

# Automatic testing of Java applications

Helder Daniel

Departamento de Engenharia Electrónica e Informática  
Faculdade de Ciências e Tecnologia  
Universidade do Algarve

July 20, 2019

## Abstract

Software quality and reliability is a major concern in the software development community and industry. Each major programming language have a set of testing frameworks and tools that can perform a variety of tests such as: black box testing or unit testing. Such frameworks can also aid in developing programming skills, at early stages of programmer's training or when acquiring competences with new programming languages. Mooshak (Leal and Silva [2003]) was designed to manage online computer programming contests, performing only black box testing on solo file applications. Therefore it lacks support for complex application testing as well as programming skill training. However it has a complete environment to manage submission of software to a server, test it and give feedback. In this paper are presented extensions to Mooshak, to support the testing of complex multi class and multi file Java source code and also java unit testing.

## 1 Introduction

Reliability of software concerns not only the industry but also the community and the developers of programmes to train programming skills. There are many frameworks and tools to test software reliability and make sure it conforms to specifications. One of such tools is Mooshak (Leal and Silva [2003]). Although it was designed to manage only programming contests according to International Collegiate Programming Contest rules (ICPC [2019]), it has a web portal where users can submit software to be tested on a server and receive feedback on that testing. With Mooshak, submission and testing of single file Java programs are quite straightforward, however there is no way to submit and test a real world multi file Java application. Java is an object oriented programming

language, where programs are defined in classes. Even fair simple programs are organized in a set of different classes. Usually each class is defined in its own file, possibly together with necessary auxiliary classes, which spreads the source code across several files. Furthermore classes are organized in packages and sub-packages, an hierarchy of namespaces with direct mapping on an hierarchy of folders.

Submitting a Java application to be tested in Mooshak, means that all the source code must be defined in just one file, throwing away all the clearness and benefits of source code organization. Another Mooshak limitation is the testing service, which performs only black box testing through the application standard input and standard output.

Here is presented Mooshak extensions to support the testing of complex multi class and multi file Java source code and also java unit testing, with JUnit4 or JUnit5 (JUnit [2021]). it allows also to inject code along with the submitted one. This can be handy to test integration of different modules.

These extensions were designed as a standalone package that can be added to Mooshak without changing a line of the original source code.

## 2 Mooshak testing operation

The default Mooshak's 1.x.x software testing process has 2 steps:

1. If necessary, the submitted code is compiled or assembled first.
2. The application is executed and tested.

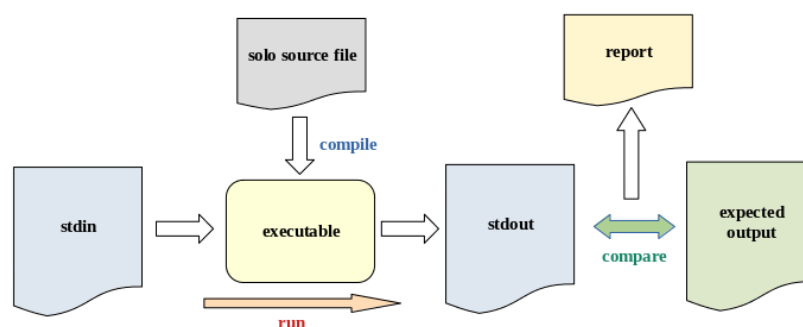


Figure 1: Mooshak testing operation

Mooshak uses a black box approach to software testing. Each test case has 2 files. An **input** file and an **expected output** file.

The testing process, executes the application as many times as the test cases. For each test case the test case input file is presented at the testing application standard input. Then the application standard output is stored in a file that is compared byte by byte with the test case expected output file. If both files match, the test is marked as successful, if not as a failure.

In the end a report with the information of each test is generated and made available for a privileged user. This means that the one that submitted the test has no access to this report. This makes sense for managing programming contests but it is not desirable when testing applications in production environment.

### 3 Extending Mooshak to support Java multi file source code testing

To extend Mooshak to test source code spawned across multiple files, it must be addressed the default submission of a solo file as shown in figure 1. Since these extensions were designed to be standalone and do not change a line of Mooshak original source code.

One way to send multiple files in just one, is to package them with zip or other compress utility. This way it is possible to submit multiple files to Mooshak, without changing it code. Then an extension must be added to unpack, compile and execute this files.

The question that arises is how to call this code extension without changing Mooshak source code. Again relying just on Mooshak default features, there is a way to specify what manages the compile and execution steps of the testing process, on Mooshak's GUI, see figure 2.

In the "**Compile**" field a program or script can be specified to handle compilation. In the "**Execute**" field also a program or script can be specified to handle compilation.

At the "**Compile**" field some Mooshak operation variables are available:

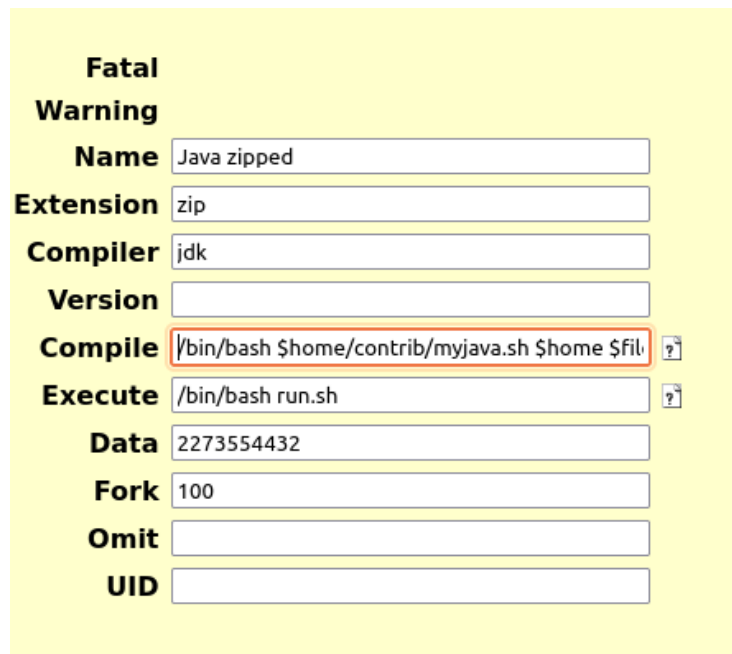
- \$home - Mooshak's home directory
- \$team - id of team submitting the file
- \$file - submission file name (with extension)
- \$name - submission file name (without extension)
- \$extension - extension

- \$environment - Problem environment file
- \$solution - Problem solution

Also at the "**Run**" field some Mooshak operation variables are available:

- \$home- Mooshak's home directory
- \$file - submission file name (with extension)
- \$name - submission file name (without extension)
- \$extension - extension
- \$args - arguments defined for each test
- \$context - context file for each test

Some variables depends on the submission, such as \$file, \$name, \$extension. Others on the test configuration: \$environment, \$args and \$context.



The image shows a web form for configuring language settings in Mooshak. The form has a yellow background and contains several input fields with labels to their left. The labels are: Fatal, Warning, Name, Extension, Compiler, Version, Compile, Execute, Data, Fork, Omit, and UID. The 'Compile' field is highlighted with a red border. The 'Execute' field has a small question mark icon to its right. The 'Data' field contains the value '2273554432'. The 'Fork' field contains the value '100'. The 'Omit' and 'UID' fields are empty.

<b>Fatal</b>	
<b>Warning</b>	
<b>Name</b>	Java zipped
<b>Extension</b>	zip
<b>Compiler</b>	jdk
<b>Version</b>	
<b>Compile</b>	/bin/bash \$home/contrib/myjava.sh \$home \$file ?
<b>Execute</b>	/bin/bash run.sh ?
<b>Data</b>	2273554432
<b>Fork</b>	100
<b>Omit</b>	
<b>UID</b>	

Figure 2: Mooshak language settings

So we have a way to execute a custom script or program in both steps of the testing process. At first step, the compilation, a custom script can unpack a

compressed submission file, expanding all the files in a folder. This way we can send multiple source files without changing Mooshak source code.

It was decided to use a bash script rather than a program to perform the unpacking and compilation, to take advantage of all the operating system tools in a seamless way.

Such a script needs information on the submission, the home directory where the submitted packed file is stored, its file name and extension. Also the \$environment variable is passed:

```
javaext.sh $home $file $extension $environment
```

The environment variable takes the code to be injected. This variable can be set in the problem and test setting page of Mooshak, figure 3

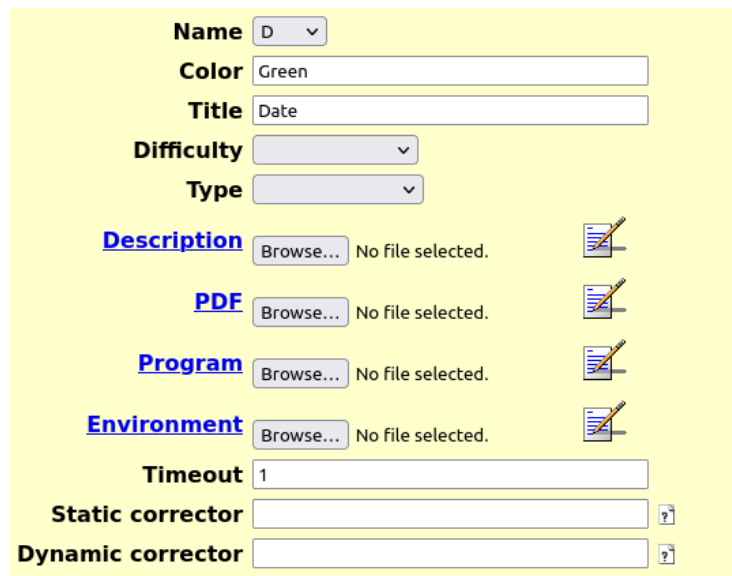
The image shows a web form for configuring a problem and its tests. The form has a yellow background. It contains several fields: 'Name' with a dropdown menu showing 'D'; 'Color' with a text input showing 'Green'; 'Title' with a text input showing 'Date'; 'Difficulty' with a dropdown menu; 'Type' with a dropdown menu; 'Description' with a 'Browse...' button, the text 'No file selected.', and a document icon with a pencil; 'PDF' with a 'Browse...' button, the text 'No file selected.', and a document icon with a pencil; 'Program' with a 'Browse...' button, the text 'No file selected.', and a document icon with a pencil; 'Environment' with a 'Browse...' button, the text 'No file selected.', and a document icon with a pencil; 'Timeout' with a text input showing '1'; 'Static corrector' with a text input and a help icon; and 'Dynamic corrector' with a text input and a help icon.

Figure 3: Mooshak problem and test settings

### 3.1 Code injection

Code injection allows the testing of submitted code against other source code. This can be useful when testing module integration. It can also be useful to inject the source code of a client program to test a submitted library against it.

Here we have 2 options. We can submit the code in the \$environment variable or we can define a path to the folder where the code to be injected is. Current

version of these extensions uses the first option only, but it is expected that future versions allow both options.

The injected code is inserted inside the folder where the submission code was unpacked.

### 3.2 Extending Mooshak to support unit testing

This extension also allows to perform unit testing on Java classes. Currently JUnit4 and JUnit5 are selectable. JUnit4 support files are executed from within the script:

```
junit-4.12.jar  
hamcrest-core-1.3.jar
```

Or for JUnit5:

```
junit-platform-console-standalone-1.6.0.jar
```

To perform JUnit4 tests no additional configuration is needed. However to perform JUnit5 some tweaks are needed, since Mooshak shields execution inside a **"safe execution environment"** that can be configured to use a limited amount of memory and launch a limited amount of threads. This can be configured in the language settings, figure 2, field Fork, which must at least be set to allow 10 concurrent threads.

<b>MaxCompFork</b>	<input type="text" value="10"/>
<b>MaxExecFork</b>	<input type="text" value="10"/>
<b>MaxCore</b>	<input type="text" value="0"/>
<b>MaxData</b>	<input type="text" value="33554432"/>
<b>MaxOutput</b>	<input type="text" value="512000"/>
<b>MaxStack</b>	<input type="text" value="8388608"/>
<b>MaxProg</b>	<input type="text" value="1500000"/>
<b>RealTimeout</b>	<input type="text" value="60"/>
<b>CompTimeout</b>	<input type="text" value="60"/>
<b>ExecTimeout</b>	<input type="text" value="5"/>

Figure 4: Mooshak all languages settings

This can also be configured on the overall language setting, figure 4, where there is more flexibility. Different number of threads can be specified for compilation and execution. Memory limits can be discriminated for code, stack and global data. Here can be configured MaxExecFork to be at least 10. Note however than the **Fork** field in the specific language settings, 2 takes precedence. Also here, the execution timeout can be specified and should be around 5 seconds to the execution of JUnit tests, rather than the usual 1 second limit. The timeout can also be specified in the problem and test setting page, figure 3.

There is yet the need to tell Mooshak that JUnit tests have all passed, or not. Since Mooshak compares the output of the execution with an expected output file, one simple way to do this is to extract from the report of JUnit testing that is printed in the stdout the number of tests that have passed and set the expected output file to hold this number.

### 3.3 Execution and testing

After compiling, it is needed to generate a script to run the compiled application accordingly to be intend to run a default Mooshak black box testing or JUnit testing. This script is called in the **"Execute"** field in the language settings, figure 2. The identification whether it is JUnit or black box testing, can be simply done searching for the existence of a main() function. If the submitted code does not include it, it consists of only classes to be tested with JUnit.

The simplified algorithm of the extended Mooshak operation is shown below.

---

**Algorithm 1:** Extended Mooshak operation (simplified)

---

```

folder ← unpack(submission);
compile(folder);
if JUnit then
    generateRunScript(folder, Junit);
else
    generateRunScript(folder, BlackBox);
end

```

---

## 4 Conclusion

This extension add features to the basic Mooshak programming contest management system towards real word Java application testing. Mooshak allows the testing of only solo source file programs. This might be enough to manage programming contests but it is not adequate to test real

world Java applications, defined in many files and classes in nested packages or namespaces. This extension adds to Mooshak support for testing such complex real world Java applications.

Mooshak default testing method is just black box testing of a solo source file program. However software reliability is also evaluated at the module or unit level. A very popular and useful framework to perform these kind of unit tests in Java source code is Java Unit. This extension gives Mooshak support to JUnit 4 and 5 and also the ability to inject tailored code along with the testable code. All this features gives more flexibility on testing, enhancing Mooshak to support detailed testing of real world Java applications.

Mooshak can be further extended, developing extensions to enhance testing support for other languages, such as C/C++, assembly other software architectures such as Client-server applications and also testing of digital circuits diagrams. Also a more detailed feedback on testing is desirable.

## References

- ICPC. International collegiate programming contest, 2019. URL <https://icpc.global/worldfinals/rules>. [Online; accessed 08-November-2019].
- Junit. Java unit testing framework, 2021. URL <https://junit.org>. [Online; accessed 08-November-2021].
- José Paulo Leal and Fernando Silva. Mooshak: a web-based multi-site programming contest system. *Software Practice & Experience*, 33(6):567–581, 2003. URL <http://www.ncc.up.pt/mooshak/>.