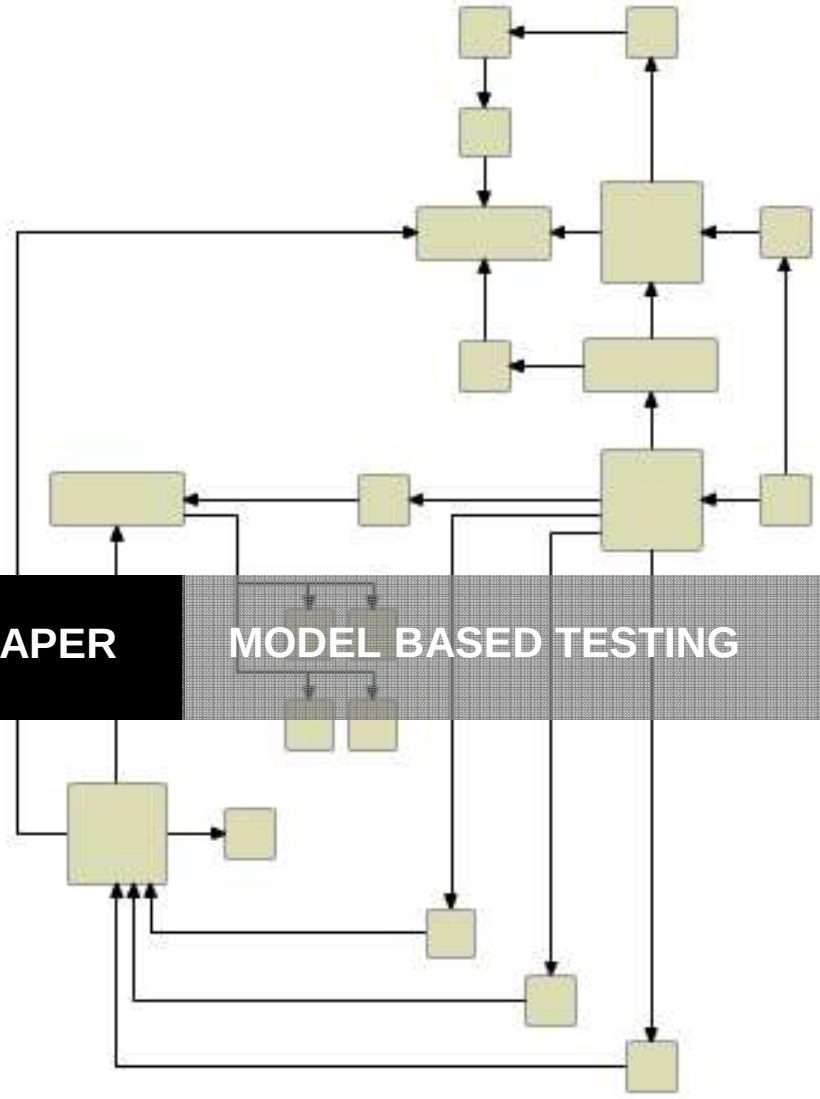
prolore



WHITE PAPER

MODEL BASED TESTING

Testramverk och Model based testing med java i praktiken

Målet med artikeln

Den här artikeln syftar till att praktiskt visa hur man med ett antal enkla steg kan komma igång med ett ramverk i java för model-based testing. Här kommer jag att exemplifiera med ett antal verktyg för att interagera med applikationen under test men det är så klart möjligt att använda andra verktyg.

Dina kunskaper som läsare

Jag förutsätter att du som läsare är insatt i utveckling i Java samt att du är bekant med modellbaserad test, om du saknar denna grund kan du gärna läsa vidare här.
För att sätta igång behövs ett antal verktyg:

- Eclipse (eller NetBeans/ IntelliJ)
- Graphwalker (model based testing tool)
- Selenium
- Maven

Projekt uppsättning

För att enkelt komma igång med ramverket använder vi *Maven* för att paketera ramverket. Först av allt behöver vi sätta upp en konfigurationsmodell, Project Object Model (**POM**), som beskriver hur projektet ska byggas och dess externa beroenden.
Här behöver vi sätta upp de olika ingående modulerna, **Graphwalker** och **Selenium**.
Generellt så lägger jag inte in specifika versionsnummer då dessa förändras relativt ofta. Värt att notera är att mina exempel bygger på graphwalker 2.5.9 och senare.
Exempel på POM.xml:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>testautomation</groupId>
  <artifactId>RegressionTest</artifactId>
  <version>1.0-SNAPSHOT</version>
  <repositories>
   <repository>
     <id>maven repo</id>
     <url>http://repo1.maven.org/maven2/</url>
   </repository>
   </repositories>
  <dependencies>
   <dependency>
     <groupId>org.graphwalker</groupId>
     <artifactId>graphwalker</artifactId>
     <version>..</version>
     <type>jar</type>
     <scope>compile</scope>
   </dependency>
   <dependency>
     <groupId>org.seleniumhq.selenium</groupId>
     <artifactId>selenium</artifactId>
     <version>..</version>
     <type>jar</type>
     <scope>compile</scope>
```

```
    </dependency>
    <dependency>
      <groupId>org.seleniumhq.selenium</groupId>
      <artifactId>selenium-common</artifactId>
      <version>..</version>
      <type>jar</type>
      <scope>compile</scope>
    </dependency>
    <dependency>
      <groupId>org.seleniumhq.selenium</groupId>
      <artifactId>selenium-firefox-driver</artifactId>
      <version>..</version>
      <type>jar</type>
      <scope>compile</scope>
    </dependency>
    <dependency>
      <groupId>log4j</groupId>
      <artifactId>log4j</artifactId>
      <version>..</version>
    </dependency>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>..</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

pom.xml

Skapa testimplementation

Nu har vi ett skal för att börja skapa våra tester. Nästa steg blir att utifrån pom-filen skapa ett nytt projekt i vald utvecklingsmiljö. Här förutsätter jag att du som läser vet hur man skapar ett maven/java projekt. Efter att vi skapat vårt projekt är det nu tid för att börja skapa en test-implementation. Vi börjar med att skapa en ny klass och för att kunna använda den enkelt i vårt ramverk ska den utöka (*extends*) en klass ifrån **Graphwalker**.

```
package se.prolore;

import org.graphwalker.generators.PathGenerator;
import org.graphwalker.multipleModels.ModelAPI;

public class ProloreTestImpl extends ModelAPI{

  public ProloreTestImpl(String model, boolean efsm,
      PathGenerator generator, boolean weight) {
    super(model, efsm, generator, weight);
  }
```

ProloreTestImpl.java

När vi kommit så här långt behöver vi lägga till ett verktyg för interaktion med testapplikationen. I det här fallet selenium. Här väljer jag att skapa möjlighet att köra med API'er för både **Selenium1** och **Selenium2**.

```
  private WebDriver masterdriver;  //Selenium2
  private DefaultSelenium driver;  //Selenium1
```

```
  public void e_startbrowser(){
    FirefoxProfile profile = new FirefoxProfile();
    masterdriver = new FirefoxDriver(profile);
    String startpage = "about:blank";
    masterdriver.get(startpage);
    driver = new WebDriverBackedSelenium(masterdriver,startpage);
  }
```

ProloreTestImpl.java

Nu är det ganska rakt på sak att skapa ytterligare implementation av olika teststeg utifrån den test- modell som ska automatiseras.

Använda ett unit test ramverk för exkvering

När vi nu skapat en testimplementation som korresponderar mot våran testmodell så är det nu hög tid att titta på hur exekverar vi egentligen?
Jag väljer här att använda ett unittest-ramverk, i mitt fall **Junit4** (det går lika bra med testng).

```
package IntegrationTest;

import static org.junit.Assert.*;
import org.junit.Test;
import org.graphwalker.conditions.EdgeCoverage;
import org.graphwalker.generators.RandomPathGenerator;
import org.graphwalker.multipleModels.ModelAPI;
import org.graphwalker.multipleModels.ModelHandler;
import org.apache.log4j.Logger;
import org.graphwalker.Util;
import se.prolore.ProloreTestImpl;

public class ProloreWebIT {
  private static final Logger logger = Util.setupLogger(ProloreWebIT.class);

  @Test
  public void proloreWeb() throws Exception {
    ModelHandler modelhandler = new ModelHandler();
    ModelAPI model = new ProloreTestImpl("graphml/proloreweb.graphml",
        true, new RandomPathGenerator(new EdgeCoverage(1.0)),false);
    modelhandler.add("StartModel", model);
    modelhandler.execute("StartModel");
    logger.info(modelhandler.getStatistics());
    assertTrue(modelhandler.isAllModelsDone());
  }
}
```

ProloreWebIT.java

På detta sätt kan vi nu enkelt starta och exekvera tester med hjälp av **Junit4**. Men i många projekt vill man ofta lägga End2End tester efter exempelvis en deploy av sin site eller applikation.

Lägga regressionstester/integrationstester i en egen fas

Nu kan vi bygga vårt projekt med hjälp av **Maven** men det sker kanske vid fel tidpunkt... Då kan vi gå in i vår **POM** och lägga till en ny bygg-fas där vi bryter ut våra regressions/integrationstester. Här exkluderar vi även testerna som slutar med *IT.java och *IntegrationTest.java ifrån att exekveras under unittest-fasen i maven bygget.

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-surefire-plugin</artifactId>
      <configuration>
        <excludes>
          <exclude>**/*IT.java,**/*IntegrationTest.java</exclude>
        </excludes>
      </configuration>
      <executions>
        <execution>
          <id>integration-tests</id>
          <phase>integration-test</phase>
          <goals>
            <goal>test</goal>
          </goals>
          <configuration>
            <skip>false</skip>
            <excludes>
              <exclude>none</exclude>
            </excludes>
            <includes>
              <include>**/*IT.java,**/*IntegrationTest.java</include>
            </includes>
          </configuration>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

pom.xml

Utöka med flera testverktyg för interaktion med applikationer

Som avslutning vill jag bara snabbt lista några ytterligare verktyg som kan vara intressanta att lägga till för att öka möjligheterna att testa olika applikationer.

- Sikuli – Automation med bildigenkänning. Ett enkelt verktyg med ett IDE skrivet i python men själva script motorn har ett java api gränssnitt som enkelt pluggas in i det ovan beskrivna ramverket.
- UISpec4J – Ett ramverk för automation av Java/Swing Gui-automation. Även det enkelt att plugga in i ramverket.
- Onestonesoup – Ett ramverk som jag använt för att skapa screen recording som helt enkelt spelar in vad som sker på skärmen under ett test.