>     capacity = cap;
```

Listing 5: AS03, AS04. Synchronize push; Capacity bug fix.

```
25,26c28,29
<   public void push(Object e) throws StackFullException {
<     if (size() == CAPACITY)
---
>   public synchronized void push(Object e) throws StackFullException {
>     if (size() == capacity)
```

Listing 6: AS05. Synchronize pop

```
32c35
<   public Object pop() throws StackEmptyException {
---
>   public synchronized Object pop() throws StackEmptyException {
```

Listing 7: AS06, AS07. Argument checking; tos bug fix

```
56,57c59,64
<     if (i > tos)
<       throw new StackOutOfScopeException(
>                 "Attempt␣to␣go␣beyond␣top␣of␣stack");
---
>     if (i < 1)
>       throw new IllegalArgumentException(
>           "Stack␣position␣must␣be␣greater␣than␣one.");
>     if (i > (tos + 1))
>       throw new StackOutOfScopeException(
>                 "Attempt␣to␣go␣beyond␣top␣of␣stack␣("
>           + i + ">" + (tos + 1) + ")");
```

Listing 8: AS08. Array index bug fix

```
64c71
<       for (int i = 1; i < size(); i++)
---
>       for (int i = 0; i < size(); i++)
```

Listing 9: AS09. Trim toString()

```
66c73
<       return Sout;
---
>       return Sout.trim();
```

6 Defects

ID	Method	Issue	Recommendation
AS01	ArrayStack(int cap)	cap not checked for less than zero.	Add parameter check.
AS02	ArrayStack(int cap)	capacity always set to CAPACITY.	Set capacity to cap.
AS03	push(Object e)	Not synchronized.	Add synchronized keyword.
AS04	push(Object e)	CAPACITY instead of capacity.	Change to capacity.
AS05	pop()	Not synchronized.	Add synchronized keyword.
AS06	atPosition(int i)	No parameter lower bounds checking.	Check i greater than zero.
AS07	atPosition(int i)	Position compared to index.	Add one to index to make comparable.
AS08	toString()	First element of array missed.	Start array index from zero.
AS09	toString()	Always ends in a space.	Should be trimmed.
AS10	atPosition(int i)	Method not in interface.	Add to interface or delete method.

7 Test Assessment

The tests have revealed some serious defects which must be remedied. Once remedied, as per FixedArrayStack, the class is fit for purpose.

7.1 Code Style

The stylistic flaws identified by Checkstyle should be addressed if the class is to be part of a larger suite.

7.2 Code Quality

The threaded tests revealed that the allocation of a very large stack takes an appreciable length of time. There might be performance advantages to allocating the stack space when it was needed, rather than allocating the maximum that might be needed, depending upon whether the maximum stack size is known.

It might be an improvement to create a `Position` type and have the bounds checking in one place rather than using `int` for position parameters.

7.3 Test Improvement

The test `testUnsafeThreadedAccess()` should arguably be moved to a separate file where it can be easily excluded from running on the Continuous Integration server, as it is a proof that misuse is harmful, rather than a failing test.

7.4 Documentation Improvement

The need for synchronization against the stack in multithreaded use should be documented in the class javadoc.

The fact that it is possible to create a stack larger than available memory probably does not need to be documented, as the same can be said for the creation of any array.

References

[Bloch] Effective Java ™- Second Edition
Item 66: Synchronize Access to Shared Mutable Data

[Burndown Chart] Burndown Chart
`http://en.wikipedia.org/wiki/Burn_down_chart`

[Caliper] Caliper microbenchmarks.
`http://code.google.com/p/caliper/`

[Caliper-CI] Integration of `http://code.google.com/p/caliper/` into Jenkins
`http://code.google.com/p/caliper-ci/`

[Checkstyle] Checkstyle is a development tool to help programmers write Java code that adheres to a coding standard.
`http://checkstyle.sourceforge.net/`

[Cloud Bees] Jenkins as a service
`http://www.cloudbees.com/`

[Cobertura] Cobertura is a free Java tool that calculates the percentage of code accessed by tests.
`http://cobertura.sourceforge.net/`

[function points] Functions points
`http://en.wikipedia.org/wiki/Function_point`

[Git] Git - a free and open source distributed version control system
`http://git-scm.com/`

[Findbugs] Find bugs - Find Bugs in Java Programs
`http://findbugs.sourceforge.net/`

[Github] Git Repositories as a Service
`http://github.com`

[Github Issue Tracker] Github Issue Tracker
`https://github.com/blog/411-github-issue-tracker`

[GroboUtils] GroboUtils
`http://groboutils.sourceforge.net/`

[Java™Naming Conventions] Java™naming conventons
`http://en.wikipedia.org/wiki/Naming_convention_(programming)#Java`

[Jenkins] Jenkins - An extendable open source continuous integration server.
`http://jenkins-ci.org/`

[JDepend] A static code analyser which reports cyclic dependencies between packages and classes.
`http://clarkware.com/software/JDepend.html` with a Maven plugin at `http://mojo.codehaus.org/jdepend-maven-plugin/index.html`

[Jira] Atlassian Jira
`http://www.atlassian.com/software/jira/overview`

[Maven 3] Apache Maven - a software project management and comprehension tool.
`http://maven.apache.org/`

[Maven Dashboard] Maven Dashboard
`http://mojo.codehaus.org/dashboard-maven-plugin/`

[Maven Release Plugin] Maven Release Plugin
`http://maven.apache.org/plugins/maven-release-plugin/`

[JATIT] NICHA KOSINDRDECHA, JIRAPUN DAENGDEJ
Journal of Theoretical and Applied Information Technology 2010
`http://www.jatit.org/volumes/research-papers/Vol18No2/4Vol18No2.pdf`

[PMD] Source Code Analyser
`http://pmd.sourceforge.net/`

[Sonar] Sonar
`http://www.sonarsource.org/`

[Synchnization] Oracle Corporation
The Java^{TM}Tutorials
Intrinsic Locks and Synchronization
`http://docs.oracle.com/javase/tutorial/essential/concurrency/locksync.html`

[Trello] Trello Kanban Boards
`https://trello.com/`