Test Design Document

Authors

Team Name : PRX
Team Members : Liang Yu, Parvathy Unnikrishnan Nair, Reto Kleeb, Xinyi Wang

Purpose of this Document

This document explains the general idea of the continuous integration setup for the development of the SCG Arena.

General Idea of a Continuous Integration System

Martin Fowler [1]:
"Continuous Integration is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily - leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible. Many teams find that this approach leads to significantly reduced integration problems and allows a team to develop cohesive software more rapidly."

A continuous integration (CI) system serves as a central, authorities instance for the state of the code and the state of the test cases. Every contribution (commit) has to compile and pass the tests on the server. This ensures that the builds and test cases do not depend on any local developer-machine configuration.

The continuous integration system works closely with a central source code repository, similar to the CI system, the repository is the central and only place for source code, code that is not checked-in in the correct directory (branch) is considered inexistent.

Procedures / Unit Test organization

Each component of the system is tested in a dedicated Test-Class in the test directory. The package of the test class is the same as the one of the SUT (System Under Test). The test class is prefixed with the word Test. The class tests each (reasonable) public method of the class with at least the following scenarios:

● Common Case
● Extreme Case

e.g. Claim claim = Claim.parse("csp.CSPInstancesSet{{ (12 800) }} scg.protocol.PositiveSecret {{}} 1.1 0.8")") -- for the quality here, we choose the 0.8, 1.0 and 1.1 to verify the behaviour close to the boundaries.

● Invalid / Unexpected Cases

e.g. *CSPInstanceSet cspInstanceSetWrong = CSPInstanceSet.parse("(523 768)") – 523 and 768 are way above the upper boundary.*

Testing the SCG Court

The network specific functionality is definitely an area that would benefit from the refactorings that are mentioned below (“Coverage could be improved”). The current coverage is insufficient (overall around 20%). We were wondering if it would make sense to build brittle and extremely expensive solutions to work around the issues of partly untestable code, or if the project would benefit from specific refactorings in the problematic areas and then tests could be introduced much more reasonable.

Testing the Playground-Specific Parts

If the project has reached a stable status, playground designers should be able to focus on the code that was introduced in the playground specific *.beh files (Example: hsrDomain.beh and hsrAvatar.beh) and the playground specific objects that were introduced with the grammar (Example: hsrDomain.cd and hsrAvatar.cd).

These testing-efforts can be split up into two sub-tasks:

- **Introducing the Playground:**
  This requires testing of all the methods that are introduced in the xxxDomain.beh file as well as the testing if the transformation from- and to strings of all playground specific elements works as expected.

- **Creating a new Avatar:**
  It is important to notice that this testing-part does not directly relate to the stability and reliability of the overall system, these tests are a guideline to help avatar creators to write avatars that run stable in tournaments: The functionality that is written in xxxAvatar.beh should be thoroughly tested to prevent an avatar from being kicked out for a small programming mistake (NPE, etc.).

Test Coverage

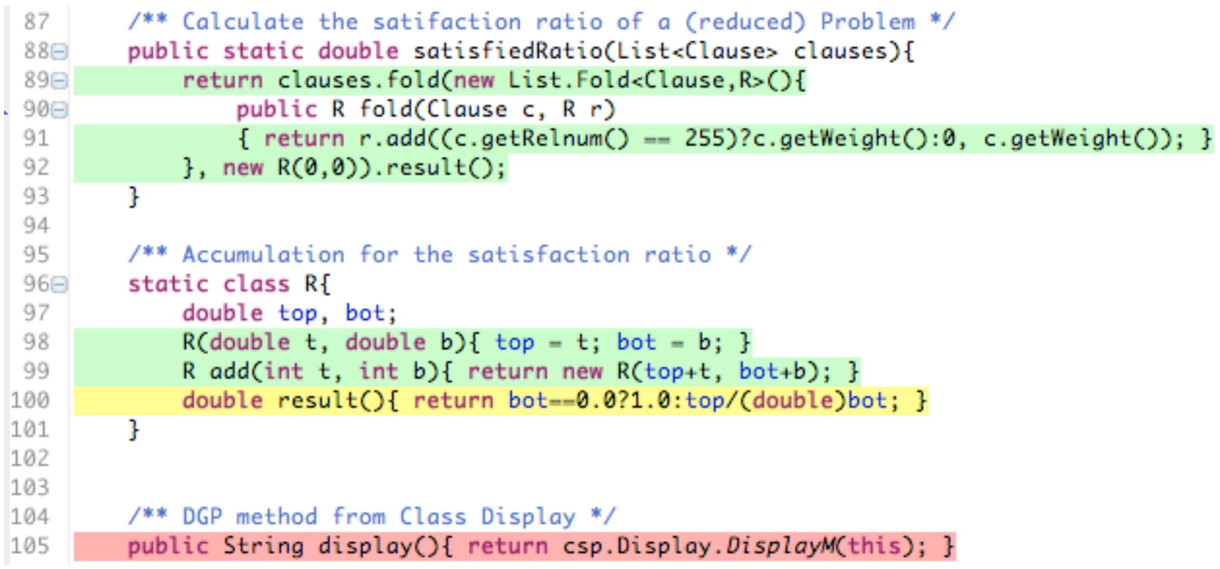
One important metric related to testing is the so called “Code Coverage” [4]. This number describes the number of source code lines that are covered by a complete run of all tests divided by the number of total lines. A naïve observer might come to the conclusion that a project that has a code-coverage of 100% is the ultimate goal and will prove the system solid. The reality is however not that simple:

- **“80 / 20 Rule”:**
  As a rule of thumb, coverage in the area of 80 - 90% is a reasonable, desirable number for most projects. The amount of work that would be necessary to reach 100% is often not proportional to the information gain [5]. Real world systems often contain elements that are too hard to test (while maintaining reasonable costs). The recommended approach is to focus on the central elements and tests these thoroughly before attempting to generally increase the code coverage.

- **Covering a line does not necessarily mean that the test was intelligent:**
  The systems to generate these numbers follow some simple rules, the most important one is, if the executed code has touched line xy, mark this line as covered. These systems however are unable to tell the difference between just executing a certain statement with a very simple case, or a tricky, edge case that could case a problem.

Test-Coverage in the SCG Project

The fact that parts of the code in the project are automatically generated makes the generation of exact number a bit more challenging. In a typical project that contains generated code, it is often undesirable to test the generated code (these tests should happen in the framework that writes the code, not in the project that uses the framework). The tools to measure coverage can be instructed to ignore certain directories (we used EclEmma [6], but there are plenty of alternatives with comparable functionality available). This often solves the problem, in our case however, the folder for generated source (typically called "gen") also contains manually written code that is copied from the *.beh files. It certainly would be possible to extend a coverage report tool to allow a more fine-grained differentiation between DemeterF [7] code and hand-written methods.



```
    /** Calculate the satifaction ratio of a (reduced) Problem */
    public static double satisfiedRatio(List<Clause> clauses){
        return clauses.fold(new List.Fold<Clause,R>(){
            public R fold(Clause c, R r)
            { return r.add((c.getRelnum() == 255)?c.getWeight():0, c.getWeight()); }
        }, new R(0,0)).result();
    }

    /** Accumulation for the satisfaction ratio */
    static class R{
        double top, bot;
        R(double t, double b){ top = t; bot = b; }
        R add(int t, int b){ return new R(top+t, bot+b); }
        double result(){ return bot==0.0?1.0:top/(double)bot; }
    }


    /** DGP method from Class Display */
    public String display(){ return csp.Display.DisplayM(this); }
```

The screenshot displays the result of a coverage analysis run using the EclEmma Eclipse plugin, lines that were covered during the run are highlighted in green. Yellow stands for a partially executed line (typically inline conditionals), and a line highlighted in red, was not executed during the test. The image shows a situation that is typical for this project. The file (CSPInstance.java) contains code that was generated by DemeterF as well as code that was copied from a *.beh file.

Considering these circumstances, we collected the test-coverage in a manual process, this approach has the advantage that it implicitly includes a cursory code-review, it is however not feasible for larger code-bases.

Solid Coverage (above 70% coverage)

The packages for the protocol-interpreter as well as the packages for the protocols have a solid coverage. Possible ideas for further improvement would be: More corner-cases, more invalid inputs.

Coverage could be improved (below 70% coverage)

One of the major difficulties while writing tests, was the fact that a lot of the code was written in a way that made method calls expensive. What does this mean? The following three items describe issues that we found particularly important:

- **“Multi-Purpose Methods”**
  Methods should be written as small units with one specific purpose. There are two very simple indicators to recognize if a certain method fulfils this criterion: 1. Is the number of lines less than 10. 2. Can I clearly describe the methods purpose in a single sentence?

- **“Tangled Objects as Arguments”**
  This is a typical issue in Java – Web projects, often the functionality lies inside of methods that can only be called with objects that are unnecessarily complex to create. A prominent example is the HTTPRequest object in Servlets. There are Mock-Frameworks that help to mitigate these issues, the more straightforward way however, would be to write and test methods that only rely on primitive or simple Domain-Specific objects. Instead of having one method that takes an object that is expensive to construct, work with two methods, the first one takes the elements from the expensive argument object and then calls the tailored method with simple arguments.

- **“State in Classes”**
  One of the ideas that are heavily promoted lately, especially by people that are experienced users of functional languages, is the idea reducing state in objects. State-Depended logic is much harder to predict and therefore to test. I will illustrate this issue with two simple examples:

Stateless	**State-full (with implicit state-changes)**

`object.calculateResult(“xy”) = 5`
`object.calculateResult(“xy”) = 5`
`object.calculateResult(“xy”) = 5`
`object.calculateResult(“xy”) = 5`
`...`
`...`
	`object.calculateResult(“xy”) = 4`
	`object.calculateResult(“xy”) = 5`
	`object.calculateResult(“xy”) =>`
	`Exception “Calculation does not make sense”`

In the first example (on the left), the result of a certain call is (and will be) always the same, the result does never depend on anything that happened before. On the right hand side however, each call updates an internal state that influences the return value of the method.

Such behavior and other examples of implicit state-updates might seem like a convenient shortcut for the initial developer, they are however extremely hard to understand for the next person that has to test or maintain the code.

Central Source Code Repository

One of the central aspects of a CI server, is its ability to check a repository for changes and based on such events trigger actions (usually build and test, but we could easily extend these capabilities to anything else that might be desired).

The most pragmatic approach in our case was to choose an existing, free hosting provider for Subversion.

Build Setup

The cornerstone of the build of the SCG arena is a fairly simple Ant [2] script:

```
<project name="cs6515" default="build" xmlns:ivy="antlib:org.apache.ivy.ant">

      <!--
      *********************
      ANT Build File for the CS 5500 project
      *********************
      -->

      <!-- Basic Directory Definitions -->
      <property name="build.dir" value="Build" />
      <property name="src.dir" value="src" />
      <property name="srcGen.dir" value="gen" />
      <property name="test.dir" value="test" />
      <property name="test.report.dir" location="testreport" />

      <!-- Classpath for the project-libraries -->
      <path id="classpath">
            <fileset dir=".">
                  <include name="**/*.jar" />
            </fileset>
      </path>


      <!-- Define the classpath which includes the junit.jar and the classes
after compiling-->
      <path id="junit.class.path">
            <pathelement location="${build.dir}" />
            <fileset dir=".">
                  <include name="**/*.jar" />
            </fileset>
      </path>
```

```
	<target name="build" description="--> resolve dependencies, compile and
run the project">
		<mkdir dir="${build.dir}" />
		<javac destdir="${build.dir}" includeantruntime="true">
			<src path="${src.dir}" />
			<src path="${srcGen.dir}" />
			<src path="${test.dir}" />
			<classpath refid="classpath" />
		</javac>
	</target>

	<!-- executes all tests (Test*.java files) and generates
	html reports in the ${test.report.dir} -->
	<target name="test" depends="build" description="--> test the project">
		<mkdir dir="${test.report.dir}" />
		<junit printsummary="on" fork="true" haltonfailure="false"
includeantruntime="true">
			<classpath refid="junit.class.path" />
			<formatter type="xml" />
			<batchtest todir="${test.report.dir}">
				<fileset dir="${test.dir}">
					<include name="**/Test*.java" />
				</fileset>
			</batchtest>
		</junit>
		<junitreport todir="${test.report.dir}">
			<fileset dir="${test.report.dir}" />
			<report todir="${test.report.dir}" />
		</junitreport>
	</target>

	<target name="clean" description="--> clean the project">
		<delete includeemptydirs="true" quiet="true">
			<fileset dir="${build.dir}" />
			<fileset dir="${test.report.dir}" />
		</delete>
	</target>

</project>
```

Explanation of the defined Ant Tasks

build: Runs JavaC for all source files in the project
test: compiles all sources (depends=”build”) and then executes all Junit-Tests in all Java classes that start with the name Test. The results of the tests are written to the directory defined in “test.report.dir”

Setup for Jenkins CI

Jenkins CI [3] is a freely available Build-System that is easily configurable and extensible. The system is distributed as a simple jar file that does have no dependencies and can be started from the command line right away. The following command starts a simple (but **unprotected**) Jenkins instance.

```
java -jar jenkins.war
```

After entering the command, a webinterface that allows the configuration of test-plans is available on http://localhost:8080

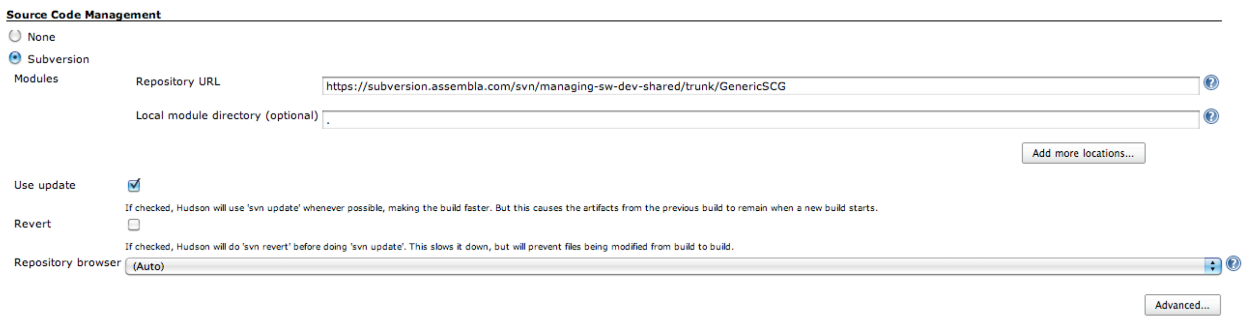
This setup however does give full access and change rights to anybody, starting the system with the following arguments:

```
java -jar jenkins.war --argumentsRealm.passwd.cs5500=scgarena --
argumentsRealm.roles.cs5500=admin
```

Will prevent these issues. The arguments that are highlighted in green are the ones that need to be changed: cs5500 is the username, scgarena is the password. The last argument assigns administrative privileges to the user cs5500.

Setting up a Build

Once Jenkins is running, new Jobs can be configured. A click on "New Job" will bring up a form with the following four main elements that are explained in the next section:



Source Code Management

None [✓ Subversion]

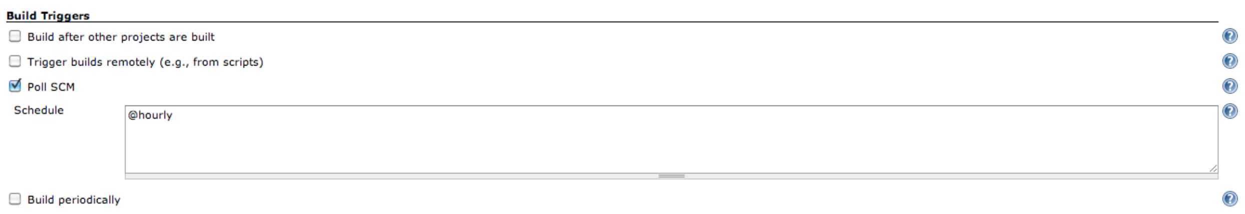
Modules
Repository URL	https://subversion.assembla.com/svn/managing-sw-dev-shared/trunk/GenericSCG
Local module directory (optional)	.

Use update ✓
If checked, Hudson will use 'svn update' whenever possible, making the build faster. But this causes the artifacts from the previous build to remain when a new build starts.

Revert ☐
If checked, Hudson will do 'svn revert' before doing 'svn update'. This slows it down, but will prevent files being modified from build to build.

Repository browser (Auto)

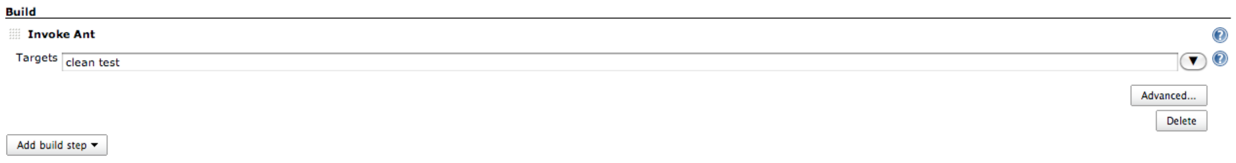
The first section defines the URL of the SVN Repository, if the repository is not publicly readable, the corresponding credentials need to be entered. It is important to notice that there are plugins for Jenkins that allow different Version Control systems (such as Git) to be used.



Build Triggers

☐ Build after other projects are built
☐ Trigger builds remotely (e.g., from scripts)
✓ Poll SCM
Schedule: @hourly
☐ Build periodically

After the Repository to be checked is defined, we need to specify how often the system should check for modifications.

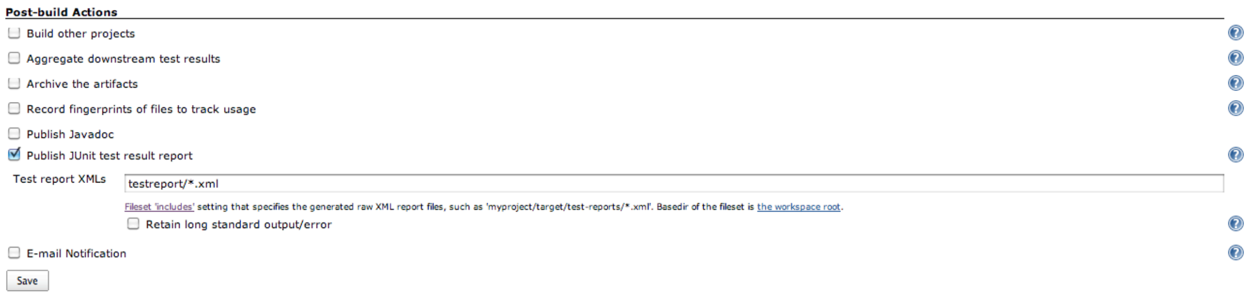


Build

Invoke Ant

Targets: clean test

Once the repository and the schedule are set up, we need to define what should happen once the system detects a change in the given repository. In our case we simply call two ant targets that we defined in the build.xml file. The first target “clean” deletes any artifacts from any previous builds, this ensures that we never have any “leftovers” that could lead to inaccurate or strange test-results. The next target is the test target (Ant allows users to simply pass a list of all targets that should be executed in order). We have seen in the build.xml file that this target does depend on the target “build”. This setup will ensure that any call to test will first compile all sources.



Post-build Actions

- ☐ Build other projects
- ☐ Aggregate downstream test results
- ☐ Archive the artifacts
- ☐ Record fingerprints of files to track usage
- ☐ Publish Javadoc
- ☑ Publish JUnit test result report

Test report XMLs: testreport/*.xml

Fileset 'includes' setting that specifies the generated raw XML report files, such as 'myproject/target/test-reports/*.xml'. Basedir of the fileset is the workspace root.

- ☐ Retain long standard output/error
- ☐ E-mail Notification

The final step before we have a build system that delivers valuable information about the status of the system, is to configure the system to read the results of the JUnit tests. It is important that the given directory matches the directory that is set in the Ant file.

Sources:

[1] http://www.martinfowler.com/articles/continuousIntegration.html
[2] http://ant.apache.org/
[3] http://jenkins-ci.org/
[4] http://www.bullseye.com/coverage.html
[5] http://jasonrudolph.com/blog/testing-anti-patterns-how-to-fail-with-100-test-coverage/
[6] http://www.eclemma.org/
[7] http://www.ccs.neu.edu/home/chadwick/demeterf/