

```
package generator;

/*
 * TODO add imports to the imports list
 * */
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Hashtable;
import java.util.Properties;

/**
 * Class that stores all the information needed to create the TestClass.<br>
 * - Array of objects<br>
 * - Array of imports<br>
 * - Package of the test<br>
 * - Name of the class<br>
 * - A dictionary of Method.toString()-&gt;ArrayList<Instruction>;

 * @author Sergio Alcocer
 *
 */
public class TestInfo {

    private static TestInfo instance = null;
    private ArrayList<ObjectCreated> objects;
    private ArrayList<String> imports;
    private String testPackage, className;
    private Hashtable<String,ArrayList<Instruction>> thingsToDo;

    private TestInfo(){
        objects = new ArrayList<ObjectCreated>();
        thingsToDo = new Hashtable<String,ArrayList<Instruction>>();
    }

    /**
     * Following singleton pattern, it gets the only instance of the class
     * @return TestInfo following singleton pattern
     */
}
```

```
public static TestInfo getInstance() {
    if (instance == null){
        instance = new TestInfo();
    }
    return instance;
}

/**
 * Adds an object to the list
 * @param object object to be added
 */
public void addObject(ObjectCreated object) {
    objects.add(object);
}

/**
 * Sets the package of the application
 * @param pack package to be set
 */
public void setPackage(String pack) {
    testPackage = pack;
}

/**
 * Sets the name of the class
 * @param className name of the class to be set
 */
public void setClassName(String className) {
    this.className = className;
}

/**
 * Sets all the imports
 * @param imports ArrayList<String> with the imports to be set
 */
public void setImports(ArrayList<String> imports) {
    this.imports = imports;
}

/**
```

```
* Check if contains an object
* @param objName name of the object
* @return true if the object has been stored, false otherwise
*/
public boolean contains(String objName){
    boolean found = false;
    for(int i = 0; i < objects.size() && !found; i++){
        if(objects.get(i).getObjectName().equals(objName)){
            found = true;
        }
    }
    return found;
}

/**
 * Gets all the objects representation
 * @return ArrayList<String> with the representation of all the objects
 */
public ArrayList<String> getObjectsString(){
    ArrayList<String> returned = new ArrayList<String>();
    for(int i = 0; i < objects.size(); i++){
        returned.add(objects.get(i).getObjectName() + " => " + objects.get(i).toString());
    }
    return returned;
}

/**
 * Gets all the instructions associated to a method
 * @param methodText method.toString() of the method
 * @return ArrayList<String> with the instructions associated to that method
 */
public ArrayList<String> getThingsToDoString(String methodText){
    ArrayList<String> returned = new ArrayList<String>();
    ArrayList<Instruction> arrayInstructions = thingsToDo.get(methodText);
    if(arrayInstructions != null){
        for(int i = 0; i < arrayInstructions.size(); i++){
            returned.add(arrayInstructions.get(i).getCode());
        }
    }
    return returned;
}
```

```
/**
 * Creates the Test file
 * @param filename full path of the file
 */
public void toFile(String filename){

    try {
        FileWriter fw = new FileWriter(filename);

        BufferedWriter out = new BufferedWriter(fw);

        out.write( "package " + testPackage + ";\r\n\r\n" +

            "import junit.framework.Test;\r\n" +
            "import junit.framework.TestCase;\r\n" +
            "import junit.framework.TestSuite;\r\n\r\n" +
            "// Other imports\r\n");

        if (imports != null){
            for (int i = 0; i < imports.size(); i++){
                out.write( "import " + imports.get(i) + ";\r\n");
            }
        }

        out.write( "public class " + className + "Test extends TestCase{\r\n");
        out.write( "\t//Attributes\r\n");

        if (objects.size() != 0){
            out.write( "\tprivate " + className);
            String attributeList = new String();
            for (int i = 0; i < objects.size(); i++){
                attributeList += objects.get(i).getObjectName() + ", ";
            }
            attributeList = attributeList.substring(0,attributeList.length() - 2); //I'm sure because size() != 0
            out.write( " " + attributeList + ";\r\n");
        }
        out.write( "\tpublic " + className + "Test(String name){\r\n" +
            "\t\tsuper(name);\r\n" +
            "\t}\r\n");
    }
}
```

```
out.write( "\tprotected void setUp() throws Exception{\r\n" +
           "\t\tsuper.setUp();\r\n"});

out.write( "\t\t// Constructors\r\n");

for (int i = 0; i < objects.size(); i++){
    out.write( "\t\t" + objects.get(i).getObjectName() + " = new " + className + objects.get(i).toString() +
";\r\n");
}

out.write( "\t}\r\n" +
           "\tprotected void tearDown() throws Exception {\r\n" +
           "\t\tsuper.tearDown();\r\n"});

out.write( "\t\t//null asignments\r\n");
for (int i = 0; i < objects.size(); i++){
    out.write( "\t\t" + objects.get(i).getObjectName() + " = null;\r\n");
}

out.write( "\t}\r\n");

out.write( "\t//testMethods()...\r\n");

Object[] objectKeys = thingsToDo.keySet().toArray();
String[] keys = new String[objectKeys.length];
for (int i = 0; i < keys.length; i++){
    String key = (String)objectKeys[i];
    key = key.substring(0, key.lastIndexOf('(') - 1);
    keys[i] = key.substring(key.lastIndexOf(' ') + 1, key.length());
    // now I have the bare function name

    out.write( "\tpublic void test" + keys[i] + "() {\r\n" +
               "\t\t\t//Here goes the asserts for this method test\r\n"});

    ArrayList<Instruction> arrayInstructions = thingsToDo.get((String)objectKeys[i]);
    for (int j = 0; j < arrayInstructions.size(); j++){
        out.write( "\t\t\t" + arrayInstructions.get(j).getCode() + "\r\n");
    }
}
```

```
        out.write( "\t}\r\n\r\n");

    }

    out.write( "\tpublic static Test suite(){\r\n" +
        "\t\t TestSuite suite = new TestSuite();\r\n");

    out.write( "\t\t//Suite creations\r\n");

    for (int i = 0; i < keys.length; i++){
        out.write( "\t\tsuite.addTest(new BookTest(\"test\" + keys[i] + "\")); \r\n");
    }

    out.write( "\t\treturn suite;\r\n" +
        "\t}\r\n" +
        "\t");

    out.close();
    fw.close();

} catch (IOException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}

}

/**
 * Adds an instruction to the instructions of a method
 * @param key method.toString()
 * @param instr Instruction to be added
 */
public void addThingsToDo(String key, Instruction instr){
    if (thingsToDo.containsKey(key)){
        thingsToDo.get(key).add(instr);
    }else{ // is new and I have to add it
        ArrayList<Instruction> temp = new ArrayList<Instruction>();
        temp.add(instr);
        thingsToDo.put(key, temp);
    }
}
```

```
/**
 * Updates an instructions of a method
 * @param key method.toString()
 * @param instr Instruction to be set
 * @param index index of the original Instruction
 */
public void addThingsToDo(String key, Instruction instr, int index){
    if (thingsToDo.containsKey(key)){
        thingsToDo.get(key).set(index, instr);
    }else{ // is new and I have to add it
        ArrayList<Instruction> temp = new ArrayList<Instruction>();
        temp.add(instr);
        thingsToDo.put(key, temp);
    }
}

/**
 * Deletes a given instruction for a given method
 * @param key method.toString()
 * @param index index of the instruction to delete
 */
public void deleteThingsToDo(String key, int index){
    if(thingsToDo.containsKey(key)){
        thingsToDo.get(key).remove(index);
    }
}

/**
 * Gets the instruction for a given method
 * @param key method.toString()
 * @param index index of the instruction
 * @return instruction located on position and method provided
 */
public Instruction getThingToDo(String key, int index){
    if(thingsToDo.containsKey(key)){
        return thingsToDo.get(key).get(index);
    }else{
        return null;
    }
}
```

```
/**
 * Gets an object from the ones to be created
 * @param i index of the object
 * @return ObjectCreated that is located at the given position
 */
public ObjectCreated getObject(int i){
    return objects.get(i);
}

/**
 * Gets the index of an object name given
 * @param obj name of the object
 * @return index in which is located
 */
public int getObjectIndex(String obj){
    int returned = -1;
    for(int i = 0; i < objects.size() && returned == -1; i++){
        if(obj.equals(objects.get(i).getObjectName())){
            returned = i;
        }
    }
    return returned;
}

/**
 * Gets the name of the class
 * @return name of the class
 */
public String getClassName(){
    return className;
}

/**
 * Loads the information of a given file
 * @param filename full path of the file to be loaded
 * @param current ArrayList of the methods.toString()
 */
public void loadFromFile(String filename, ArrayList<String> current){
    Properties configFile = new Properties();
```

```
try {
    configFile.load(new java.io.FileInputStream(new java.io.File(filename)));

    objects = new ArrayList<ObjectCreated>();
    thingsToDo = new Hashtable<String, ArrayList<Instruction>>();
    //objects
    int objectsCount = Integer.parseInt(configFile.getProperty("OBJECT_COUNT"));
    for(int i = 0; i < objectsCount; i++){
        String objectName = configFile.getProperty("O_" + i + "_OBJECT_NAME");
        int paramCount = Integer.parseInt(configFile.getProperty("O_" + i + "_PARAM_COUNT"));
        ArrayList<String> params = new ArrayList<String>();
        for (int j = 0; j < paramCount; j++){
            params.add(configFile.getProperty("O_" + i + "_" + j));
        }

        ObjectCreated oc = new ObjectCreated(objectName, params);
        objects.add(oc);
    }

    //instructions
    int methods = Integer.parseInt(configFile.getProperty("METHOD_COUNT"));
    for(int i = 0; i < methods; i++){
        int instructions = Integer.parseInt(configFile.getProperty(i + "_INSTRUCTIONS"));
        String methodName = configFile.getProperty(i + "_METHOD");
        if(current.contains(methodName)){
            for(int j = 0; j < instructions; j++){
                String instType = configFile.getProperty(i + "_" + j + "_TYPE");
                if(instType.equals("ASSERT")){
                    String assertType = configFile.getProperty(i + "_" + j + "_ASSERT_TYPE");
                    int assertTypeInt;
                    if (assertType.equals("TRUE")){
                        assertTypeInt = Assert.ASSERT_TRUE;
                    }else if(assertType.equals("FALSE")){
                        assertTypeInt = Assert.ASSERT_FALSE;
                    }else{
                        assertTypeInt = Assert.ASSERT_EQUALS;
                    }
                    String objectInvolved = configFile.getProperty(i + "_" + j + "_OBJECT_INVOLVED");
                    String methodAssociated = configFile.getProperty(i + "_" + j + "_METHOD_ASSOCIATED");
                    Assert loaded;
```

```

        if(assertTypeInt == Assert.ASSERT_TRUE || assertTypeInt == Assert.ASSERT_FALSE){
            loaded = new Assert(assertTypeInt == Assert.ASSERT_TRUE, objectInvolved, methodAssociated);
            int paramCount = Integer.parseInt(configFile.getProperty(i + "_" + j + "_PARAM_COUNT"));

            for(int k = 0; k < paramCount; k++){
                String paramData = configFile.getProperty(i + "_" + j + "_" + k);
                loaded.addParam(paramData);
            }
            loaded.setRestCode(configFile.getProperty(i + "_" + j + "_REST_CODE"));
        }else{
            String obj1 = configFile.getProperty(i + "_" + j + "_OBJ1");
            String obj2 = configFile.getProperty(i + "_" + j + "_OBJ2");
            loaded = new Assert(methodAssociated, obj1, obj2);
        }

        addThingsToDo(methodName, loaded);

    }else if(instType.equals("CUSTOM")){
        Custom loaded = new Custom();
        loaded.setCode(configFile.getProperty( i + "_" + j + "_CODE"));
        addThingsToDo(methodName, loaded);
    }
}

}

}

}catch(Exception e){
    e.printStackTrace();
}

}

/**
 * Saves the information of the test
 * @param filename full path of the file to be used
 */
public void saveToFile(String filename){
    try {
        FileWriter fw = new FileWriter(filename);
        BufferedWriter out = new BufferedWriter(fw);

```

```
// TODO save object creations
out.write("# OBJECTS INFORMATION\r\n");
out.write("OBJECT_COUNT = " + objects.size() + "\r\n");
for(int i = 0; i < objects.size(); i++){
    ObjectCreated obj = objects.get(i);
    out.write("\tO_" + i + "_OBJECT_NAME = " + obj.getObject_name() + "\r\n");
    ArrayList<String> params = obj.getParams();
    out.write("\tO_" + i + "_PARAM_COUNT = " + params.size() + "\r\n");
    for (int j = 0; j < params.size(); j++){
        out.write("\t\tO_" + i + "_" + j + " = " + params.get(j) + "\r\n");
    }
}

}
```

```
out.write("\r\n");
Object[] objectKeys = thingsToDo.keySet().toArray();
```

```
out.write("# INSTRUCTIONS INFORMATION\r\n");
out.write("METHOD_COUNT = " + objectKeys.length + "\r\n");
```

```
for (int i = 0; i < objectKeys.length; i++){
    ArrayList<Instruction> insts = thingsToDo.get(objectKeys[i]);
    out.write("\t" + i + "_INSTRUCTIONS = " + insts.size() + "\r\n");
    String methodName = (String)objectKeys[i];
    out.write("\t" + i + "_METHOD = " + methodName + "\r\n");
    for (int j = 0; j < insts.size(); j++){

        switch(insts.get(j).getType()){
            case Instruction.T_ASSERT:
                Assert inst = (Assert) insts.get(j);
                out.write("\t\t" + i + "_" + j + "_TYPE = ASSERT\r\n");
                out.write("\t\t" + i + "_" + j + "_ASSERT_TYPE = " + (inst.getAssertType() == Assert.ASSERT_TRUE?
"TRUE":(inst.getAssertType()==Assert.ASSERT_FALSE?"FALSE":"EQUALS")) + "\r\n");
                out.write("\t\t" + i + "_" + j + "_OBJECT_INVOLVED = " + inst.getObjectInvolved() + "\r\n");

                String auxiliar = methodName.substring(0,methodName.lastIndexOf('(') - 1);

                out.write("\t\t" + i + "_" + j + "_METHOD_ASSOCIATED = " + auxiliar.substring(auxiliar.lastIndexOf(
' ') + 1, auxiliar.length()) + "\r\n");
```

```
        ArrayList<String> params = inst.getParams();
        if (inst.getAssertType() == Assert.ASSERT_EQUALS) {
            out.write("\t\t" + i + "_" + j + "_OBJ1 = " + params.get(0) + "\r\n");
            out.write("\t\t" + i + "_" + j + "_OBJ2 = " + params.get(1) + "\r\n");
        } else {

            out.write("\t\t" + i + "_" + j + "_PARAM_COUNT = " + params.size() + "\r\n");
            for (int k = 0; k < params.size(); k++) {
                out.write("\t\t\t" + i + "_" + j + "_" + k + " = " + params.get(k) + "\r\n");
            }
            out.write("\t\t" + i + "_" + j + "_REST_CODE = " + inst.getRestCode() + "\r\n");
        }
        break;
    case Instruction.T_CUSTOM:
        out.write("\t\t" + i + "_" + j + "_TYPE = CUSTOM\r\n");
        out.write("\t\t" + i + "_" + j + "_CODE = " + insts.get(j).getCode() + "\r\n");
        break;
    default: break;
    }
}

}

out.close();
fw.close();
} catch (IOException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}

}

}
```