JExample

Bachelor's Thesis

Lea Hänsenberger
Supervised by: Adrian Kuhn
University of Bern, Switzerland
Software Composition Group

March 10, 2008

Abstract

Unit tests are primarily written as a good practice to help developers identify and fix bugs, to refactor code and to serve as documentation for a unit of software under test. To achieve these benefits, unit tests ideally should cover all the possible paths in a program. One unit test usually covers one specific path in one function or method. However a test method is not necessary an encapsulated, independent entity. Often there are implicit dependencies between test methods, hidden in the implementation scenario of a test. In this work we present JExample, an extension to the JUnit testing framework, that supports the declaration of explicit dependencies between test methods. Such dependencies either only define the order in which the test methods are to be executed or they additionally manage the returning of an instance of the test fixture by the provider and passing it to the dependent methods. As JExample extends JUnit, yielding compatible test results, JExample test cases can for example be executed in Eclipse's JUnit plugin.

1 Introduction

The key benefits of writing unit tests are that they facilitate changes to a unit of code and that they serve as documentation. When refactoring code at a later date, unit tests help the developer to make sure that the single methods and functions still work correctly. Good unit tests also help a developer to identify and fix new bugs quickly. Moreover, unit tests provide a documentation for the unit under test. What functionality a unit provides and how to use it can usually be learned by browsing the unit tests.

The key question is, what makes a good unit test? In order to be able to identify and fix potential bugs, unit tests must cover all the possible paths in a program. Furthermore they need to be well factorized, i.e. every test method covers one specific path in a method or function. For example when testing the pop method of a stack implementation, one test method tests pop with a full stack, making sure that after pop is performed the stack contains one element less than before. Another test method ensures that an exception is thrown when popping an element from an empty stack. A fine granularity of test methods in unit tests makes it easier to exactly locate a defect.

Nevertheless, fine grained unit tests do not eliminate some shortcomings JUnit has:

- *defect localization:* Since, in JUnit, every test method needs to be an independent artifact, one method of the unit under test might be called in several test methods to, for example, modify the test fixture. Hence, one failing method of the unit under test might cause several test methods to fail. This domino effect makes it more difficult to find the defect.

- *the set up method is not a test method:* In JUnit, the set up method itself is not a test method. Thus, the set up of a test fixture can not be tested. Additionally, a failing set up method causes all the test methods of a test case to fail, without being marked as failed itself.

- *a growing unit under test:* The instance of the unit under test is the only thing all the test methods in a JUnit test case share. This instance needs to be modified during a test run so the test case can cover all the possible paths. Most probably, multiple test methods need to make the same or similar modifications to the unit under test. This leads to a lot of duplicated code in a test case.

Our solution to these problems is to make dependencies between test methods explicit. As mentioned previously, the xUnit family of testing frameworks advises to avoid dependencies between tests because every test method is supposed to be an independent artifact, sharing at most a test fixture creating the unit under test. On the other hand the framework TestNG exists, supporting explicit test dependencies [1]. The advantages and disadvantages of dependencies between tests have received a lot of attention in the literature [2, 3, 7].

As shown by previous research, xUnit tests tend to have implicit dependencies [5]. This suggests that dependencies between tests are inevitable. To illustrate implicit dependencies consider the methods in Listing 1. `testAdd()` and `testRemove()` are implemented to run independently of each other. Nevertheless, `testRemove()` implicitly depends on the outcome of `testAdd()`. Both methods must cover `Set`'s `add` method, hence, an implicit dependency between `testAdd()` and `testRemove()` exists: if `testAdd()` fails `testRemove()` is likely to fail too.

Listing 1: Implicit dependency between test methods.

```
@Test
public void testAdd() {
    Set set = new TreeSet();
    set.add("Foo");
    assertEquals(1, set.size());
}

@Test
public void testRemove() {
    Set set = new TreeSet();
    set.add("Foo");
    set.remove("Foo");
    assertEquals(0, set.size());
}
```

Such implicit dependencies are hard to avoid, as operations for software entities naturally depend on each other [3, 7, 2]. They are not at all uncommon in test code.

This report introduces the testing framework JExample, an extension to JUnit supporting explicit dependencies between test methods (Figure 1). Unlike conventional JUnit tests a JExample test can return an object that may be used as input for other tests. Listing 2 shows the implementation of the test methods shown in Listing 1 as JExample tests.

Listing 2: Explicit dependency between test methods in JExample.

```
@Test
public Set testAdd() {
    Set set = new TreeSet();
    set.add("Foo");
    assertEquals(1, set.size());
    return set;
}

@Test
@Depends( "testAdd" )
public void testRemove(Set set) {
    set.remove("Foo");
    assertEquals(0, set.size());
}
```

JExample solves the previously mentioned problems of conventional JUnit tests as follows:

- Test methods depending on a failing test method are not executed and marked as *white* (ignored). Thus, only the test method covering the actual cause of the domino effect is marked as *red* (failed). The failing test method points directly to the defect location.
- The instance of the unit under test is created in a normal test method that can contain assertions. The instance of the unit under test is then returned and passed as argument to all the dependent methods.

- A certain modification to the unit under test is done in one test method. The unit under test is then returned and passed as argument to all the test methods needing this sort of modified unit under test.

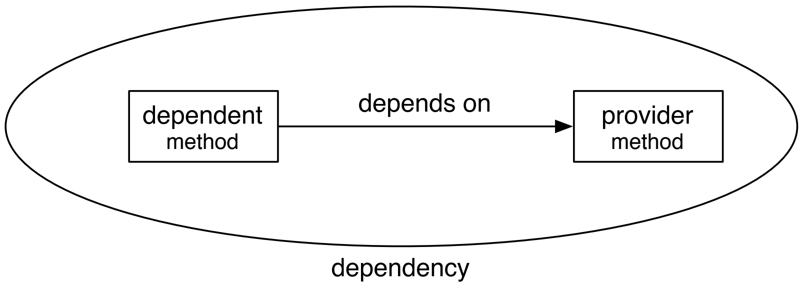


Figure 1: A dependency in JExample

We present the results of a case study where we compare unit tests implemented as JExample tests with conventional JUnit tests. We focus our analysis on the following criteria:

- *Defect localization:* in the presence of defects, the testing framework should focus a developer's attention on the most relevant failing tests. We randomly introduce defects to compare the locality of defects as reported by the framework.
- *Runtime performance:* we measure the execution time of each implementation.
- *Test suite size:* We compute the number of lines, number of test methods, and number of assertion statements.
- *Maintainability:* We evaluate the presence of duplicated code, and other common tests smells, namely General Fixture and Eager Test [2].

2 Related Work

JUnit is probably the best known and widely used testing framework for Java. In April 2006 JUnit 4 was released. This release introduced some major changes to the testing framework. JUnit 4 makes use of annotations to identify the different type of methods used for a test case. `@Before` is used to mark the `setUp()` method and `@Test` to mark test methods. Test methods can now be placed anywhere in the source code, a separate class is not necessary anymore. A simple JUnit test case might look as follows:

Listing 3: A simple JUnit test case.

```
public class SimpleTest {

       @Before
       public void setup() {

       }

       @Test
       public void someTestMethod() {

       }
}
```

The method annotated with `@Before` is executed before the execution of every single test method. Test methods are not allowed to take arguments or to return any object. So every test method stands on its own, sharing only the units created by the set up method.

JExample also uses annotations to build test cases. `@Test` serves the same purpose as it does in JUnit. We introduce a new annotation `@Depends` to specify dependencies between test methods. Furthermore, test methods can take arguments and have return types other than `void`. Because developers can specify explicit dependencies between test methods and test methods can return an object and take an object as an argument, the set up method is no longer needed. The test fixture is created in a normal test method and passed to the dependent test methods by returning it. The advantage is that you can test the set up of the fixture already.

A dependency is specified by passing the `@Depends` annotation a string with the names of the test methods a method depends on. Potential arguments taken by the provider need to be specified with their fully qualified class names so that JExample can locate overloaded methods, e.g. `@Depends("aStringTest(java.lang.String)")`. Multiple providers are separated by semicolons. To use an object returned by a provider, the dependent method must take an object of the same type as an argument. Listing 4 shows a sample test case with dependent test methods.

Listing 4: Dependent methods in JExample.

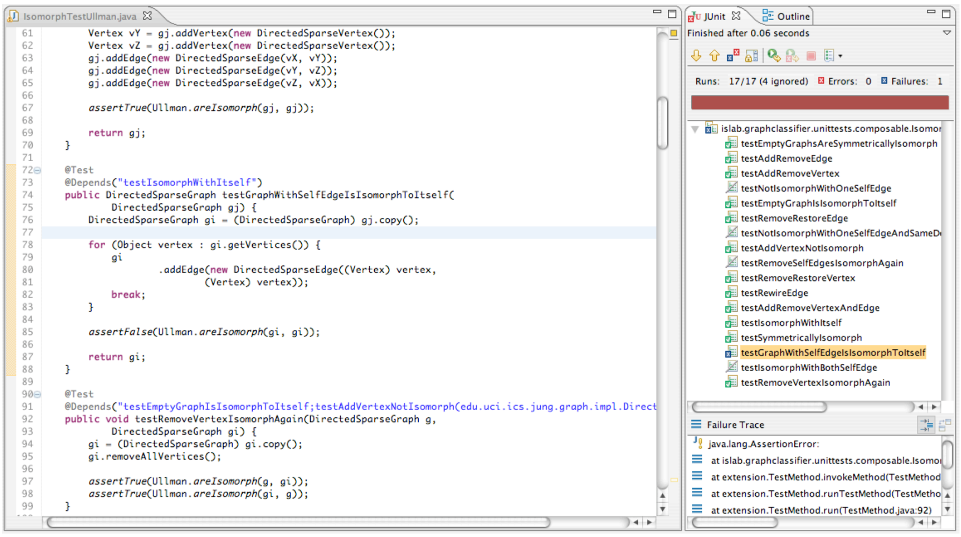
```
public class SimpleTest {

        @Test
        public String setup() {

        }

        @Test
        @Depends("setup")
        public String someTest(String string) {

        }

        @Test
        @Depends("setup; someTest(java.lang.String)")
        public void someOtherTest(String first, String second) {

        }
}
```

A method may also depend on a method of another class. Such provider methods need to be specified with their fully qualified names, for example `@Depends(AnotherTest.someOtherTestMethod)`.

In JExample a test method is not executed until all of its provider methods have been executed successfully. If one of the provider methods fails, is ignored or skipped, the test method is skipped. If in JExample the method setting up the fixture fails for some reason you see immediately where and why it failed. In JUnit, however, all test methods are marked as failed, if the creation of the test fixture failed.

As JExample is built as an extension of JUnit, developers can mix JExample and JUnit code in the same test classes. Figure 2 shows a screenshot of Eclipse, illustrating that JExample test cases are also fully compatible with Eclipse's JUnit plugin.

Another testing framework for Java that allows specification of explicit dependencies is TestNG. Compared to JExample, TestNG does not introduce a new annotation to specify dependencies, it overloads the `@Test` annotation with the value `dependsOnMethods`. Such a dependency specification might look as follows:



```
        Vertex vY = gj.addVertex(new DirectedSparseVertex());
        Vertex vZ = gj.addVertex(new DirectedSparseVertex());
        gj.addEdge(new DirectedSparseEdge(vX, vY));
        gj.addEdge(new DirectedSparseEdge(vY, vZ));
        gj.addEdge(new DirectedSparseEdge(vZ, vX));

        assertTrue(Ullman.areIsomorph(gj, gj));

        return gj;
    }

    @Test
    @Depends("testIsomorphWithItself")
    public DirectedSparseGraph testGraphWithSelfEdgeIsIsomorphToItself(
            DirectedSparseGraph gj) {
        DirectedSparseGraph gi = (DirectedSparseGraph) gj.copy();

        for (Object vertex : gi.getVertices()) {
            gi
                    .addEdge(new DirectedSparseEdge((Vertex) vertex,
                            (Vertex) vertex));
            break;
        }

        assertFalse(Ullman.areIsomorph(gi, gi));

        return gi;
    }

    @Test
    @Depends("testEmptyGraphIsIsomorphToItself;testAddVertexNotIsomorph(edu.uci.ics.jung.graph.impl.Direct
    public void testRemoveVertexIsomorphAgain(DirectedSparseGraph g,
            DirectedSparseGraph gi) {
        gi = (DirectedSparseGraph) gi.copy();
        gi.removeAllVertices();

        assertTrue(Ullman.areIsomorph(g, gi));
        assertTrue(Ullman.areIsomorph(gi, g));
    }
```

```
Finished after 0.06 seconds

Runs: 17/17 (4 ignored)   Errors: 0   Failures: 1

islab.graphclassifier.unittests.composable.Isomor
    testEmptyGraphsAreSymmetricallyIsomorph
    testAddRemoveEdge
    testAddRemoveVertex
    testNotIsomorphWithOneSelfEdge
    testEmptyGraphIsIsomorphToItself
    testRemoveRestoreEdge
    testNotIsomorphWithOneSelfEdgeAndSameD
    testAddVertexNotIsomorph
    testRemoveSelfEdgesIsomorphAgain
    testRemoveRestoreVertex
    testRewireEdge
    testAddRemoveVertexAndEdge
    testIsomorphWithItself
    testSymmetricallyIsomorph
    testGraphWithSelfEdgeIsIsomorphToItself
    testIsomorphWithBothSelfEdge
    testRemoveVertexIsomorphAgain

Failure Trace
java.lang.AssertionError:
 at islab.graphclassifier.unittests.composable.Isomor
 at extension.TestMethod.invokeMethod(TestMethod
 at extension.TestMethod.runTestMethod(TestMetho
 at extension.TestMethod.run(TestMethod.java:92)
```

Figure 2: Screenshot of a JEXAMPLE test case in Eclipse IDE: JEXAMPLE test cases are fully compatible with Eclipse's JUNIT plugin.

Listing 5: Dependent test methods with TESTNG.

```
public class SimpleTest {

        @Test
        public void someTest() {

        }

        @Test(dependsOnMethods = { "someTest" })
        public void dependentTest() {

        }
}
```

Furthermore, TESTNG introduces grouping of test methods. A method's group is defined by the group value of the `@Test` annotation. Due to those method groups a test method can also depend on a whole group instead of a single method. And methods can depend on methods in a superclass. This gives you the possibility to put different parts of tests in different classes, like the initialization of an environment in one class and the tests depending on that initialization in a subclass of this class. A dependency in TESTNG and JEXAMPLE is not exactly the same thing. In TESTNG normal test methods cannot return objects, hence there are no provider methods. To create the test fixture you still need to implement a set up method, identified by the `@Before` annotation. TESTNG introduces the notion of soft dependencies. A soft dependency means that the dependent method is executed after the method it depends on, but it's executed whether this method succeeds or not. Such a dependency is specified with the `@Test` annotation value `alwaysRun = true`. Hard dependencies, however, work as in JExample. If a method fails or is skipped, the dependent methods are skipped.

Test methods in TestNG can also take arguments. The difference between TestNG and JExample is what these arguments are used for. In JExample the arguments are used to pass the test fixture from a test method to its dependent methods, and thus, they are simply the objects returned from the providers. In TestNG the intention is to parameterize tests. There are two possibilities to do that. You can either define an argument in the configuration xml and tell the method with the `@Parameters` annotation which values to take. Or you specify a method as data provider. A data provider is annotated with `@DataProvider` and has to return `Object[][]` or `Iterator<Object[]>`. Methods using this data need to specify which data provider to use by setting the `dataProvider` value of the `@Test` annotation.

Listing 6: Data providing in TestNG with `@DataProvider`.

```
@DataProvider(name = "test1")
public Object[][] createData(){
        return new Object[][]{
                {"SomeString", new Integer(1)}
                {"SomeOtherString", new Integer(2)}
        };
}

@Test(dataProvider = "test1")
public void testData(String str, Integer integer){

}
```

Considering the returned array in Listing 6 the method `testData()` is run two times. First with `"SomeString"` and `new Integer(1)` as parameters, then with `"SomeOtherString"` and `new Integer(2)`.

TestNG provides other features which JExample or JUnit do not. But this is beyond the scope of this report.

There exist other testing frameworks, e.g. JTiger. However, they do not provide support for the declaration of explicit dependencies. They emerged before the release of JUnit 4 because of shortcomings in JUnit 3 such as naming conventions for methods and insufficient possibilities for configuration. With the release of JUnit 4 these frameworks have been rendered obsolete and thus are not relevant in the scope of this work.

3 JExample in a Nutshell

To facilitate dependent test methods JExample extends JUnit as follows:

- Test methods may return values,
- test methods may take arguments,
- and test methods may declare dependencies.

As previously mentioned, in JExample set up methods are obsolete. Any test method M_0 may be used as a set up method using its return value x as test fixture for its dependent methods. JExample takes the return value x of M_0 and passes it as argument to all the methods depending on M_0.

The notion of dependent tests is related to the idea of example-driven testing outlined in the work of Gaelli [4], which states that test fixture instances are valuable objects, and hence, to be reused and treated first-order by the testing framework. Using the terminology of Gaelli we say that each test method consists of an *example*, creating and testing an *example instance* of the unit under test.

JExample uses three colors to indicate a test outcome:

green for successfully passed tests.

red to point out sources of failures.

white to indicate follow-up methods that have been skipped due to failing prerequisites.

This color scheme helps to locate defects: only the test method covering the actual cause of the domino effect is marked as *red*, however, dependent test methods will be skipped and thus marked *white*.

3.1 Writing Tests with JExample

This section illustrates how JExample makes dependencies explicit by exercising a sample test case. Please refer to the the Quick Start Guide in Appendix A to learn how to install and use JExample.

The unit under test is a simple stack implementation. A conventional JUnit test case might go as follows:

Listing 7: Conventional JUnit test case.

```
public class StackTest {

    private Stack stack;

    @Before
    public void setup() {
        stack = new Stack();
    }

    @Test
    public void testPush() {
        stack.push("Foo");
        assertFalse(stack.isEmpty());
        assertEquals("Foo", stack.top());
    }

    @Test
    public void testPop() {
        stack.push("Foo");
        Object top = stack.pop();
        assertTrue(stack.isEmpty());
        assertEquals("Foo", top);
    }

    @Test(expected=IllegalStateException.class)
    public void testPopFails() {
        stack.pop();
    }

    @Test
    public void testPushAll() {
        List list = Arrays.asList(new String[] { ... });
        last = list.get(list.size() - 1);
        stack.pushAll(list);
        assertEquals(last, stack.top());
    }

}
```

When running Listing 7 `setup()` is executed before every test method and the test fixture is passed as field to the test methods. Considering this we may say that all the test methods depend on `setup()`. In JExample `setup()` is a normal test method returning an instance of a Stack.

Listing 8: Promote fixture to test with return value.

```
    @Test
    public Stack testEmpty() {
        Stack empty = new Stack();
        assertTrue(empty.isEmpty());
        return empty;
    }
```

Note, that by setting up the fixture in a test method instead of a set up method, the initialization of the fixture can be tested too. Next, we rewrite `testPush()` and `testPopFails()`. They depend on `testEmpty()` because they need an empty stack as example instance. This is done by annotating them with `@Depends`. The Stack returned by `testEmpty()` is passed to the dependent methods as an argument. In order for a dependency to be valid, the type of the returned object and the type of the argument taken by the dependent test method have to be the same.

Listing 9: Take another test's result as input value.

```
@Test
@Depends("setup")
public Stack testPush(Stack stack) {
    stack.push("Foo");
    assertFalse(empty.isEmpty());
    assert("Foo", empty.top());
    return stack;
}

@Test(expected= IllegalStateException.class)
@Depends("setup")
public Stack testPopFails(Stack empty) {
    stack.pop();
}
```

When running Listing 9 JExample runs `testEmpty()` first, as it is the root of the dependency hierarchy. If `testEmpty()` has run successfully `testPush()` and `testPopFails()` are run with the return value from `testEmpty()` as argument (if `testEmpty()` fails, `testPush()` and `test-PopFails()` are ignored). In order to have the same state of the stack for both dependent test methods the return value needs to be cloned. If the return value's class or one of its superclasses implements `Object.clone()`, this can be accomplished by calling the clone method on the return value before passing it to its dependent methods. Else, the provider method needs to be run again before passing the return value to its dependent methods.

Let's now reconsider Listing 7 to find deeper levels of dependencies in order to get a real graph of composed test methods. We find two methods that depend on `testPush()`: `testPop()` cannot be executed without pushing an element on the stack first and `testPushAll()` will probably also fail if pushing a single element fails.

The new implementation of `testPop()` needs to depend on `testPush()`'s return value. This also avoids the duplicate call to `push`.

Listing 10: Avoid code duplication using dependencies.

```
@Test
@Depends("testPush")
public Stack testPop(Stack stack) {
    Object top = stack.pop();
    assertEquals(true, empty.isEmpty());
    assertEquals("Foo", top);
    return stack;
}
```

`testPushAll()` needs a list of elements as a second argument. Let's assume we have a class `ListTest` with a method `testAddAll()`. The refactored `testPushAll()` is shown in Listing 11. Note that a test may have more than one dependency that can also refer to methods in other test case files.

Listing 11: A test may have multiple dependencies.

```
@Test
@Depends("testPush;ListTest.testAddAll")
public stack testPushAll(Stack stack, List list) {
    stack.pushAll(list);
    last = list.get(list.size() - 1);
    assertEquals(last, stack.top());
    return stack;
}
```

Figure 3 illustrates a sample sequence diagram of the refactored test case: with `Stack`'s `push` method failing, running the test case with JExample will lead to two `white` tests and thus point precisely to the defect location:

1. First, JExample creates a test graph and populates it with the `StackTest` class. The framework processes all methods and their dependencies to initialize the graph of methods. Each method is linked with its providers and marked as colorless, i.e. not yet executed.

2. Then the framework orders the graph to run the test methods. It is topographically sorted and the methods that have no dependencies are executed first. Subsequently the methods depending on these initial methods are executed, etc.

3. `testEmpty()` has no dependencies and thus is executed first. It is marked as *green* and its return value is cached by the framework.

4. Next, the framework attempts to run `testPush()`: it first checks if `testEmpty()` is marked *green*, which is the case, and thus catches the cached return value to pass it as an argument to `testPush()`. But `testPush()` fails and is marked as *red*.

5. Now the framework attempts to run `testPop()`: it checks if `testPush()` has passed, which failed, and hence skips `testPop()` and marks it as *white*.

6. The same holds for `testPushAll()`.

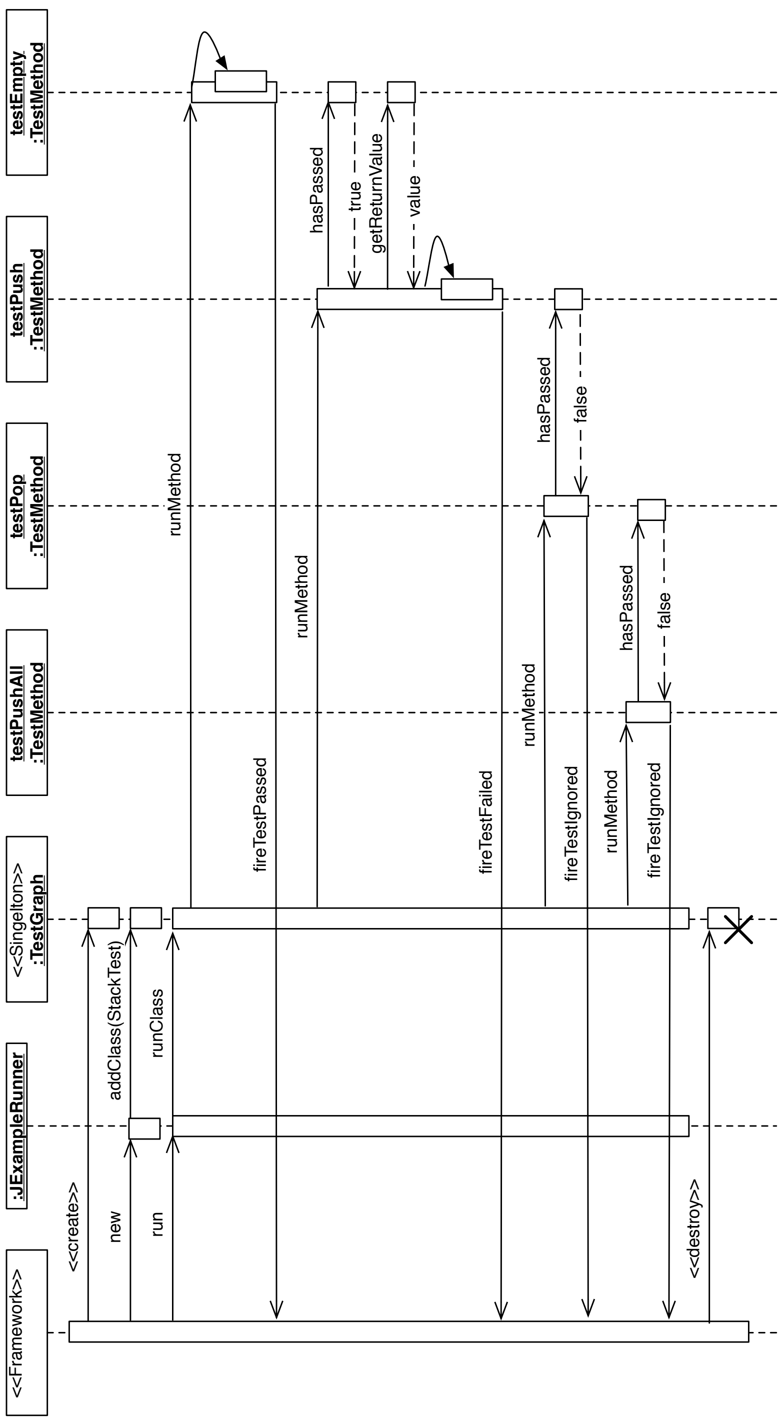


Figure 3: Sequence diagram with `Stack's push` method failing

4 Implementation

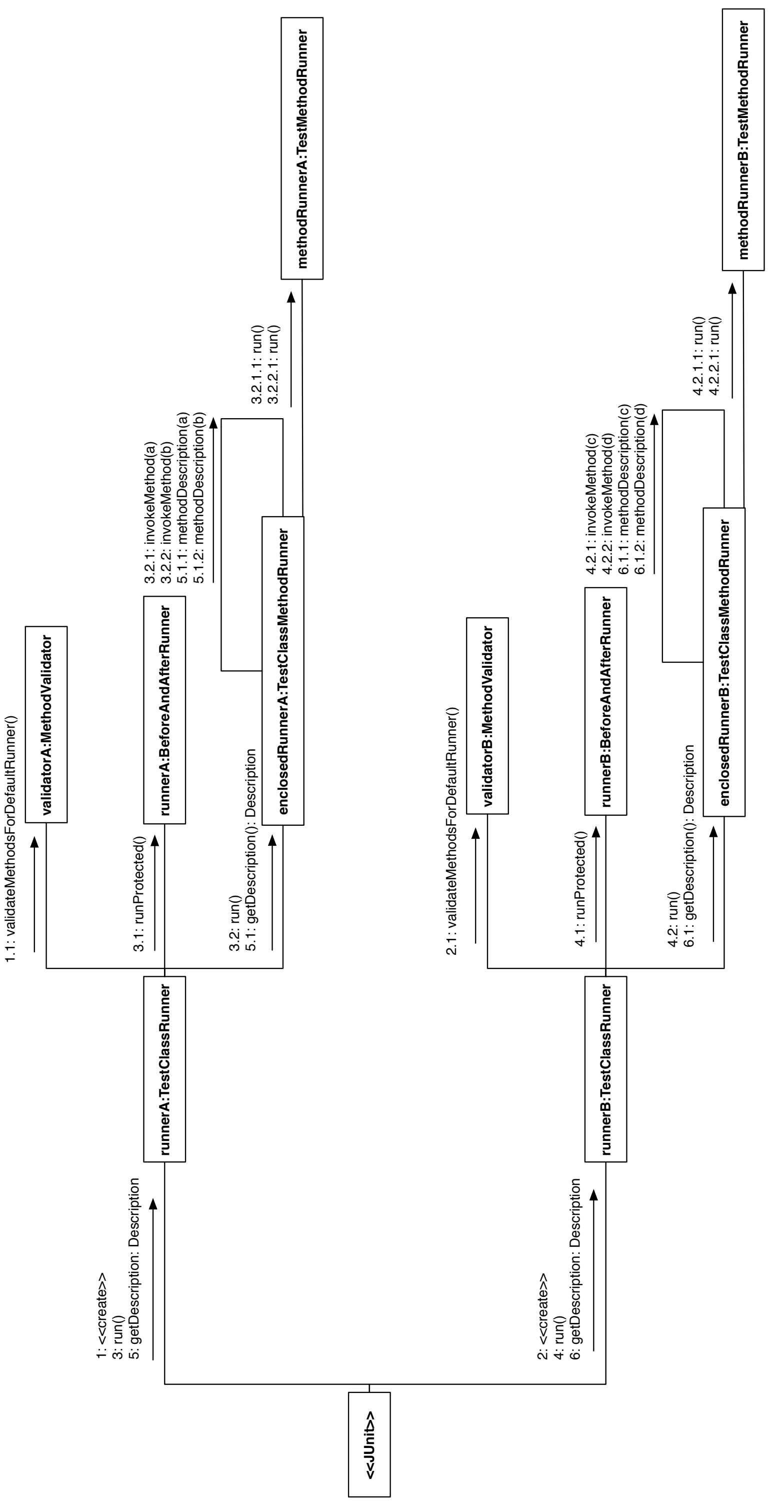
This section first gives a brief overview of what happens behind the scene when running a test in JUnit. Subsequently we explain how JExample is implemented and the motivation and reasons behind our implementation choices.

So, what happens when running a test case? Every test case has a runner object. This runner is either the default runner of the JUnit framework or it is specified by the user with the class annotation `@RunWith`. This annotation takes as value the class of the runner the developer wishes to run his tests with. A runner extends the abstract class `Runner` from the JUnit framework and consists of three parts corresponding to the three steps of a test run. See Figure 4 for running a test case in JUnit.

1. a constructor to initialize the runner (#1 and #2 in Figure 4)
2. a method `run()` to run the test methods (#3 and #4 in Figure 4)
3. a method `getDescription()` to get information about the executed test case (#5 and #6 in Figure 4)

The first thing happening when running a test is initializing a runner instance per test case. Initialization consists of collecting and validating the test methods. The runner gets a list of all the methods declared in the class under test that are annotated with `@Test`. The validation is simply to check if every method is public, has `void` as return type and takes no arguments (of course, this is valid for conventional JUnit methods only, we discuss how JExample differs later in this section). If there are no methods annotated with `@Test` or the validation for one of the methods fails an `InitializationError` is thrown and step two is skipped. If the initialization of the runner is successful, `run()` is called, all the test methods are run and the results are saved. To save the results JUnit provides a `RunNotifier` which tells the `RunListener` when a test is started, when it ends, when it fails or when it is ignored because it is annotated with `@Ignore`. The final result is saved in a `Result` object that knows the number of tests executed, and how many of them passed, failed or were ignored, and what exceptions and errors have been thrown. Finally the framework retrieves the description of the test case that has been run. This description does not contain test results. It just tells the framework the name of the test methods that have been invoked.

One of the goals of JExample was to allow test methods to return values and take arguments. Another goal was to change when and how test methods are invoked. To achieve these goals we had to implement a new runner. Hence, to run a test case with JExample we need to tell JUnit which runner to run it with. This is done with the `@RunWith` annotation. The class definition of a test case in JExample looks as follows:



<<JUnit>>
1: <<create>>
3: run()
5: getDescription: Description
runnerA:TestClassRunner
1.1: validateMethodsForDefaultRunner()
validatorA:MethodValidator
3.1: runProtected()
runnerA:BeforeAndAfterRunner
3.2: run()
5.1: getDescription(): Description
enclosedRunnerA:TestClassMethodRunner
3.2.1: invokeMethod(a)
3.2.2: invokeMethod(b)
5.1.1: methodDescription(a)
5.1.2: methodDescription(b)
3.2.1.1: run()
3.2.2.1: run()
methodRunnerA:TestMethodRunner
2: <<create>>
4: run()
6: getDescription: Description
runnerB:TestClassRunner
2.1: validateMethodsForDefaultRunner()
validatorB:MethodValidator
4.1: runProtected()
runnerB:BeforeAndAfterRunner
4.2: run()
6.1: getDescription(): Description
enclosedRunnerB:TestClassMethodRunner
4.2.1: invokeMethod(c)
4.2.2: invokeMethod(d)
6.1.1: methodDescription(c)
6.1.2: methodDescription(d)
4.2.1.1: run()
4.2.2.1: run()
methodRunnerB:TestMethodRunner

Figure 4: Test run in JUnit

Listing 12: test class definition in JExample

```
@RunWith(JExampleRunner.class)
public class SomeTest {

    .....

}
```

Unlike JUnit's default runner JExample's `JExampleRunner` is a mere delegator. It delegates all of its responsibilities to the `TestGraph` because we need the runner of JExample to be a singleton. When running multiple test cases at once, we ensure that every method is added only once to the test graph and subsequently also executed only once. See Figure 5 for a test run in JExample.

As previously mentioned, the allowed form of methods is different in JExample. Furthermore, the declared dependencies have to be validated. Therefore, the validation of a test case in JExample is more elaborate than it is in JUnit. In Figure 6 you see the classes responsible for the initialization and validation of a test run. During the `addClass` procedure the test graph creates instances of `MethodCollector`, `CycleDetector` and `MethodValidator` for every class added to the graph. The `MethodValidator`, in turn, creates a `DependencyValidator` instance for every validated method.

Nonetheless, the validation of the form of the methods is easier: the only restriction is that they have to be public and cannot be static.

Dependencies, however, must meet the following constraints:

1. a method declared as provider must be a test method, thus must be annotated with `@Test`.

2. if the dependent method takes arguments, the number of dependencies and the number of arguments must be the same.

3. the types of the return values of the provider methods must match the types of arguments of the dependent methods.

4. the dependencies must be acyclic.

Constraints 1-3 are validated by the `DependencyValidator`. The `Cycle-Detector` class detects cycles. The graph of dependencies between methods is topologically sorted. The algorithm for topological sorting also detects cycles. If a cycle is detected during the sorting, the validation is aborted and the framework throws an `InitializationError`.

Also gathering the test methods is not as simple as for conventional test cases. A method can depend on a method of another class that is not part of the current test suite. These methods must be gathered without adding the whole class to the current test run. This is done in the `MethodCollector`

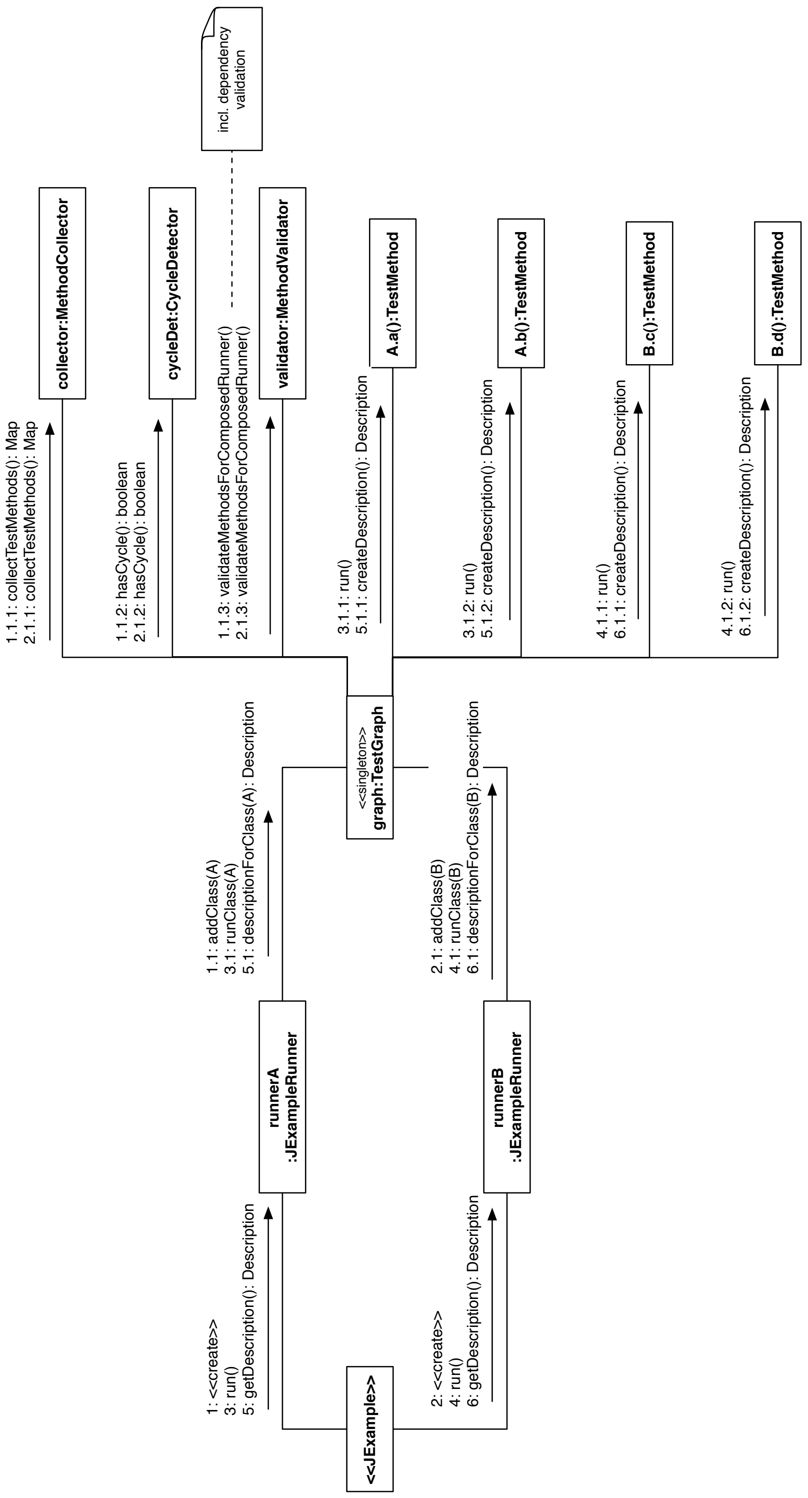
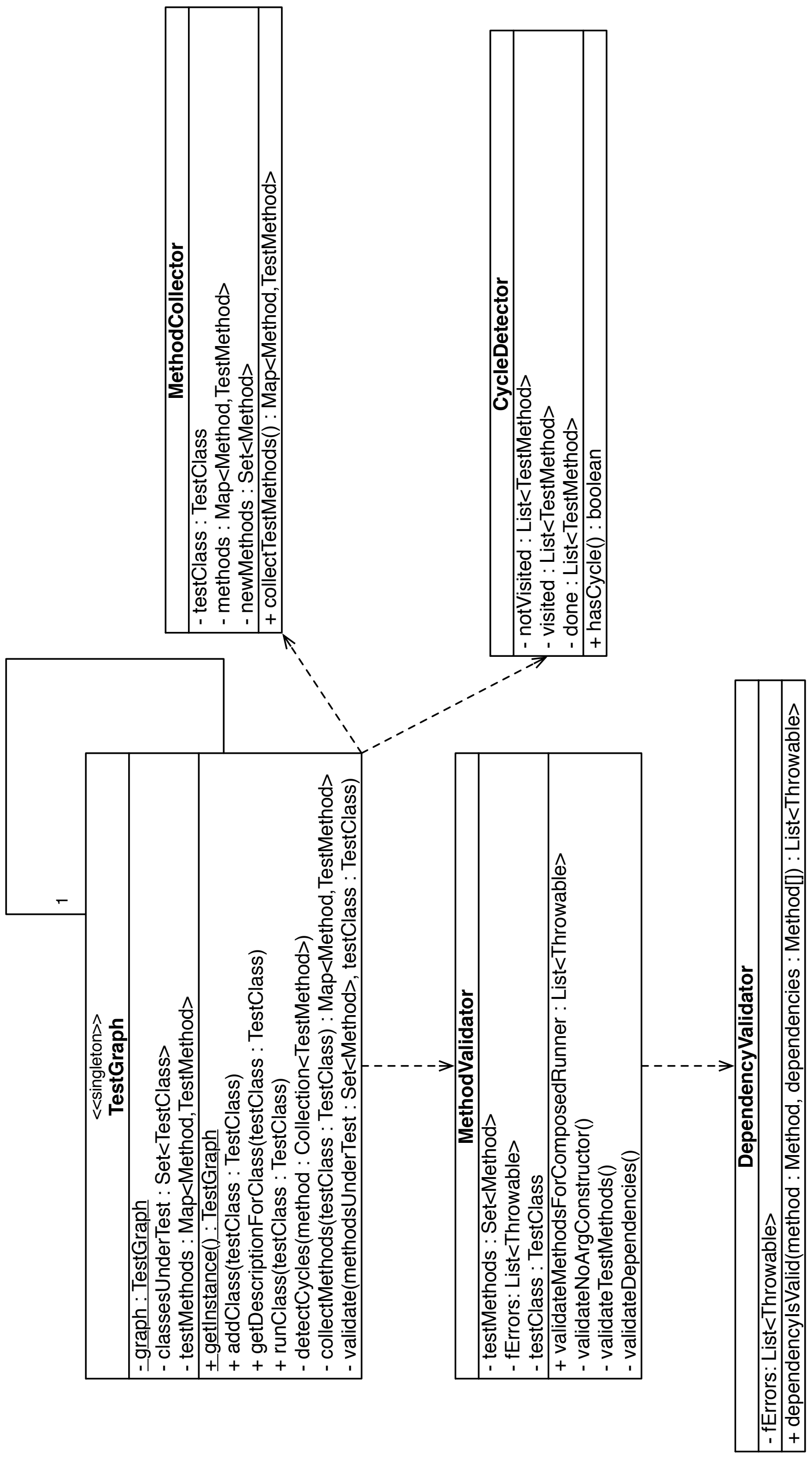

collector:MethodCollector
cycleDet:CycleDetector
validator:MethodValidator
incl. dependency validation
A.a():TestMethod
A.b():TestMethod
B.c():TestMethod
B.d():TestMethod
1.1.1: collectTestMethods(): Map
2.1.1: collectTestMethods(): Map
1.1.2: hasCycle(): boolean
2.1.2: hasCycle(): boolean
1.1.3: validateMethodsForComposedRunner()
2.1.3: validateMethodsForComposedRunner()
3.1.1: run()
5.1.1: createDescription(): Description
3.1.2: run()
5.1.2: createDescription(): Description
4.1.1: run()
6.1.1: createDescription(): Description
4.1.2: run()
6.1.2: createDescription(): Description
<<singleton>>
graph:TestGraph
1.1: addClass(A)
3.1: runClass(A)
5.1: descriptionForClass(A): Description
2.1: addClass(B)
4.1: runClass(B)
6.1: descriptionForClass(B): Description
runnerA
:JExampleRunner
runnerB
:JExampleRunner
1: <<create>>
3: run()
5: getDescription(): Description
2: <<create>>
4: run()
6: getDescription(): Description
<<JExample>>

Figure 5: Test run in JExample



<<singleton>>
TestGraph
- graph : TestGraph
- classesUnderTest : Set<TestClass>
- testMethods : Map<Method,TestMethod>
+ getInstance() : TestGraph
+ addClass(testClass : TestClass)
+ getDescriptionForClass(testClass : TestClass)
+ runClass(testClass : TestClass)
- detectCycles(method : Collection<TestMethod>)
- collectMethods(testClass : TestClass) : Map<Method,TestMethod>
- validate(methodsUnderTest : Set<Method>, testClass : TestClass)
1
MethodCollector
- testClass : TestClass
- methods : Map<Method,TestMethod>
- newMethods : Set<Method>
+ collectTestMethods() : Map<Method,TestMethod>
CycleDetector
- notVisited : List<TestMethod>
- visited : List<TestMethod>
- done : List<TestMethod>
+ hasCycle() : boolean
MethodValidator
- testMethods : Set<Method>
- fErrors: List<Throwable>
- testClass : TestClass
+ validateMethodsForComposedRunner : List<Throwable>
- validateNoArgConstructor()
- validateTestMethods()
- validateDependencies()
DependencyValidator
- fErrors: List<Throwable>
+ dependencyIsValid(method : Method, dependencies : Method[]) : List<Throwable>

Figure 6: Classes responsible for the initialization of a test run in JExample

class. Every test method is checked for whether all of its providers are methods of the same class, and thus, already implicitly added to the test graph. If one provider is located in another test case file, the method is explicitly added to the test graph and its providers are processed too. This procedure is repeated until every provider method of every test method is added to the test graph.

The second step of a test run, i.e. actually running the test methods, in JExample is different as well. The difference to JUnit is the part where the test cases are actually executed (see Figures 4 and 5, procedures 2 and 3 to compare the running of tests in JUnit and JExample respectively). In JUnit a so called `TestClassMethodRunner` exists that creates a `TestMethodRunner` for every test method, which then invokes the actual test method after executing the set up method, and eventually runs the tear down method afterwards (see Figure 4). In JExample every test method is wrapped in a `TestMethod` object. This `TestMethod` knows whether it was already executed, and if so, what was the outcome. Additionally, it knows its providers. The test graph runs its the test methods. The test methods then make sure that they are only executed if all of their providers have been executed and marked *green* and that they are only executed if they have not been executed before because they might be a provider of an already executed test method.

In the remain of the section the classes of JExample are covered in more detail. See Figure 7 for an overview of these classes.

4.1 The JExampleRunner

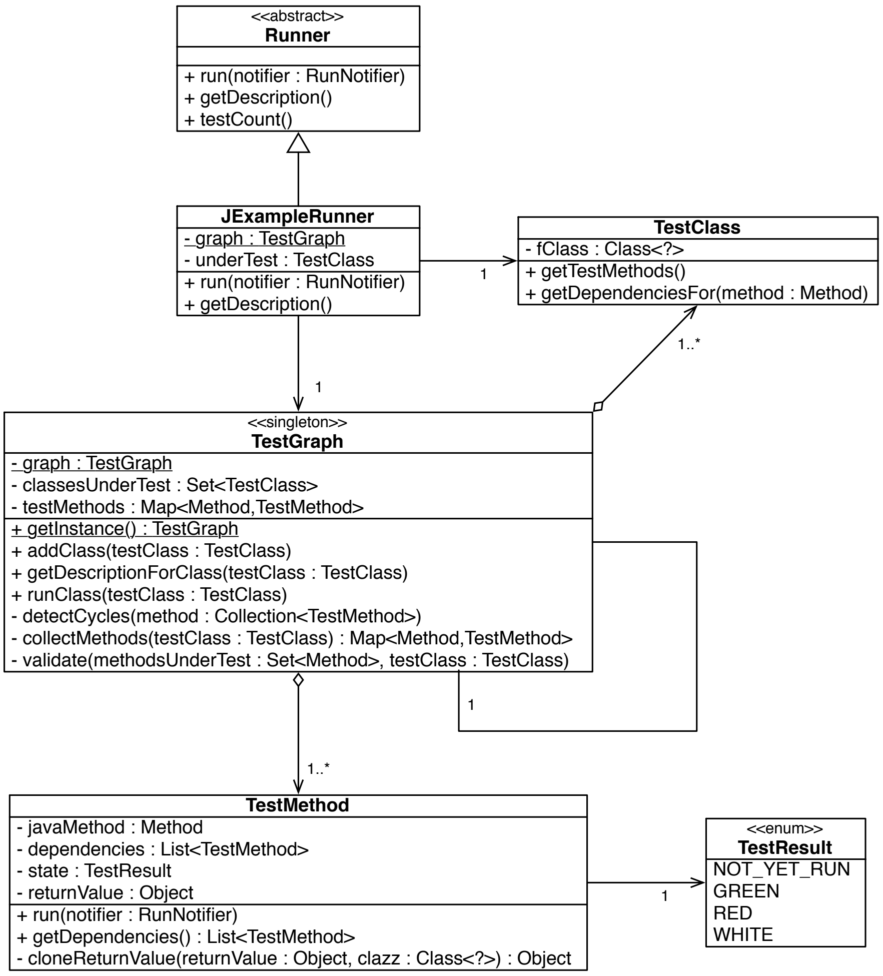
The runner is the main connection to the JUnit framework. It extends the abstract class `org.junit.runner.Runner` and implements two methods `Runner.run()` and `Runner.getDescription()` that call, since `JExample-Runner` is a delegator, `TestGraph.runClass()` and `TestGraph.getDescrip-tionForClass()` respectively. In the constructor of `JExampleRunner` the method `TestGraph.addClass()` is called to initialize the test graph. As mentioned above, a test run consists of three steps:

1. initializing the Runner; this includes collecting and validating methods and test dependencies,

2. running the methods,

3. getting the description of the run.

These three steps exactly correspond to the three methods, respectively one constructor and two methods, of `JExampleRunner`.



<<abstract>>
Runner
+ run(notifier : RunNotifier)
+ getDescription()
+ testCount()
JExampleRunner
- graph : TestGraph
- underTest : TestClass
+ run(notifier : RunNotifier)
+ getDescription()
TestClass
- fClass : Class<?>
+ getTestMethods()
+ getDependenciesFor(method : Method)
1
1
1..*
<<singleton>>
TestGraph
- graph : TestGraph
- classesUnderTest : Set<TestClass>
- testMethods : Map<Method,TestMethod>
+ getInstance() : TestGraph
+ addClass(testClass : TestClass)
+ getDescriptionForClass(testClass : TestClass)
+ runClass(testClass : TestClass)
- detectCycles(method : Collection<TestMethod>)
- collectMethods(testClass : TestClass) : Map<Method,TestMethod>
- validate(methodsUnderTest : Set<Method>, testClass : TestClass)
1
1..*
TestMethod
- javaMethod : Method
- dependencies : List<TestMethod>
- state : TestResult
- returnValue : Object
+ run(notifier : RunNotifier)
+ getDependencies() : List<TestMethod>
- cloneReturnValue(returnValue : Object, clazz : Class<?>) : Object
1
<<enum>>
TestResult
NOT_YET_RUN
GREEN
RED
WHITE

Figure 7: Class Diagram

4.2 The TestGraph

The real runner in JExample is the `TestGraph`. As mentioned earlier, the `TestGraph` is a singleton because it is possible to declare dependencies to methods in other test case files, and thus, we have to make sure, that every method is added and executed only once during a test run. This is possible because of the behavior of JUnit when running multiple test cases at once. First, a runner per test case is initialized, then all the test cases are run and in the end, the description for every test case is collected. See Figures 4 and 5 for a test run with two test cases and two methods each in JUnit and JExample.

The `TestGraph` reflects the three steps of a test run in its three public methods `addClass()` for the initialization, `runClass()` for running the methods and `getDescriptionForClass()` for getting the description of all the run methods. If methods declare dependencies to methods of other classes, those providers are added to the test graph and run as they were part of the class under test. When getting the description, however, it is made clear which method belongs to which class.

4.3 The TestClass

`TestClass` is a wrapper for the class of the test case and responsible for managing and gathering the test methods of the test case. It is only responsible for the test methods declared in this very class file, though.

`TestClass` knows all of its test methods by filtering those that are annotated with `@Test`. In addition it has the responsibility to get the list of providers for methods annotated with `@Depends`. The providers are extracted by parsing the String value of each method's `@Depends` annotation. The `DependencyParser` class separates multiple providers by splitting the string where a semicolon is found. Then it extracts method name, argument types and eventually the class name for each provider and catches the method object from the declaring class.

4.4 The TestMethod

`TestMethod` is the wrapper for the test methods to be run. The `TestMethod` knows its providers and, after it was run, its `TestResult` and its return value. Possible test results are:

NOT_YET_RUN the method has not been run yet

GREEN the method was run and finished successfully

RED the method was run but failed

WHITE [1] the method is either annotated with `@Ignore` or one of its providers has the result RED or WHITE

It makes sure, that all of its providers have been run before it's run itself. It also checks the results of its providers. If one of the providers was marked *red* or *white*, it marks itself *white* too.

If a method takes arguments, `TestMethod` gets the return values from the providers and passes them to the method to be run. The return value is cloned before it is passed to the method if its class or one of its superclasses implements `Cloneable` and overrides `Object.clone()`. Otherwise, the provider method is rerun before the return value is passed to the method.

As a future work, one might consider using transaction semantics instead of cloning, such that an underlying database layer (e.g. Hibernate) might roll back the fixture for each dependent of a test method.

5 Case Study

In this section we report about a case study comparing four different implementations of the same unit test suite. The goal is to check how explicit dependencies between test methods improve defect localization, and how JExample adheres to quality criteria like performance, size and duplication.

5.1 System under Study

The (pre-existing) JUnit test suite under study exercises an implementation of the *Ullmann subgraph isomorphism* algorithm [9] — i.e. an algorithm to compare the structure of graphs. The implementation of 194 SLOC[2] (Cyclomatic complexity of 62 [10]) forms the core of a research tool for frequent subgraph mining in bioinformatics. As such, this set of rigorous, white box tests were written to verify the implementation as well as the interaction with a third party graph library (Jung[3]). As this algorithm is so crucial to the whole tool, the purpose of the test suite goes beyond verification, becoming documentation for this part of the system as well.

[1]To remain compatible to JUnit the test result WHITE overloads JUnit's result IGNORED.

[2]source lines of code as measured by sloccount — http://www.dwheeler.com/sloccount/

[3]Jung — `http://jung.sourceforge.net/`

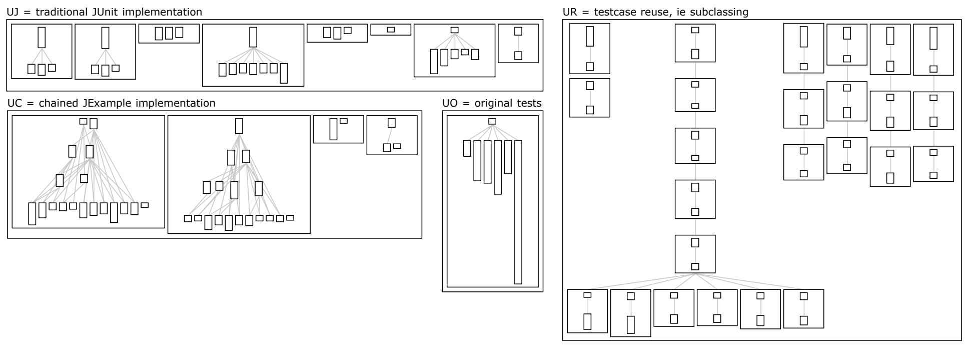

UJ = traditional JUnit implementation
UR = testcase reuse, ie subclassing
UC = chained JExample implementation
UO = original tests

Figure 8: Polymetric view of the four alternative test suite implementations: The innermost boxes represent test methods (including setup methods); the height of the boxes shows the method's LOC (lines of code) metric; edges show test dependencies. The enclosing boxes represent test cases; edges show test case inheritance.

Ullman Original (UO) This is the original implementation of the case-study. As Figure 8 illustrated, the test suite is implemented as a single test case consisting of six very long test methods. The test method concentrate on a *growing unit under test* and are hence implemented as an alternating series of initialization and assertion code, entangling fixture and test code. For example, one of the methods starts with an empty graph object, performing some assertions, adding a vertex to the graph, performing some more assertion, adding another vertex, performing yet more assertions. The method continues to mix assertion code with code that extends the graph initialization with multiple vertices and edges, with reflexive edges, with loops, etc.

5.2 Alternative Implementations

We refactored this original test suite implementation to three alternatives (i) best practice JUnit tests; (ii) JUnit using test case inheritance; and (iii) dependent tests using JExample. The goal of this refactoring was to obtain equivalent implementations of the same test suite using different test design styles.

Ullmann JUnit-style (UJ) This implementation follows the original JUnit test guidelines as described by Beck and Gamma [11]. In summary, they advise to group tests on the same unit under test together, by storing the fixture objects in instance variables of the test case subclass and initializing them in the setup method. To apply this style to the Ullmann case-study, the original test suite is split into eight test cases that each focuses on a different snapshot of the growing unit under test.

Ullman test case Reuse (UR) This implementation relies on test case subclassing to build a set of chained yet isolated test cases. This implementation uses a specific subclassing pattern, turning each iterative initialization step found in the original test into a subclass of if its own, that calls `super.setUp()` in its setup method to reuse previous initialization code. With this design, the dependencies between tests are declared using the test case inheritance hierarchy as a workaround.

Ullmann JExample-Style (UC) This implementation introduces explicit dependencies using JExample. Mirroring the iterative initialization code of the original test methods, using the dependency mechanism presented in Section 3.1: a root test method creates an example instance of an empty graph object, and passes the instance on to its dependent methods. The dependent methods modify the example instance a bit, check some assertion, an eventually pass the instance on to another level of dependent instances, and so on. Dependencies between methods are declared by the developer using `@Depend` annotations, whereas passing on a method's return values to its dependents is done by the framework while running the test suite.

Figure 8 presents the test design of the four implementations using polymetric views [12]. The innermost boxes represent test methods (including setup methods). The height of these boxes show the method's SLOC (source lines of code) metric, while edges show test dependencies. The enclosing boxes represent test cases; edges show test case inheritance.

5.3 Evaluation Procedure

To evaluate *defect localization*, the main improvement goal of JExample, we work with test scenarios during which some tests fail. When executing a test suite, a developer aims at quickly identifying the root location of a failure to repair any defect(s). As such, the number of reported failures should be minimized. Using the explicit dependencies between tests, JExample ignores tests depending on a failed test. As such, we expect the UC implementation to report fewer failures than the other three implementations.

To evaluate the four implementations for this criterion, we randomly introduce defects in the system's code with a mutation testing tool (Jester [8]) and count the number of errors for each test suite implementation.

From the set of common test quality criteria, we select *test suite runtime performance*, *test size* and *code duplication* as evaluation criteria for the four alternative implementations of the Ullmann test suite. In particular, we quantify the adherence of the implementations to these criteria using the following set of metrics:

- *Performance.* To have feedback on the latest changes fast, regression tests should execute fast. With JExample, test runs should *perform better* compared to the alternative implementations, because the unit under test is not reinitialized before every test. Instead, the results of one test are passed on to the next one. Moreover, in case of a failing tests, all dependent tests are ignored (i.e. not executed). Holding back these dependent tests reduces the test execution time as well. We measure the execution time of each implementation, from launching the Java VM until test executions passed and return command to the terminal. To measure execution time of test suites with failures, we reuse the created mutations.

- *Size.* We aim at minimizing the size of the test code, as it is an artifact that needs to co-evolve with the software system. We assume that the JExample implementation forms an improvement over alternative implementation for the size criterion, as there is no need for recurring fixtures or setup. We calculate the overall size (source lines of code) of the test suite as well as the number and size of test methods. While the former tells us something about the code base as a whole that needs to be maintained, the latter identifies how well the test suite is factorized.

- *Duplication.* Reported as one of the worst smells in source code, duplicated code should be avoided. In the alternative implementations, we tried to avoid introducing code clones. As such, by evaluating duplication we want to verify how much inherent duplication is due to the nature of the implementation. The UC implementation is expected to contain less duplication than UJ, due to the absence of recurrent setup behavior. Compared to UO and UR, the duplication should be at about the same level, as these approaches are already focussed on reuse. We measure the amount of duplication[4] in each implementation as a result of the presence or absence of reuse possibilities.

For comparison reasons, we control a couple of test suite equality factors. First of all, we ensured that all four implementations exhibit the same coverage, being 96.9% (Java) instruction coverage (measured using Emma[5]). Secondly, we aimed at keeping the same amount of asserts. Ultimately, there appeared slight differences (between 81 and 85 asserts) due to reuse opportunities. Furthermore, all implementations received the same layout and use — to the extent possible — the same test framework idioms (e.g. all implementations use try/catch clauses with a *fail* assert instead of relying on the JUnit 4 annotation system). Finally, all four implementations were compiled with the same Sun Java 6.0 compiler.

[4]using CCFinderX — http://www.ccfinder.net

[5]http://emma.sourceforge.net

5.4 Results

Defect localization. In order to quantify the traceability quality of the four implementations, we create eight scenarios MUT1-MUT8 where a single mutation in the Ullmann code causes the tests to fail. Jester changes constants in the code and adds clauses in boolean conditions to test the defect detection strength of a test suite. The following changes were applied:

- In MUT1, the starting value for a loop iterator was changed from 0 to 1.
- In MUT2, a *false &&* clause is added to a composed condition, resulting in the code block within the condition never being executed.
- In MUT3, an integer actual parameter to a recursive call is increased by 1.
- In MUT4, a assignment of value 0 to an array location is replaced by 1.
- In MUT5, an *n-1* statement in an condition is changed into *n-2*.
- In MUT6, *o != null* in a condition is replaced with *o == null*.
- In MUT7, a assignment of value 1 to an array location is replaced by 2.
- In MUT8, an actual parameter is changed from 0 to 1.

For each mutation scenario, we then measure the number of failures JUnit reports. Knowing that only one mutation has been introduced at a single location, ideally only a single failure should be reported.

Table 1: Number of failures: absolute number/ignored tests (relative number).

	UO	**UJ**	**UR**	**UC**
MUT1	4 (66%)	12 (46%)	14 (52%)	2/12 (6%)
MUT2	2 (33%)	2 (8%)	2 (7%)	2/0 (4%)
MUT3	1 (17%)	10 (38%)	9 (37%)	1/12 (3%)
MUT4	1 (17%)	1 (4%)	1 (4%)	1/0 (3%)
MUT5	1 (17%)	10 (38%)	9 (37%)	1/12 (3%)
MUT6	1 (17%)	1 (4%)	1 (4%)	1/1 (3%)
MUT7	1 (17%)	10 (38%)	9 (37%)	1/12 (3%)
MUT8	4 (66%)	11 (42%)	9 (37%)	2/14 (4%)

Table 1 presents the number of failures, in absolute numbers as well as a percentage of the amount of tests, reported during test runs on the mutated Ullmann code. For UC, we add the number of tests ignored by JExample. The results show that for a single mutation, typically multiple tests fail. In the case of the original implementation, the number of failures varies between 1 and 4 (out of 6 test methods). For UJ and UR however, up to 14 tests fail as a result of the factorization of test methods. Due to the ordered test execution of the UC implementation, at most 2 tests fail during a test run, while up to 14 are being skipped.

Overall, test runs on the UJ and UR implementations report five times more defect locations than the JExample tests. The original implementation, however, only reports about 36% more defect locations.

Performance. Table 2 lists the average execution time for the four test suite implementations, and for 5 scenarios. In the first scenario called SUCC(ESS), we execute the tests on the original Ullmann implementation. In the four cases we reuse the mutations of Ullmann created earlier. The results are collected as the average execution time of 30 test runs, measured with the UNIX *time* command on an Intel Pentium 4, 3 Ghz, 1 Gb Ram, Sun JDK 1.6.0, JUnit 4.3.1.

Table 2: Average execution time of 30 test runs

	UO	**UJ**	**UR**	**UC**
SUCC	0.512	0.600	0.747	0.554
MUT1	0.506	0.603	0.753	0.539
MUT2	0.516	0.609	0.753	0.554
MUT3	0.500	0.597	0.748	0.534
MUT4	0.516	0.600	0.751	0.556
MUT5	0.499	0.594	0.752	0.533
MUT6	0.513	0.610	0.747	0.547
MUT7	0.502	0.600	0.755	0.537
MUT8	0.515	0.613	0.754	0.546

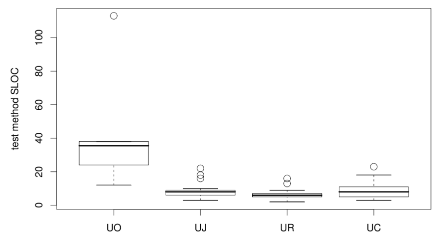
For the successful test run on the original implementation of the Ullmann algorithm, we observe that the original test suite implementation has the fastest average, followed by UC, UJ and UR. We apply Student's t-test to verify whether there exists a significant difference between the test execution time sample sets. As a result, we can indeed conclude — with a confidence of 95% — that UC is 7-8% faster than UO, yet 8-12% faster than UJ and 35-41% faster than the UR approach.

The results do not indicate significant performance increases for the mutation scenarios, except for mutations where 12 to 14 tests are skipped in the UC implementation. Both UO and UC then run about 10% faster compared to their SUCCESS scenario.

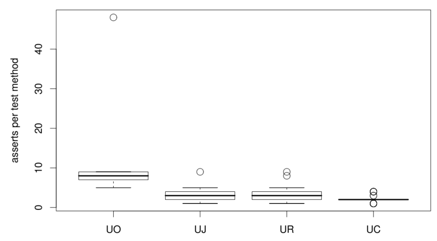
Size. Table 3 summarizes the size and duplication results. The original Ullmann test implementation, lacking any form of encapsulation for individual tests or setup, is the most concise one. As a consequence, the UJ and UC implementations, about the same size, are 77% and 87% larger. UR is even 2.4 times as large.

Table 3: Size of the four implementations expressed in Source Lines Of Code (SLOC), Number Of Test Cases (NOTC), Number Of Test Setups (NOTS) and Number Of Test Methods (NOTM).

	UO	**UJ**	**UR**	**UC**
SLOC	311	551	735	582
NOTC	1	8	26	4
NOTS	1	8	25	0
NOTM	6	52	54	34



(a) Distribution of source lines in test method



(b) Distribution of asserts in test methods

Figure 9: Boxplots for the distribution of test method properties.

Due to explicit setups and fixtures for the multiple test cases the original Ullmann test case (NOTC - number of test cases) has been refactored into, and the method header code for many test commands, the alternative implementations are better factored, as Figures 9(a) and 9(b) show. The average test method length has dropped from close to 40 to around 10, while the number of asserts per test method decreased from 14 to below 4.

Duplication. To evaluate the level of code duplication, we used CCFinderX to calculate and report code clones. Table 4 summarizes the results. The original implementation contains the least duplication, with 266 tokens (out of 3446) involved in any code clone. The alternative implementations contain between 4200 and 4500 tokens, 13 to 22% of which is listed in code clones. UC contains about 5% more code covered by clones than UJ. The reasons for this higher code duplication in UC are twofold. i) the clone method of the third party graph library doesn't work properly, so we had to copy the graph objects with the properly working `copy()` function before modifying them, ii) the vertices and edges of a graph cannot be accessed through a name or ID, therefore we had to remove or modify vertices and edges by iterating over the set of vertices respectively edges.

Table 4: Code clone results, configured with a minimum clone token size of 50, soft shaper and p-match in CCFinderX.

	#tokens	**% tokens**
UO	266	7.7%
UJ	597	13%
UR	938	22%
UC	780	18%

5.5 Conclusion

In this work we introduced the idea of making dependencies between tests explicit to improve defect localization. In a small example we demonstrated that many dependencies between tests are hidden in the implementation scenario of a test that is being defined.

We implemented JExample as an extension to JUnit. JExample allows developers to explicitly declare dependencies between test methods. This information is exploited by JExample's test suite runner by (i) executing tests according to the order defined by these dependencies; and (ii) skipping tests that depend upon a failing test.

Our experience with the case study yields promising results. JExample tests indeed have better defect localization. The execution time compared to the one of UO is somewhat slower and UC contains some more source code. But JExample tests rely upon good unit testing practices of encapsulated, concise test methods. Thus, JExample combines the best of both worlds: it exhibits the benefits of dependencies between test methods with the test quality aspects of JUnit style testing.

Availability A prototype of JExample and the two implementations of the case study are available for download at: `http://scg.unibe.ch/Resources/JExample/`.

References

[1] C. Beust and H. Suleiman. *Next Generation Java Testing: TestNG and Advanced Concepts.* Addison-Wesley, 2007.

[2] Arie van Deursen, Leon Moonen, Alex van den Bergh, and Gerard Kok. Refactoring test code. In M. Marchesi, editor, *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes (XP2001)*, pages 92–95. University of Cagliari, 2001.

[3] M. Fewster and D. Graham. *Software Test Automation*, chapter 7: Building maintainable tests. ACM Press, 1999.

[4] Markus Gaelli. *Modeling Examples to Test and Understand Software.* PhD thesis, University of Berne, November 2006.

[5] Markus Gaelli, Oscar Nierstrasz, and Stéphane Ducasse. One-method commands: Linking methods and their tests. In *OOPSLA Workshop on Revival of Dynamic Languages*, October 2004.

[6] Adrian Kuhn, Bart Van Rompaey, Lea Haensenberger, Oscar Nierstrasz, Serge Demeyer, Markus Gaelli, and Koenraad Van Leemput. Jexample: Exploiting dependencies between tests to improve defect localization. In *Proceedings of the 9nd International Conference on Extreme Programming and Flexible Processes (XP2008)*, page submitted, 2008.

[7] Gerarde Meszaros. *XUnit Test Patterns - Refactoring Test Code.* Addison Wesley, jun 2007.

[8] I. Moore. Jester — a JUnit test tester. In M. Marchesi, editor, *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes (XP2001).* University of Cagliari, 2001.

[9] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42, 1976.

[10] T. McCabe. A measure of complexity. *IEEE Transactions on Software Engineering*, 2(4):308–320, Dec. 1976.

[11] K. Beck and E. Gamma. Test infected: Programmers love writing tests. *Java Report*, 3(7):51–56, 1998.

[12] M. Lanza and S. Ducasse. Polymetric views—a lightweight visual approach to reverse engineering. *Transactions on Software Engineering (TSE)*, 29(9):782–795, Sept. 2003.

A Quick Start

A.1 Install JExample

1. Install eclipse: eclipse can be downloaded from `http://www.eclipse.org`.
2. Create a new Java Project: click on `File -> New -> Project -> Java Project`, insert a name for your project and choose at least Java 1.5 to build your project.
3. Include JUnit in your build path: right click on your project and select Properties. Go to `Java Build Path -> Libraries` and click `Add Library...`. Choose `JUnit`. On the next page choose `JUnit 4` and click `Finish`.
4. Include JExample in your build path: download JExample from `http://scg.unibe.ch/Resources/JExample/`. In eclipse right click on your project and select Properties. Go to `Java Build Path -> Libraries` and click `Add External JARs...` and select the downloaded JExample.jar.

A.2 Write unit tests with JExample

First we create a new JUnit 4 Test Case for Java's stack implementation. To run the test with JExample, the class needs to be annotated with `@RunWith` as follows:

```
@RunWith(JExampleRunner.class)
public class StackTest {

    .....

}
```

Next we create our first test method. Like in conventional JUnit test cases, a test method needs to be annotated with `@Test`. This method will be the root of our dependency hierarchy, and thus, create a Stack instance and return it.

```
@Test
public Stack<Integer> testEmptyStack() {
        Stack<Integer> aStack = new Stack<Integer>();
        assertTrue(aStack.isEmpty());

        return aStack;
}
```

Note that test methods in JExample can return objects. A test method depending on `testEmptyStack()` is `testPush()`.

```
@Test
@Depends("testEmptyStack")
public Stack<Integer> testPush(Stack<Integer> aStack) {
        aStack.push(2);
        assertFalse(aStack.isEmpty());

        return aStack;
}
```

`testPush()` takes a Stack as an argument in order to use the Stack returned by its provider `testEmptyStack()`. Finally, we write a method to test `Stack`'s pop method.

```
@Test
@Depends("testPush(java.util.Stack)")
public void testPop(Stack<Integer> aStack) {
        int element = aStack.pop();
        assertEquals(2, element);
        assertTrue(aStack.isEmpty());
}
```

Note that when depending on a test method that takes arguments the fully qualified class name of the argument needs to be specified too.

B Javadoc

The full documentation of JExample is available at `http://scg.unibe.ch/Resources/JExample/`.

C Ullman Original

Listing 13: TestUllman.java

```
public class TestUllman {
  private Graph g;
  private Graph gi;
  private Graph gj;

  @Before
  public void setUp() {
    g = new DirectedSparseGraph();
    gi = new DirectedSparseGraph();
    gj = new DirectedSparseGraph();
  }

  /*
   * Test method for
   * 'islab.graphclassifier.algorithms.graph.Ullman.areIsomorph(Graph, Graph)'
   */
  @Test
  public void testAreIsomorph() {
    // reflexive, symmetric, transitive, consistent
    // for empty graphs

    // reflexive
    assertTrue(Ullman.areIsomorph(g, g));
    assertTrue(Ullman.areIsomorph(gi, gi));

    // symmetric
    assertTrue(Ullman.areIsomorph(g, gi));
    assertTrue(Ullman.areIsomorph(gi, g));

    Vertex v1 = g.addVertex(new DirectedSparseVertex());

    assertFalse(Ullman.areIsomorph(g, gi));
    assertFalse(Ullman.areIsomorph(gi, g));

    Vertex v2 = g.addVertex(new DirectedSparseVertex());
    Vertex v3 = g.addVertex(new DirectedSparseVertex());

    g.addEdge(new DirectedSparseEdge(v1, v2));
    g.addEdge(new DirectedSparseEdge(v2, v3));
    g.addEdge(new DirectedSparseEdge(v3, v1));

    // reflexive
    assertTrue(Ullman.areIsomorph(g, g));

    Vertex vA = gi.addVertex(new DirectedSparseVertex());
    Vertex vB = gi.addVertex(new DirectedSparseVertex());
    Vertex vC = gi.addVertex(new DirectedSparseVertex());
    gi.addEdge(new DirectedSparseEdge(vA, vB));
    gi.addEdge(new DirectedSparseEdge(vB, vC));
    gi.addEdge(new DirectedSparseEdge(vC, vA));

    // reflexive
    assertTrue(Ullman.areIsomorph(gi, gi));

    // symmetric
    assertTrue(Ullman.areIsomorph(g, gi));
    assertTrue(Ullman.areIsomorph(gi, g));
```

```
Vertex vX = gj.addVertex(new DirectedSparseVertex());
Vertex vY = gj.addVertex(new DirectedSparseVertex());
Vertex vZ = gj.addVertex(new DirectedSparseVertex());
gj.addEdge(new DirectedSparseEdge(vX, vY));
gj.addEdge(new DirectedSparseEdge(vY, vZ));
gj.addEdge(new DirectedSparseEdge(vZ, vX));

// transitive
assertTrue(Ullman.areIsomorph(g, gi));
assertTrue(Ullman.areIsomorph(gi, gj));
assertTrue(Ullman.areIsomorph(gj, g));

// self-edges
g.addEdge(new DirectedSparseEdge(v1, v1));
gi.addEdge(new DirectedSparseEdge(vC, vC));

assertTrue(Ullman.areIsomorph(g, gi));
assertTrue(Ullman.areIsomorph(gi, g));

assertFalse(Ullman.areIsomorph(g, gj));
assertFalse(Ullman.areIsomorph(gj, g));

// add an edge between nodes --> degree() will become the same

g.addEdge(new DirectedSparseEdge(v2, v2));
gj.addEdge(new DirectedSparseEdge(vY, vX));
// should still fail
assertFalse(Ullman.areIsomorph(g, gj));
assertFalse(Ullman.areIsomorph(gj, g));

// restore
g.removeEdge(v1.findEdge(v1));
g.removeEdge(v2.findEdge(v2));
gi.removeEdge(vC.findEdge(vC));
gj.removeEdge(vY.findEdge(vX));
assertTrue(Ullman.areIsomorph(g, gi));
assertTrue(Ullman.areIsomorph(gi, gj));
assertTrue(Ullman.areIsomorph(gj, g));

// add Edge
gj.addEdge(new DirectedSparseEdge(vX, vZ));
assertFalse(Ullman.areIsomorph(g, gj));
assertFalse(Ullman.areIsomorph(gj, g));
// restore
gj.removeEdge(vX.findEdge(vZ));
assertTrue(Ullman.areIsomorph(g, gj));
assertTrue(Ullman.areIsomorph(gj, g));

// remove Edge
gj.removeEdge(vY.findEdge(vZ));
assertFalse(Ullman.areIsomorph(g, gj));
assertFalse(Ullman.areIsomorph(gj, g));
// restore
gj.addEdge(new DirectedSparseEdge(vY, vZ));
assertTrue(Ullman.areIsomorph(g, gj));
assertTrue(Ullman.areIsomorph(gj, g));

// rewire Edge
gj.removeEdge(vX.findEdge(vY));
gj.addEdge(new DirectedSparseEdge(vX, vZ));
assertFalse(Ullman.areIsomorph(g, gj));
assertFalse(Ullman.areIsomorph(gj, g));
```

```
// restore
gj.removeEdge(vX.findEdge(vZ));
gj.addEdge(new DirectedSparseEdge(vX, vY));
assertTrue(Ullman.areIsomorph(g, gj));
assertTrue(Ullman.areIsomorph(gj, g));

// add Vertex
Vertex vAdd = gj.addVertex(new DirectedSparseVertex());
assertFalse(Ullman.areIsomorph(g, gj));
assertFalse(Ullman.areIsomorph(gj, g));
// restore
gj.removeVertex(vAdd);
assertTrue(Ullman.areIsomorph(g, gj));
assertTrue(Ullman.areIsomorph(gj, g));

// add Vertex & Edge
gj.addVertex(vAdd);
gj.addEdge(new DirectedSparseEdge(vZ, vAdd));
assertFalse(Ullman.areIsomorph(g, gj));
assertFalse(Ullman.areIsomorph(gj, g));
// restore
gj.removeVertex(vAdd);
assertTrue(Ullman.areIsomorph(g, gj));
assertTrue(Ullman.areIsomorph(gj, g));

// remove Vertex
Set<Edge> vZedges = vZ.getIncidentEdges();
gj.removeVertex(vZ);
assertFalse(Ullman.areIsomorph(g, gj));
assertFalse(Ullman.areIsomorph(gj, g));
// restore
gj.addVertex(vZ);
for (Edge edge : vZedges) {
  gj.addEdge(edge);
}
assertTrue(Ullman.areIsomorph(g, gj));
assertTrue(Ullman.areIsomorph(gj, g));

// test special case with self-loop
Graph sg = new DirectedSparseGraph();
Vertex s1 = sg.addVertex(new DirectedSparseVertex());
Vertex s2 = sg.addVertex(new DirectedSparseVertex());
Vertex s3 = sg.addVertex(new DirectedSparseVertex());
Vertex s4 = sg.addVertex(new DirectedSparseVertex());
Vertex s5 = sg.addVertex(new DirectedSparseVertex());

sg.addEdge(new DirectedSparseEdge(s1, s2));
sg.addEdge(new DirectedSparseEdge(s3, s3));
sg.addEdge(new DirectedSparseEdge(s3, s4));
sg.addEdge(new DirectedSparseEdge(s3, s5));

Graph sg2 = new DirectedSparseGraph();
Vertex ss1 = sg2.addVertex(new DirectedSparseVertex());
Vertex ss2 = sg2.addVertex(new DirectedSparseVertex());
Vertex ss3 = sg2.addVertex(new DirectedSparseVertex());
Vertex ss4 = sg2.addVertex(new DirectedSparseVertex());
Vertex ss5 = sg2.addVertex(new DirectedSparseVertex());

sg2.addEdge(new DirectedSparseEdge(ss1, ss3));
sg2.addEdge(new DirectedSparseEdge(ss3, ss2));
sg2.addEdge(new DirectedSparseEdge(ss3, ss4));
sg2.addEdge(new DirectedSparseEdge(ss3, ss5));
```

```
  assertFalse(Ullman.areIsomorph(sg, sg2));
  assertFalse(Ullman.areIsomorph(sg2, sg));

}

@Test
public void simpleSubgraphIsomorphism() {

  // one edge
  Vertex v1 = g.addVertex(new DirectedSparseVertex());
  Vertex v2 = g.addVertex(new DirectedSparseVertex());
  g.addEdge(new DirectedSparseEdge(v1, v2));

  // single Edge
  Vertex vX = gj.addVertex(new DirectedSparseVertex());
  Vertex vY = gj.addVertex(new DirectedSparseVertex());
  gj.addEdge(new DirectedSparseEdge(vX, vY));

  List subgraphs = Ullman.subgraphIsomorphism(gj, g);

  assertEquals(1, subgraphs.size());
  subgraphs = Ullman.subgraphIsomorphism(g, gj);

  assertEquals(1, subgraphs.size());

  // add another vertex
  Vertex v3 = g.addVertex(new DirectedSparseVertex());

  subgraphs = Ullman.subgraphIsomorphism(gj, g);

  assertEquals(1, subgraphs.size());

  try {
    subgraphs = Ullman.subgraphIsomorphism(g, gj);
    fail("Expected AssertionError - assertions enabled?");
  } catch (AssertionError e) {
    assertEquals(e.getMessage(),
        "model graph must have less or equal nr of vertices as input graph");

  }

  // remove vertex and add self-loop
  g.removeVertex(v3);
  g.addEdge(new DirectedSparseEdge(v1, v1));

  subgraphs = Ullman.subgraphIsomorphism(gj, g);

  assertEquals(1, subgraphs.size());
  subgraphs = Ullman.subgraphIsomorphism(g, gj);

  assertEquals(0, subgraphs.size());

}

@Test
public void simpleSubgraphIsomorphism2() {

  // circle
  Vertex v1 = g.addVertex(new DirectedSparseVertex());
  Vertex v2 = g.addVertex(new DirectedSparseVertex());
  Vertex v3 = g.addVertex(new DirectedSparseVertex());
Vertex v4 = g.addVertex(new DirectedSparseVertex());
g.addEdge(new DirectedSparseEdge(v1, v2));
g.addEdge(new DirectedSparseEdge(v2, v3));
g.addEdge(new DirectedSparseEdge(v3, v4));
g.addEdge(new DirectedSparseEdge(v4, v1));

// 2-Chain
Vertex vX = gj.addVertex(new DirectedSparseVertex());
Vertex vY = gj.addVertex(new DirectedSparseVertex());
Vertex vZ = gj.addVertex(new DirectedSparseVertex());
gj.addEdge(new DirectedSparseEdge(vX, vY));
gj.addEdge(new DirectedSparseEdge(vY, vZ));

List subgraphs = Ullman.subgraphIsomorphism(gj, g);

assertEquals(4, subgraphs.size());

try {
  subgraphs = Ullman.subgraphIsomorphism(g, gj);
  fail("Expected AssertionError - assertions enabled?");
} catch (AssertionError e) {
  assertEquals(e.getMessage(),
      "model graph must have less or equal nr of vertices as input graph");

}

// System.out.println(subgraphs.size());
// System.out.println(subgraphs);
// assertEquals(0, subgraphs.size());

// remove edge
g.removeEdge(v4.findEdge(v1));

subgraphs = Ullman.subgraphIsomorphism(gj, g);

assertEquals(2, subgraphs.size());
try {
  subgraphs = Ullman.subgraphIsomorphism(g, gj);
  fail("Expected AssertionError - assertions enabled?");
} catch (AssertionError e) {
  assertEquals(e.getMessage(),
      "model graph must have less or equal nr of vertices as input graph");

}

// System.out.println(subgraphs.size());
// System.out.println(subgraphs);
// assertEquals(0, subgraphs.size());

// self-loop
g.addEdge(new DirectedSparseEdge(v4, v4));

subgraphs = Ullman.subgraphIsomorphism(gj, g);

assertEquals(2, subgraphs.size());
try {
  subgraphs = Ullman.subgraphIsomorphism(g, gj);
  fail("Expected AssertionError - assertions enabled?");
} catch (AssertionError e) {
  assertEquals(e.getMessage(),
      "model graph must have less or equal nr of vertices as input graph");
  }

}

@Test
public void subgraphIsomorphism() {
  // for empty graphs

  // TODO

  // circle
  Vertex v1 = g.addVertex(new DirectedSparseVertex());
  Vertex v2 = g.addVertex(new DirectedSparseVertex());
  Vertex v3 = g.addVertex(new DirectedSparseVertex());
  Vertex v4 = g.addVertex(new DirectedSparseVertex());
  g.addEdge(new DirectedSparseEdge(v1, v2));
  g.addEdge(new DirectedSparseEdge(v2, v3));
  g.addEdge(new DirectedSparseEdge(v3, v4));
  g.addEdge(new DirectedSparseEdge(v4, v1));

  // circle with extra reflexive edge
  Vertex vA = gi.addVertex(new DirectedSparseVertex());
  Vertex vB = gi.addVertex(new DirectedSparseVertex());
  Vertex vC = gi.addVertex(new DirectedSparseVertex());
  Vertex vD = gi.addVertex(new DirectedSparseVertex());
  gi.addEdge(new DirectedSparseEdge(vA, vB));
  gi.addEdge(new DirectedSparseEdge(vB, vC));
  gi.addEdge(new DirectedSparseEdge(vC, vD));
  gi.addEdge(new DirectedSparseEdge(vD, vA));
  gi.addEdge(new DirectedSparseEdge(vA, vD));

  List subgraphs = Ullman.subgraphIsomorphism(g, gi);

  assertEquals(4, subgraphs.size());
  subgraphs = Ullman.subgraphIsomorphism(gi, g);

  assertEquals(0, subgraphs.size());

  // single Vertex
  Vertex vX = gj.addVertex(new DirectedSparseVertex());

  subgraphs = Ullman.subgraphIsomorphism(g, gi);

  assertEquals(4, subgraphs.size());
  subgraphs = Ullman.subgraphIsomorphism(gi, g);

  assertEquals(0, subgraphs.size());

  subgraphs = Ullman.subgraphIsomorphism(gj, g);

  assertEquals(4, subgraphs.size());
  subgraphs = Ullman.subgraphIsomorphism(gj, gi);

  assertEquals(4, subgraphs.size());

  // single Edge
  Vertex vY = gj.addVertex(new DirectedSparseVertex());
  gj.addEdge(new DirectedSparseEdge(vX, vY));

  subgraphs = Ullman.subgraphIsomorphism(gj, g);

  assertEquals(4, subgraphs.size());
  subgraphs = Ullman.subgraphIsomorphism(gj, gi);

  assertEquals(5, subgraphs.size());

}

@Test
public void subgraphIsomorphismSelfLoops() {

  // circle
  Vertex v1 = g.addVertex(new DirectedSparseVertex());
  Vertex v2 = g.addVertex(new DirectedSparseVertex());
  Vertex v3 = g.addVertex(new DirectedSparseVertex());
  Vertex v4 = g.addVertex(new DirectedSparseVertex());
  g.addEdge(new DirectedSparseEdge(v1, v2));
  g.addEdge(new DirectedSparseEdge(v2, v3));
  g.addEdge(new DirectedSparseEdge(v3, v4));
  g.addEdge(new DirectedSparseEdge(v4, v1));

  // circle with extra self-loop and reflexive edge
  Vertex vA = gi.addVertex(new DirectedSparseVertex());
  Vertex vB = gi.addVertex(new DirectedSparseVertex());
  Vertex vC = gi.addVertex(new DirectedSparseVertex());
  Vertex vD = gi.addVertex(new DirectedSparseVertex());
  gi.addEdge(new DirectedSparseEdge(vA, vB));
  gi.addEdge(new DirectedSparseEdge(vB, vC));
  gi.addEdge(new DirectedSparseEdge(vC, vD));
  gi.addEdge(new DirectedSparseEdge(vD, vA));
  gi.addEdge(new DirectedSparseEdge(vA, vD));

  gi.addEdge(new DirectedSparseEdge(vA, vA));
  gi.addEdge(new DirectedSparseEdge(vC, vC));

  List subgraphs = Ullman.subgraphIsomorphism(g, gi);

  assertEquals(4, subgraphs.size());
  subgraphs = Ullman.subgraphIsomorphism(gi, g);

  assertEquals(0, subgraphs.size());

  // single vertex
  Vertex vX = gj.addVertex(new DirectedSparseVertex());

  subgraphs = Ullman.subgraphIsomorphism(g, gi);

  assertEquals(4, subgraphs.size());
  subgraphs = Ullman.subgraphIsomorphism(gi, g);

  assertEquals(0, subgraphs.size());
  subgraphs = Ullman.subgraphIsomorphism(gj, g);
  assertEquals(4, subgraphs.size());
  subgraphs = Ullman.subgraphIsomorphism(gj, gi);
  assertEquals(4, subgraphs.size());

  // single self-loop
  gj.addEdge(new DirectedSparseEdge(vX, vX));
  subgraphs = Ullman.subgraphIsomorphism(gj, g);
  assertEquals(0, subgraphs.size());
  subgraphs = Ullman.subgraphIsomorphism(gj, gi);
  assertEquals(2, subgraphs.size());
}
```

```
  @Test
  public void testVertexPair() {
    Vertex v1 = new DirectedSparseVertex();
    Vertex v2 = new DirectedSparseVertex();
    Vertex v3 = new DirectedSparseVertex();

    VertexPair p1 = new VertexPair(v1, v2);
    VertexPair p2 = new VertexPair(v1, v2);
    VertexPair p3 = new VertexPair(v1, v3);

    assertEquals(p1, p2);
    // reflexive
    assertTrue(p1.equals(p2));
    assertTrue(p2.equals(p1));

    assertFalse(p1.equals(p3));
    assertFalse(p3.equals(p1));
  }
}
```

D Ullman refactored

Listing 14: IsomorphTestUllman.java

```
@RunWith(ComposedTestRunner.class)
public class IsomorphTestUllman {

  public IsomorphTestUllman() {

  }

  @Test
  public DirectedSparseGraph testEmptyGraphIsIsomorphToItself() {
    DirectedSparseGraph g = new DirectedSparseGraph();

    assertTrue(Ullman.areIsomorph(g, g));
    return g;
  }

  @Test
  @Depends("testEmptyGraphIsIsomorphToItself")
  public void testEmptyGraphsAreSymmetricallyIsomorph(DirectedSparseGraph g) {
    DirectedSparseGraph gi = new DirectedSparseGraph();

    assertTrue(Ullman.areIsomorph(g, gi));
    assertTrue(Ullman.areIsomorph(gi, g));
  }

  @Test
  @Depends("testEmptyGraphIsIsomorphToItself")
  public DirectedSparseGraph testAddVertexNotIsomorph(DirectedSparseGraph g) {
    DirectedSparseGraph gi = (DirectedSparseGraph) g.copy();
    gi.addVertex(new DirectedSparseVertex());

    assertFalse(Ullman.areIsomorph(g, gi));
    assertFalse(Ullman.areIsomorph(gi, g));

    return gi;
  }
```

```
@Test
public DirectedSparseGraph testIsomorphWithItself() {
  DirectedSparseGraph gj = new DirectedSparseGraph();
  Vertex vX = gj.addVertex(new DirectedSparseVertex());
  Vertex vY = gj.addVertex(new DirectedSparseVertex());
  Vertex vZ = gj.addVertex(new DirectedSparseVertex());
  gj.addEdge(new DirectedSparseEdge(vX, vY));
  gj.addEdge(new DirectedSparseEdge(vY, vZ));
  gj.addEdge(new DirectedSparseEdge(vZ, vX));

  assertTrue(Ullman.areIsomorph(gj, gj));

  return gj;
}

@Test
@Depends("testIsomorphWithItself")
public DirectedSparseGraph testGraphWithSelfEdgeIsIsomorphToItself(
    DirectedSparseGraph gj) {
  DirectedSparseGraph gi = (DirectedSparseGraph) gj.copy();

  for (Object vertex : gi.getVertices()) {
    gi
        .addEdge(new DirectedSparseEdge((Vertex) vertex,
            (Vertex) vertex));
    break;
  }

  assertTrue(Ullman.areIsomorph(gi, gi));

  return gi;
}

@Test
@Depends("testEmptyGraphIsIsomorphToItself;testAddVertexNotIsomorph(edu.uci.ics.
     jung.graph.impl.DirectedSparseGraph)")
public void testRemoveVertexIsomorphAgain(DirectedSparseGraph g,
    DirectedSparseGraph gi) {
  gi = (DirectedSparseGraph) gi.copy();
  gi.removeAllVertices();

  assertTrue(Ullman.areIsomorph(g, gi));
  assertTrue(Ullman.areIsomorph(gi, g));
}

@Test
@Depends("testEmptyGraphIsIsomorphToItself;testIsomorphWithItself")
public DirectedSparseGraph testSymmetricallyIsomorph(DirectedSparseGraph g,
    DirectedSparseGraph gj) {
  g = (DirectedSparseGraph) g.copy();

  Vertex v1 = g.addVertex(new DirectedSparseVertex());
  Vertex v2 = g.addVertex(new DirectedSparseVertex());
  Vertex v3 = g.addVertex(new DirectedSparseVertex());

  g.addEdge(new DirectedSparseEdge(v1, v2));
  g.addEdge(new DirectedSparseEdge(v2, v3));
  g.addEdge(new DirectedSparseEdge(v3, v1));

  assertTrue(Ullman.areIsomorph(g, gj));
  assertTrue(Ullman.areIsomorph(gj, g));
  return g;
}

@Test
@Depends("testSymmetricallyIsomorph(edu.uci.ics.jung.graph.impl.
     DirectedSparseGraph,edu.uci.ics.jung.graph.impl.DirectedSparseGraph);"
    + "testGraphWithSelfEdgeIsIsomorphToItself(edu.uci.ics.jung.graph.impl.
        DirectedSparseGraph)")
public void testNotIsomorphWithOneSelfEdge(DirectedSparseGraph g,
    DirectedSparseGraph gi) {

  assertFalse(Ullman.areIsomorph(g, gi));
  assertFalse(Ullman.areIsomorph(gi, g));
}

@Test
@Depends("testSymmetricallyIsomorph(edu.uci.ics.jung.graph.impl.
     DirectedSparseGraph,edu.uci.ics.jung.graph.impl.DirectedSparseGraph);"
    + "testGraphWithSelfEdgeIsIsomorphToItself(edu.uci.ics.jung.graph.impl.
        DirectedSparseGraph)")
public DirectedSparseGraph testNotIsomorphWithOneSelfEdgeAndSameDegree(
    DirectedSparseGraph g, DirectedSparseGraph gi) {
  g = (DirectedSparseGraph) g.copy();

  assertFalse(g.getEdges().size() == gi.getEdges().size());

  for (Object edge : g.getEdges()) {
    DirectedSparseEdge anEdge = (DirectedSparseEdge) edge;
    g.addEdge(new DirectedSparseEdge(anEdge.getDest(), anEdge
        .getSource()));
    break;
  }

  assertEquals(g.getEdges().size(), gi.getEdges().size());

  assertFalse(Ullman.areIsomorph(g, gi));
  assertFalse(Ullman.areIsomorph(gi, g));

  return g;
}

@Test
@Depends("testSymmetricallyIsomorph(edu.uci.ics.jung.graph.impl.
     DirectedSparseGraph,edu.uci.ics.jung.graph.impl.DirectedSparseGraph);"
    + "testGraphWithSelfEdgeIsIsomorphToItself(edu.uci.ics.jung.graph.impl.
        DirectedSparseGraph)")
public DirectedSparseGraph testIsomorphWithBothSelfEdge(DirectedSparseGraph g,
    DirectedSparseGraph gi) {
  g = (DirectedSparseGraph) g.copy();

  for (Object vertex : g.getVertices()) {
    g.addEdge(new DirectedSparseEdge((Vertex) vertex, (Vertex) vertex));
    break;
  }

  assertTrue(Ullman.areIsomorph(g, gi));
  assertTrue(Ullman.areIsomorph(gi, g));

  return g;
}
```

```
@Test
@Depends("testIsomorphWithBothSelfEdge(edu.uci.ics.jung.graph.impl.
     DirectedSparseGraph,edu.uci.ics.jung.graph.impl.DirectedSparseGraph);"
    + "testGraphWithSelfEdgeIsIsomorphToItself(edu.uci.ics.jung.graph.impl.
         DirectedSparseGraph);" +
    "testSymmetricallyIsomorph(edu.uci.ics.jung.graph.impl.DirectedSparseGraph,edu.
         uci.ics.jung.graph.impl.DirectedSparseGraph)")
public void testRemoveSelfEdgesIsomorphAgain(DirectedSparseGraph g,
     DirectedSparseGraph gi, DirectedSparseGraph gj){
  g = (DirectedSparseGraph) g.copy();
  gi = (DirectedSparseGraph) gi.copy();

  // remove self-edges
  DirectedSparseEdge anEdge = null;
  for (Object edge : g.getEdges()) {
    anEdge = (DirectedSparseEdge) edge;
    if(anEdge.getSource().equals(anEdge.getDest())){
      break;
    }
  }
  g.removeEdge(anEdge);
  for (Object edge : gi.getEdges()) {
    anEdge = (DirectedSparseEdge) edge;
    if(anEdge.getSource().equals(anEdge.getDest())){
      break;
    }
  }
  gi.removeEdge(anEdge);

  assertTrue(Ullman.areIsomorph(g, gi));
  assertTrue(Ullman.areIsomorph(gi, gj));
  assertTrue(Ullman.areIsomorph(gj, g));
}

@Test
@Depends("testSymmetricallyIsomorph(edu.uci.ics.jung.graph.impl.
     DirectedSparseGraph,edu.uci.ics.jung.graph.impl.DirectedSparseGraph);"
    + "testIsomorphWithItself")
public void testAddRemoveEdge(DirectedSparseGraph g, DirectedSparseGraph gi){
  g = (DirectedSparseGraph) g.copy();

  DirectedSparseEdge edge = (DirectedSparseEdge) g.getEdges().iterator().next();
  Edge edgeToAdd = new DirectedSparseEdge(edge.getDest(), edge.getSource());

  g.addEdge(edgeToAdd);
  assertFalse(Ullman.areIsomorph(g, gi));
  assertFalse(Ullman.areIsomorph(gi, g));

  g.removeEdge(edgeToAdd);
  assertTrue(Ullman.areIsomorph(g, gi));
  assertTrue(Ullman.areIsomorph(gi, g));
}

@Test
@Depends("testSymmetricallyIsomorph(edu.uci.ics.jung.graph.impl.
     DirectedSparseGraph,edu.uci.ics.jung.graph.impl.DirectedSparseGraph);"
    + "testIsomorphWithItself")
public void testRemoveRestoreEdge(DirectedSparseGraph g, DirectedSparseGraph gi){
  g = (DirectedSparseGraph) g.copy();

  DirectedSparseEdge edgeToRemove = (DirectedSparseEdge) g.getEdges().iterator().
       next();
  g.removeEdge(edgeToRemove);
  assertFalse(Ullman.areIsomorph(g, gi));
  assertFalse(Ullman.areIsomorph(gi, g));

  g.addEdge(edgeToRemove);
  assertTrue(Ullman.areIsomorph(g, gi));
  assertTrue(Ullman.areIsomorph(gi, g));
}

@Test
@Depends("testSymmetricallyIsomorph(edu.uci.ics.jung.graph.impl.
     DirectedSparseGraph,edu.uci.ics.jung.graph.impl.DirectedSparseGraph);"
    + "testIsomorphWithItself")
public void testRewireEdge(DirectedSparseGraph g, DirectedSparseGraph gj) {
  g = (DirectedSparseGraph) g.copy();

  Vertex source = null, oldDest = null, newDest = null;

  for (Object edge : g.getEdges()) {
    if (source == null) {
      source = ((DirectedSparseEdge) edge).getSource();
      oldDest = ((DirectedSparseEdge) edge).getDest();
    } else {
      newDest = ((DirectedSparseEdge) edge).getDest();
      break;
    }
  }

  g.removeEdge(source.findEdge(oldDest));
  g.addEdge(new DirectedSparseEdge(source, newDest));

  assertFalse(Ullman.areIsomorph(g, gj));
  assertFalse(Ullman.areIsomorph(gj, g));

  g.removeEdge(source.findEdge(newDest));
  g.addEdge(new DirectedSparseEdge(source, oldDest));

  assertTrue(Ullman.areIsomorph(g, gj));
  assertTrue(Ullman.areIsomorph(gj, g));
}

@Test
@Depends("testSymmetricallyIsomorph(edu.uci.ics.jung.graph.impl.
     DirectedSparseGraph,edu.uci.ics.jung.graph.impl.DirectedSparseGraph);"
    + "testIsomorphWithItself")
public void testAddRemoveVertex(DirectedSparseGraph g,
    DirectedSparseGraph gj) {
  g = (DirectedSparseGraph) g.copy();

  Vertex added = g.addVertex(new DirectedSparseVertex());

  assertFalse(Ullman.areIsomorph(g, gj));
  assertFalse(Ullman.areIsomorph(gj, g));

  g.removeVertex(added);

  assertTrue(Ullman.areIsomorph(g, gj));
  assertTrue(Ullman.areIsomorph(gj, g));
}

@Test
```

```
  @Depends("testSymmetricallyIsomorph(edu.uci.ics.jung.graph.impl.
      DirectedSparseGraph,edu.uci.ics.jung.graph.impl.DirectedSparseGraph);"
      + "testIsomorphWithItself")
  public void testAddRemoveVertexAndEdge(DirectedSparseGraph g,
      DirectedSparseGraph gj) {
    g = (DirectedSparseGraph) g.copy();

    Vertex added = g.addVertex(new DirectedSparseVertex());
    Vertex existing = (Vertex) g.getVertices().iterator().next();

    g.addEdge(new DirectedSparseEdge(existing, added));

    assertFalse(Ullman.areIsomorph(g, gj));
    assertFalse(Ullman.areIsomorph(gj, g));

    g.removeVertex(added);

    assertTrue(Ullman.areIsomorph(g, gj));
    assertTrue(Ullman.areIsomorph(gj, g));
  }

  @Test
  @Depends("testSymmetricallyIsomorph(edu.uci.ics.jung.graph.impl.
      DirectedSparseGraph,edu.uci.ics.jung.graph.impl.DirectedSparseGraph);"
      + "testIsomorphWithItself")
  public void testRemoveRestoreVertex(DirectedSparseGraph g,
      DirectedSparseGraph gj) {
    g = (DirectedSparseGraph) g.copy();

    Vertex toRemove = (Vertex) g.getVertices().iterator().next();
    Set<DirectedSparseEdge> toRestore = toRemove.getIncidentEdges();

    g.removeVertex(toRemove);

    assertFalse(Ullman.areIsomorph(g, gj));
    assertFalse(Ullman.areIsomorph(gj, g));

    g.addVertex(toRemove);
    for (DirectedSparseEdge directedSparseEdge : toRestore) {
      g.addEdge(directedSparseEdge);
    }

    assertTrue(Ullman.areIsomorph(g, gj));
    assertTrue(Ullman.areIsomorph(gj, g));
  }
}
```

Listing 15: SimpleSubgraphTest.java

```java
@RunWith(ComposedTestRunner.class)
public class SimpleSubgraphTest {
  public SimpleSubgraphTest() {
  }

  @Test
  public DirectedSparseGraph testChainSubgraphIsomorphism() {
    DirectedSparseGraph gj = new DirectedSparseGraph();

    Vertex v1 = gj.addVertex(new DirectedSparseVertex());
    Vertex v2 = gj.addVertex(new DirectedSparseVertex());
    Vertex v3 = gj.addVertex(new DirectedSparseVertex());

    gj.addEdge(new DirectedSparseEdge(v1, v2));
    gj.addEdge(new DirectedSparseEdge(v2, v3));

    DirectedSparseGraph g = (DirectedSparseGraph) gj.copy();

    List<Set<VertexPair>> subgraphs = Ullman.subgraphIsomorphism(gj, g);
    assertEquals(1, subgraphs.size());
    subgraphs = Ullman.subgraphIsomorphism(g, gj);
    assertEquals(1, subgraphs.size());

    return g;
  }

  @Test
  @Depends("testChainSubgraphIsomorphism")
  public DirectedSparseGraph testInputMoreVertices(DirectedSparseGraph g) {
    g = (DirectedSparseGraph) g.copy();
    DirectedSparseGraph gj = (DirectedSparseGraph) g.copy();

    g.addVertex(new DirectedSparseVertex());

    List<Set<VertexPair>> subgraphs = Ullman.subgraphIsomorphism(gj, g);
    assertEquals(1, subgraphs.size());

    return g;
  }

  @Test
  public DirectedSparseGraph testCircleSubgraphIsomorphisms() {
    DirectedSparseGraph gj = new DirectedSparseGraph();

    Vertex v1 = gj.addVertex(new DirectedSparseVertex());
    Vertex v2 = gj.addVertex(new DirectedSparseVertex());
    Vertex v3 = gj.addVertex(new DirectedSparseVertex());
    Vertex v4 = gj.addVertex(new DirectedSparseVertex());

    gj.addEdge(new DirectedSparseEdge(v1, v2));
    gj.addEdge(new DirectedSparseEdge(v2, v3));
    gj.addEdge(new DirectedSparseEdge(v3, v4));
    gj.addEdge(new DirectedSparseEdge(v4, v1));

    DirectedSparseGraph g = (DirectedSparseGraph) gj.copy();

    List<Set<VertexPair>> subgraphs = Ullman.subgraphIsomorphism(gj, g);
    assertEquals(4, subgraphs.size());
    subgraphs = Ullman.subgraphIsomorphism(g, gj);
    assertEquals(4, subgraphs.size());
  return g;
}

@Depends("testInputMoreVertices(edu.uci.ics.jung.graph.impl.DirectedSparseGraph);
     testChainSubgraphIsomorphism")
public void testModelMoreVertices(DirectedSparseGraph g,
    DirectedSparseGraph gj) {
  try {
    Ullman.subgraphIsomorphism(g, gj);
    fail("Expected AssertionError - assertions enabled?");
  } catch (AssertionError e) {
    assertEquals(e.getMessage(),
        "model graph must have less or equal nr of vertices as input graph");
  }
}

@Test
@Depends("testChainSubgraphIsomorphism")
public DirectedSparseGraph testInputSelfEdgeSubgraphIsomorphism(
    DirectedSparseGraph g) {
  DirectedSparseGraph gj = (DirectedSparseGraph) g.copy();
  g = (DirectedSparseGraph) g.copy();

  for (Object vertex : g.getVertices()) {
    g.addEdge(new DirectedSparseEdge((Vertex) vertex, (Vertex) vertex));
    break;
  }

  List<Set<VertexPair>> subgraph = Ullman.subgraphIsomorphism(gj, g);
  assertEquals(1, subgraph.size());

  return g;
}

@Test
@Depends("testInputSelfEdgeSubgraphIsomorphism(edu.uci.ics.jung.graph.impl.
     DirectedSparseGraph);testChainSubgraphIsomorphism")
public void testModelSelfEdgeNoSubgraphIsomorphism(DirectedSparseGraph g,
    DirectedSparseGraph gj) {
  List<Set<VertexPair>> subgraphs = Ullman.subgraphIsomorphism(g, gj);
  assertEquals(0, subgraphs.size());
}

@Test
@Depends("testChainSubgraphIsomorphism;testCircleSubgraphIsomorphisms")
public void testInputCircleModelChainSubgraphIsomorphism(
    DirectedSparseGraph gj, DirectedSparseGraph g) {
  List<Set<VertexPair>> subgraphs = Ullman.subgraphIsomorphism(gj, g);
  assertEquals(4, subgraphs.size());
}

@Depends("testChainSubgraphIsomorphism;testCircleSubgraphIsomorphisms")
public void testModelCircleInputChainNoSubgraphIsomorphism(
    DirectedSparseGraph gj, DirectedSparseGraph g) {
  try {
    Ullman.subgraphIsomorphism(g, gj);
    fail("Expected AssertionError - assertions enabled?");
  } catch (AssertionError e) {
    assertEquals(e.getMessage(),
        "model graph must have less or equal nr of vertices as input graph");
  }
}
```

```
@Test
@Depends("testChainSubgraphIsomorphism;testCircleSubgraphIsomorphisms")
public DirectedSparseGraph testInputLongerChainSubgraphIsomorphism(
    DirectedSparseGraph gj, DirectedSparseGraph g) {
  g = (DirectedSparseGraph) g.copy();

  g.removeAllEdges();
  Vertex lastVertex = null;
  for (Object vertex : g.getVertices()) {
    if (lastVertex != null) {
      g.addEdge(new DirectedSparseEdge(lastVertex, (Vertex) vertex));
    }
    lastVertex = (Vertex) vertex;
  }

  List<Set<VertexPair>> subgraphs = Ullman.subgraphIsomorphism(gj, g);
  assertEquals(2, subgraphs.size());

  return g;
}

@Depends("testChainSubgraphIsomorphism;testInputLongerChainSubgraphIsomorphism(edu
     .uci.ics.jung.graph.impl.DirectedSparseGraph,edu.uci.ics.jung.graph.impl.
     DirectedSparseGraph)")
public void testModelLongerChainNoSubgraphIsomorphism(
    DirectedSparseGraph gj, DirectedSparseGraph g) {
  try {
    Ullman.subgraphIsomorphism(g, gj);
    fail("Expected AssertionError - assertions enabled?");
  } catch (AssertionError e) {
    assertEquals(e.getMessage(),
        "model graph must have less or equal nr of vertices as input graph");
  }
}

@Test
@Depends("testChainSubgraphIsomorphism;testInputLongerChainSubgraphIsomorphism(edu
     .uci.ics.jung.graph.impl.DirectedSparseGraph,edu.uci.ics.jung.graph.impl.
     DirectedSparseGraph)")
public void testSelfEdgeSubgraphIsomorphism(DirectedSparseGraph gj,
    DirectedSparseGraph g) {
  g = (DirectedSparseGraph) g.copy();

  for (Object vertex : g.getVertices()) {
    g.addEdge(new DirectedSparseEdge((Vertex) vertex, (Vertex) vertex));
    break;
  }

  List<Set<VertexPair>> subgraphs = Ullman.subgraphIsomorphism(gj, g);
  assertEquals(2, subgraphs.size());
}

@Test
@Depends("testCircleSubgraphIsomorphisms")
public DirectedSparseGraph testOneCircleReflexiveEdge(DirectedSparseGraph g) {
  DirectedSparseGraph gi = (DirectedSparseGraph) g.copy();

  Edge edge = null;
  for (Object e : gi.getEdges()) {
    edge = (Edge) e;
    break;
  }
  gi.addEdge(new DirectedSparseEdge(
      ((DirectedSparseEdge) edge).getDest(),
      ((DirectedSparseEdge) edge).getSource()));

  List<Set<VertexPair>> subgraphs = Ullman.subgraphIsomorphism(g, gi);
  assertEquals(4, subgraphs.size());
  subgraphs = Ullman.subgraphIsomorphism(gi, g);
  assertEquals(0, subgraphs.size());

  return gi;
}

@Test
@Depends("testOneCircleReflexiveEdge(edu.uci.ics.jung.graph.impl.
     DirectedSparseGraph);"
    + "testCircleSubgraphIsomorphisms;"
    + "IsomorphTestUllman.testAddVertexNotIsomorph(edu.uci.ics.jung.graph.impl.
        DirectedSparseGraph)")
public void testSingleVertexVersCircle(DirectedSparseGraph gi,
    DirectedSparseGraph g, DirectedSparseGraph gj) {
  List<Set<VertexPair>> subgraphs = Ullman.subgraphIsomorphism(gj, gi);
  assertEquals(4, subgraphs.size());
  subgraphs = Ullman.subgraphIsomorphism(gj, g);
  assertEquals(4, subgraphs.size());
}

@Test
@Depends("testOneCircleReflexiveEdge(edu.uci.ics.jung.graph.impl.
     DirectedSparseGraph);"
    + "testCircleSubgraphIsomorphisms;"
    + "IsomorphTestUllman.testAddVertexNotIsomorph(edu.uci.ics.jung.graph.impl.
        DirectedSparseGraph)")
public void testSingleEdgeVersCircle(DirectedSparseGraph gi,
    DirectedSparseGraph g, DirectedSparseGraph gj) {
  gj = (DirectedSparseGraph) gj.copy();
  Vertex vert = null;
  for (Object vertex : gj.getVertices()) {
    vert = (Vertex) vertex;
  }

  Vertex v2 = gj.addVertex(new DirectedSparseVertex());
  gj.addEdge(new DirectedSparseEdge(vert, v2));

  List<Set<VertexPair>> subgraphs = Ullman.subgraphIsomorphism(gj, gi);
  assertEquals(5, subgraphs.size());
  subgraphs = Ullman.subgraphIsomorphism(gj, g);
  assertEquals(4, subgraphs.size());
}

@Test
@Depends("testOneCircleReflexiveEdge(edu.uci.ics.jung.graph.impl.
     DirectedSparseGraph);testCircleSubgraphIsomorphisms")
public DirectedSparseGraph testCircleReflexiveEdgeAndSelfLoop(
    DirectedSparseGraph gi, DirectedSparseGraph g) {

  gi = (DirectedSparseGraph) gi.copy();
  int count = 1;
  for (Object vertex : gi.getVertices()) {
    gi
        .addEdge(new DirectedSparseEdge((Vertex) vertex,
            (Vertex) vertex));
      if (count == 2) {
        break;
      }
      count++;
    }

    List<Set<VertexPair>> subgraphs = Ullman.subgraphIsomorphism(g, gi);
    assertEquals(4, subgraphs.size());
    subgraphs = Ullman.subgraphIsomorphism(gi, g);
    assertEquals(0, subgraphs.size());

    return gi;
  }

  @Test
  @Depends("testCircleReflexiveEdgeAndSelfLoop(edu.uci.ics.jung.graph.impl.
       DirectedSparseGraph,edu.uci.ics.jung.graph.impl.DirectedSparseGraph);"
      + "testCircleSubgraphIsomorphisms;"
      + "IsomorphTestUllman.testAddVertexNotIsomorph(edu.uci.ics.jung.graph.impl.
          DirectedSparseGraph)")
  public void testSingleVertexVersCircle2(DirectedSparseGraph gi,
      DirectedSparseGraph g, DirectedSparseGraph gj) {
    List<Set<VertexPair>> subgraphs = Ullman.subgraphIsomorphism(gj, gi);
    assertEquals(4, subgraphs.size());
    subgraphs = Ullman.subgraphIsomorphism(gj, g);
    assertEquals(4, subgraphs.size());
  }

  @Test
  @Depends("testCircleReflexiveEdgeAndSelfLoop(edu.uci.ics.jung.graph.impl.
       DirectedSparseGraph,edu.uci.ics.jung.graph.impl.DirectedSparseGraph);"
      + "testCircleSubgraphIsomorphisms;"
      + "IsomorphTestUllman.testAddVertexNotIsomorph(edu.uci.ics.jung.graph.impl.
          DirectedSparseGraph)")
  public void testSingleSelfLoopVersCircle(DirectedSparseGraph gi,
      DirectedSparseGraph g, DirectedSparseGraph gj) {
    gj = (DirectedSparseGraph) gj.copy();

    for (Object vertex : gj.getVertices()) {
      gj
          .addEdge(new DirectedSparseEdge((Vertex) vertex,
              (Vertex) vertex));
    }

    List<Set<VertexPair>> subgraphs = Ullman.subgraphIsomorphism(gj, gi);
    assertEquals(2, subgraphs.size());
    subgraphs = Ullman.subgraphIsomorphism(gj, g);
    assertEquals(0, subgraphs.size());
  }
}
```

Listing 16: SpecialSubgraphTest.java

```java
@RunWith(ComposedTestRunner.class)
public class SpecialSubgraphTest {

  public SpecialSubgraphTest(){}

  @Test
  @Depends("IsomorphTestUllman.testEmptyGraphIsIsomorphToItself")
  public void testWithSelfLoop(DirectedSparseGraph sg){
    sg = ( DirectedSparseGraph ) sg.copy();
    DirectedSparseGraph sg2 = ( DirectedSparseGraph ) sg.copy();

    Vertex s1 = sg.addVertex(new DirectedSparseVertex());
    Vertex s2 = sg.addVertex(new DirectedSparseVertex());
    Vertex s3 = sg.addVertex(new DirectedSparseVertex());
    Vertex s4 = sg.addVertex(new DirectedSparseVertex());
    Vertex s5 = sg.addVertex(new DirectedSparseVertex());

    sg.addEdge(new DirectedSparseEdge(s1, s2));
    sg.addEdge(new DirectedSparseEdge(s3, s3));
    sg.addEdge(new DirectedSparseEdge(s3, s4));
    sg.addEdge(new DirectedSparseEdge(s3, s5));

    Vertex ss1 = sg2.addVertex(new DirectedSparseVertex());
    Vertex ss2 = sg2.addVertex(new DirectedSparseVertex());
    Vertex ss3 = sg2.addVertex(new DirectedSparseVertex());
    Vertex ss4 = sg2.addVertex(new DirectedSparseVertex());
    Vertex ss5 = sg2.addVertex(new DirectedSparseVertex());

    sg2.addEdge(new DirectedSparseEdge(ss1, ss3));
    sg2.addEdge(new DirectedSparseEdge(ss3, ss2));
    sg2.addEdge(new DirectedSparseEdge(ss3, ss4));
    sg2.addEdge(new DirectedSparseEdge(ss3, ss5));

    assertFalse(Ullman.areIsomorph(sg, sg2));
    assertFalse(Ullman.areIsomorph(sg2, sg));
  }
}
```

Listing 17: VertexPairTest.java

```
@RunWith(ComposedTestRunner.class)
public class VertexPairTest {
  public VertexPairTest() {}

  @Test
  public VertexPair testPairEquals() {
    Vertex v1 = new DirectedSparseVertex();
    Vertex v2 = new DirectedSparseVertex();

    VertexPair p1 = new VertexPair(v1, v2);
    VertexPair p2 = new VertexPair(v1, v2);

    assertEquals(p1, p2);
    assertTrue(p1.equals(p2));
    assertTrue(p2.equals(p1));

    return p1;
  }

  @Test
  @Depends("testPairEquals")
  public void testPairsNotEqual(VertexPair p1) {
    Vertex v1 = p1.v1;
    Vertex v3 = new DirectedSparseVertex();

    VertexPair p3 = new VertexPair(v1, v3);
    assertFalse(p1.equals(p3));
    assertFalse(p3.equals(p1));
  }
}
```