MASS Java Developers Guide

August 31, 2016

1. Intro

This document is to help understand the methods and rules for developing features for the MASS Java library and applications for it.

2. Table of Contents

1. Intro 1
2. Table of Contents 1
3. Setup 1
   3.1. Location of MASS Java and Applications 1
   3.2. Getting a Copy of the Code for Development 1
   3.3. Environment 2
4. Making Changes to MASS 2
   4.1. Pushing Changes 2
   4.2. Building MASS 2
   4.3. Testing Changes 3
   4.4. Debugging MASS 3
5. MASS Components 3
   5.1. Documentation 3
   5.2. Debugger Package 4
6. Creating Applications for MASS 4
   6.1. Creating Applications 4
   6.2. Compiling Applications 5
   6.3. Running Applications 5
   6.4. Debugging Applications 6

3. Setup

3.1. Location of MASS Java and Applications

MASS Java is found in a BitBucket repository at mass_java_core (the repository is private for now). The most current committed code is found in the **develop** branch, so if any work is to be done on the MASS Java library, do it on the code in that branch.

The applications for MASS Java are, again, stored in a BitBucket repository at mass_java_appl (the repository is private for now). As with the code for the library, the most recent code is found in the **develop** branch, so work on the code based in that branch.

3.2. Getting a Copy of the Code for Development

The easiest way to get a copy of the MASS Java code is to clone the repository listed above by either typing

```
$git clone -b develop
      https://bitbucket.org/mass_library_developers/mass_java_core
```

from the command line or by using a Git client like SourceTree.

3.3. Environment

MASS requires the use of multiple computing nodes and it is required that the master node to connect to the remote nodes by using SSH without a password. To do this, a private key needs to be created. If running MASS on a distributed file system, type the following commands into the terminal.

```
$cd ~/.ssh/
$ssh-keygen -t rsa
$mv id_rsa.pub authorized_keys
```

When prompted in the ssh-keygen command, just press enter to accept the default value. Make sure not to include a passphrase, because the MASS code does not allow for one. Just press enter when prompted for one. You can test to see if this works by SSH'ing into a different computer in the cluster.

4. Making Changes to MASS

4.1. Pushing Changes

If you are unfamiliar with using Git, checkout the Git training files found in the *~/Training/GitTraining$* directory on the 320 lab machines for an in depth discussion on how to use it. There are also many resources on the internet to help learn Git.

If you want to add a feature, branch off of the develop branch and commit changes to that branch. When you are finished with your feature, and it is tested, merge your branch to the develop branch and fix any merge conflicts if they show up.

4.2. Building MASS

The core library is structured around Maven. Maven is able to build the library with its dependencies and manage those dependencies. Maven will need to be installed on your computer to build a properly functional JAR.

Many Java IDEs include support for Maven, either natively or with a plugin, and can build the library from there. Refer to the IDE documentation for Maven to learn how to set this up. Maven can also build projects from the command line if the Maven and javac executables are in the system path. Typing

```
$mvn package
```

from the command line in the directory where the *pom.xml* is located will begin the build process. Maven will download any necessary dependencies, compile the classes, and create the final JAR executable. The JAR executable will be located in the *target* directory, even if you used the IDE invocation method.

If you wish to include your most recently built MASS library in an application, type

```
$mvn clean package install
```

to save the newly created JAR in your local Maven repository.

4.3. Testing Changes

Changes in MASS are to be tested with simple MASS applications, which are also Maven projects. The process for creating, compiling and running them are found below.

4.4. Debugging MASS

Included within MASS, is a package named *edu.uw.bothell.css.dsl.MASS.logging* that writes error and debug messages to a file. Currently, only error messages are written to the logs, but debugging messages can be set to be written to the log. Setting the DEFAULT_LOG_LEVEL variable in Log4J2Logger.java to *LogLevel.DEBUG* will allow debug messages to be written to the log files. The log files are written to a text document that can be found in the log/ subdirectory that is created where MASS is run. MASS will create a file for each node, so each node can be debugged separately.

The log files can be quite large and difficult to find useful information in them if the debug messages are set, so it is recommended to run the MASS application with a small number of Places and Agents. Also, if the MASS application is being run somewhere with limited disk space, it is recommended to delete the log files or to move them somewhere else, otherwise Java will crash.

To debug the methods available to the user, the *printOutput* variable in MASS.java can be set too true. This will print status messages to the command line.

5. MASS Components

5.1. Documentation

This project makes use of an automatic documentation generator called Javadocs. It is strongly encouraged to use this method of documentation to keep all the documentation standardized and easily accessible of anyone needing to learn about the methods and classes in this library. The generated documentation of the latest build can be found here.

The documentation is written between the /** and */ symbols at the beginning of a class, method, or variable. Many IDE's will automatically generate this by typing /** and pressing enter above what you want to document. Inside those tags, there is a description of what that class, method, or variable does; there are also other tags that can give descriptions for parameters, return values, or exceptions that can be thrown in the case of methods. Details for the most common tags are found below.

@param *name description*	Describes a method parameter. One for each parameter.
@return *description*	Describes the return value.
@throws *classname description*	Describes an exception that may be thrown from this method.
@author *author_name*	Describes an author
@version *version_number*	Provides software version entry.

This list is not all encompassing and a more complete list and guide can be found here. Leaving entries blank may cause warnings when compiling the project, so if you create a Javadoc comment make sure to fill it out completely and fill out blank ones you come across.

5.2. Debugger Package

Part of the MASS Java library are functions and classes that connect to a debugger application. The debugger is mostly handled by functions in *MASS.java*, but there are also classes contained in the package named *edu.uw.bothell.css.dsl.MASS.MassData* that are used to pass the information about the current state of the **Places** and **Agents** to the debugger application. These classes are **NOT** to be edited, without those changes being reflected in the debugger application. These classes implement *serializable*, and are the classes that are sent to the debugger. If these classes are changed without the changes being made in the debugger, a *StreamCorruptedException* or an *InvalidClassException* can be thrown.

6. Creating Applications for MASS

6.1. Creating Applications

MASS applications are organized as Maven projects just like how the core library is organized and packaged into executable JAR files. To create a new MASS application, navigate to the top level folder in the MASS Java Applications repository and complete the following steps.

1. Enter the following command in the parent level directory

   ```
   $mvn archetype:generate -DarchetypeArtifactId=maven-archetype-quickstart
   ```

   a. When prompted to enter *groupId* use "*edu.uw.bothell.css.dsl.mass.apps"* without the quotes.
   b. When prompted to enter *artifactId*, enter the name of your application. This name should not contain spaces or periods; instead, use a hyphen for separating words e.g. *mass-wave2d, mass-simulation-network-motif, UWCA*.
   c. When prompted to enter *version*, press enter to accept the default *1.0-SNAPSHOT* or enter a desired version but make sure it ends with *-RELEASE* or *-SNAPSHOT* such as *1.0.0-RELEASE* or *0.8.5-SNAPSHOT*.
   d. When prompted to enter package name, the default package name is the same as the *groupId*, but it is necessary to include one more level for the application. The package name should reflect the name of the application, for example if the application name is *mass-wave2d*, a suitable package name would be "*edu.uw.bothell.css.dsl.mass.apps.wave2d"* without the quotes. It is customary to specify package names using lowercase letters, no spaces, or non-letter characters.
   e. Maven will then show a summary of configured options for review. Enter *Y* to confirm or *N* to change a selection.
2. A new sub-directory with the app name will be created by Maven.
3. Copy source files to the newly created subdirectory relative to the new project directory: */src/main/java/edu/uw/bothell/css/dsl/mass/apps/<app name>/*, this folder structure is the same as the package name selected in step 1d.

4. Remove the *App.java* and *AppTest.java* auto generated source files found in the directory above and in the */src/test/java/edu/uw/bothell/css/dsl/mass/apps/<app name>/* directory.
5. Modify the *pom.xml* in the new project directory.
    a. Add *<packaging>jar</packaging>* after the *<artifactId>* element which is not inside the *<parent>* element.
    b. Change the *<name>* element to something more descriptive.
    c. Optionally, add a *<description>* element after the *<name>* element to include a brief description of the application.
    d. Remove the *<url>* element.
    e. Remove the following section

```
1: <dependency>
2:   <groupId>junit</groupId>
3:   <artifactId>junit</artifactId>
4:   <version>3.8.1</version>
5:   <scope>test</scope>
6: </dependency>
```

    f. Add the following section at the end of the XML document, before the closing *</project>* element.

```
1: <build>
2:   <plugins>
3:     <plugin>
4:       <groupId>org.apache.maven.plugins</groupId>
5:       <artifactId>maven-javadoc-plugin</artifactId>
6:     </plugin>
7:     <plugin>
8:       <artifactId>maven-assembly-plugin</artifactId>
9:       <configuration>
10:        <archive>
11:          <manifest>
12:            <mainClass>edu.uw.bothell.css.dsl.mass.apps.
13:                (project package).(main class)</mainClass>
14:          </manifest>
15:        </archive>
16:      </configuration>
17:    </plugin>
18:  </plugins>
19:</build>
```

In the source files, add the package name to each file by adding *package edu.uw.bothell.css.dsl.mass.apps.(project package);* to each source file. When declaring **Places** and **Agents** in the main, use *(class name).class.getName()* in the className field.

6.2. Compiling Applications

Compiling MASS Applications with Maven is similar to how the core library is compiled. In the directory of the application (the same directory with the *pom.xml*) type the following command into the command line and the executable JAR will be found in the *target/* directory.

```
$mvn package
```

6.3. Running Applications

To run your MASS Java application, you will need to include the file, *nodes.xml*, in the same directory you are running the application from for MASS to run. This file is an XML document that passes information about the computing nodes you wish to be using to run your MASS

application on. If you do not have this file, MASS will not initialize causing your application to fail. An example of this file is shown in the code sample below.

```
1:      <nodes>
2:          <node>
3:              <master>true</master>
4:              <hostname>Node0</hostname>
5:              <javahome>/usr/bin/</javahome>
6:              <masshome>~/MASS_Application/</masshome>
7:              <username>Node0UsernameHere</username>
8:              <privatekey>~/.ssh/id_rsa</privatekey>
9:          </node>
10:         <node>
11:             <hostname>Node1</hostname>
12:             <javahome>/usr/bin/</javahome>
13:             <masshome>~/MASS_Application/</masshome>
14:             <username>Node1UsernameHere</username>
15:             <privatekey >~/.ssh/id_rsa</privatekey >
16:         </node>
17:     </nodes>
```

This sample defines two nodes, one at Node0 and the other at Node1. Each element is described as follows:

<nodes>	This element contains each node element that the MASS application will run on.
<node>	This element contains information about a specific node that the MASS application will run on. The number of these elements will determine how many computing nodes will be used i.e. if there were eight **<node>** elements, the application will run using eight computing nodes.
<master>	A boolean which is true if that node is the master node or the node that the application was initially run. Only required for the master node.
<hostname>	The hostname or IP address for this node.
<javahome>	The location of the *java* executable. If the *java* executable is part of the system path, this element can be left blank.
<masshome>	The location of the application JAR executable.
<username>	The username for a profile to log onto this node.
<privatekey>	The location of the key used to SSH into different computing nodes. If the technique above was used to generate a key, use *~/.ssh/id_rsa* in this field.

To run the executable, type the following commands in the command line.

```
$java -jar <MASS App Name>.jar <arguments>
```

A few applications have UI components, so if you are running the application remotely, you will need to enable X11 forwarding. If you are using SSH, you will have to use the *-X* flag when entering the SSH command. If you are using PuTTY, you will need to download and install Xming, and check the *Enable X11 forwarding* setting in *Connection/SSH/X11*.

6.4. Debugging Applications

The best tool to use to debug MASS application is the MASS debugger. This application was developed as a way to see the data inside **Places** and **Agents** and pause the application at any

time. The MASS application needs an extra two lines in the main to initialize the MASS end of the debugger as well as two extra functions added to both of the overridden **Place** and **Agent** classes. The debugger GUI, found here, has to be run separately at the same time as the MASS application. More information about the debugger can be found in the MASS Java Specification and the MASS Debugger (Java) Specification.