Apparatus Developer’s Guide

Jennifer Dempsey, Andrew Emmott, Andrey Kvochko,
Amit Shrestha, Richard T. Snodgrass, and Robert Trame

May 30, 2014

TD-12

A Tau Technical Document
LoCuS (Laboratory for CompUter Science) System is instructional software for teaching computer science theories through experiments. Experiments within the context of LoCuS are laboratory excursions akin to those found in other sciences like biology, chemistry, and physics. A locus (Latin for “place”) is a “collection of points which share a property” or “the path through which a point moves to satisfy a given condition.” The LoCuS logo is a Boy’s surface, named after its founder, Werner Boy. Boy’s surface is a particularly intricate locus which is described by a system of complex equations. Just like the Mobius strip, Boy’s surface has only a single side. Our goal is for LoCuS to define a new place to emphasize the science of computation as well as a new path for computer science, in the form of labs.
Contents

1 Overview 1
  1.1 Apparatuses in Labs .............................................. 1
  1.2 Process of Constructing an Apparatus ......................... 1
  1.3 Apparatus Options ............................................. 2
  1.4 Testing Process ............................................... 2

2 Developing an Apparatus 3
  2.1 Naming Conventions ........................................... 3
  2.2 Directory Structure ........................................... 3
    2.2.1 The Descriptor File ..................................... 4
    2.2.2 The Ant Script ........................................... 5
    2.2.3 Directories ................................................ 5
    2.2.4 The Default Options File ................................ 5
    2.2.5 Log Properties File ..................................... 5
    2.2.6 Other Apparatus Resources ............................... 6
  2.3 The Primary Apparatus Class ................................ 6
    2.3.1 Logging .................................................. 6
    2.3.2 Core Classes ............................................. 6
    2.3.3 Model-View-Controller .................................. 7
  2.4 Implementing an Apparatus Subclass ......................... 7
    2.4.1 Abstract Methods ........................................ 8
    2.4.2 Requestable ............................................... 9
    2.4.3 Apparatus Utility Methods ............................... 9
  2.5 Options for an Apparatus .................................... 10
    2.5.1 Defining Expected Options ............................... 10
    2.5.2 Defining Default Options ................................ 11
    2.5.3 Defining Attributes ..................................... 11
  2.6 Creating HTML Tabs .......................................... 11
  2.7 The Apparatus View ........................................... 12

3 Additional Functionality 13
  3.1 The Socket Framework ..................................... 13
    3.1.1 Reading From a Socket .................................. 14
    3.1.2 Writing to a Socket ..................................... 14
    3.1.3 Socket Implementation Guidelines ..................... 15
  3.2 Controlling the Chalkboard ................................. 15
  3.3 Accessing Resources ......................................... 15

4 Testing 17
  4.1 Standalone Testing ........................................... 17
  4.2 Testing Within a Lab ......................................... 17
  4.3 Interactions with WATCHER ................................. 18
Chapter 1

Overview

LoCuS supports many labs that are written to teach computer science theories. One of the key aspects of these labs are interactive features in the form or games, tools, and demonstrations of these theories, known as apparatuses. An apparatus is used to give students a clearer idea of the principles behind a computer science theory. In this technical document we will describe the steps necessary to create and test one of these apparatuses and detail the components that are involved in this process.

1.1 Apparatuses in Labs

In order to integrate an apparatus into a lab, the apparatus must be specified in the XML which is described in Section ?? and more information can be found in the “Lab Author's Guide” Technical Document. Once the XML for the apparatus is in place LoCuS will parse the XML for the apparatus and create a button in the lab that will open the apparatus in a separate window in front of the lab. In order to accomplish this the apparatus must extend the Apparatus.java class which is a abstract class that allows the apparatus to communicate with the lab as well as with other apparatuses.

1.2 Process of Constructing an Apparatus

The process for creating an apparatus plugin for LoCuS has been designed to be as simple and straightforward as possible. By utilizing the plugin framework and the XML specifications outlined in this document, one should be able to rapidly deploy robust apparatuses which can be interfaced with LoCuS labs and with other apparatuses. This document explains the process of creating an apparatus plugin by using HelloWorld apparatus as an example. Before reading this document, please read the Getting Started Guide for the LoCuS Project.

The construction of an apparatus involves the following steps.

1. Name the apparatus (see Section 2.1).
3. Arrange all necessary resources in the ./resources folder (enumerated in detail in Section 2.2).
4. Implement a subclass of the Apparatus class (discussed in detail in Sections 2.3 and 2.4).
5. Define default options for standalone runs (discussed in detail in Section ??).
6. Create all necessary HTML files for use in information tabs (discussed in detail in Section 2.6).
7. Some apparatuses have additional functionality, discussed in Chapter 3.
1.3 Apparatus Options

Each apparatus can be designed for use across multiple labs and through different grade levels. In order to accommodate this functionality an apparatus can make use of XML options in order to make an apparatus more customizable to fit a wider variety of situations. For example options can be used to disable certain buttons that are not needed for a lower-level lab in order for the apparatus to be easier to use, or it can be used to specify certain characteristics of an apparatus. In the HelloWorld Apparatus options are used to designate the number of explanation points that will be added each time a button in the apparatus is clicked.

For more information on how to implement and use options in an apparatus refer to Section 2.5.

1.4 Testing Process

Before an apparatus is used in a public deployment, it should be thoroughly tested using the following steps.

1. Set the options which you plan on initially testing.
2. Invoke the apparatus as a standalone.
3. Invoke the apparatus within a lab.
4. Consider all options that would be needed within a lab and test them.
5. Consider any interactions between the lab and the apparatus specifically how the apparatus should handle errors in XML that specifies the apparatus in a lab.
Chapter 2

Developing an Apparatus

Apparatuses are distributed as plugins to the main LoCuS platform. A plugin is a separate class that can be loaded by LoCuS and then interact with the environment. A plugin is not strictly part of LoCuS. In order to ensure that LoCuS can communicate with plugins, each plugin must satisfy several conditions, which are discussed in detail below. The easiest way to start developing an apparatus is to copy and modify the existing HelloWorld apparatus. We will go through the structure of HelloWorld, pointing out what should and should not be modified during development process, to create a new apparatus.

2.1 Naming Conventions

Apparatuses should be named as “descriptive noun + function classification”. For instance, the descriptive nouns could be along the lines of “Calc”, “Color”, “Sequence”, and “Plot”. These descriptive nouns would always be in the singular form. Function classifications could be as follows.

**Tool**: Apparatus that can be used in concert with others, e.g. calculator or plotter.

**Demo**: Apparatus that is used to demonstrate a principle without necessary challenge, just adjustable parameters, e.g. sequencer or color-by-number.

**Game**: Apparatus that demonstrates a principle in a game-like fashion. This could include objectives, distinct challenge, and parameters that are adjusted without intervention from (but visible to) the user. E.g. simon or tower-builder.

Using this convention, we end up with names like ColorGame, PlotTool and SimonToy. This convention allows maintainers to easily identify both the distinct form of the apparatus and its functionality.

2.2 Directory Structure

The HelloWorld apparatus satisfies all the requirements for the apparatus structure and exemplifies all the functionality available to apparatuses.

Let’s start with the overview of HelloWorld project structure. At the root level there are the following.

- **Files**:
  - plugin.xml
  - build.xml

- **Directories**:
  - src
  - dist
2.2.1 The Descriptor File

The plugin.xml file describes key properties of a plugin, such as plugin’s ID, main class, resources folder location and more. It is called a descriptor file. Let’s look at the descriptor file for the HelloWorld apparatus. Modify this to create a descriptor file for your apparatus. Specifically, you will modify lines 3, 8, 13, 14 and 15, as described below.

```
<?xml version="1.0" ?>
<!DOCTYPE plugin PUBLIC "+//JPF//Java Plug-in Manifest 1.0" "http://jpf.sourceforge.net/plugin_1.0.dtd">
<plugin id="HelloWorld" version="1.0">
  <requires>
    <import plugin-id="edu.arizona.cs.locus.plugins.apparatus.core"/>
  </requires>
  <runtime>
    <library id="HelloWorld" path="/" type="code">
      <export prefix="*"/>
    </library>
    <library type="resources" path="resources/" id="resources"/>
  </runtime>
  <extension plugin-id="edu.arizona.cs.locus.plugins.apparatus.core" point-id="Apparatus" id="HelloWorld">
    <parameter id="class" value="edu.arizona.cs.locus.plugins.apparatus.helloworld.HelloWorld"/>
    <parameter id="name" value="Hello World"/>
    <parameter id="icon" value="icon.png"/>
  </extension>
</plugin>
```

Listing 1: HelloWorld’s plugin.xml file

- Lines 1 and 2 are required and should not be changed.
- In Line 3 we declare that plugin ID is HelloWorld and that the version is 1.0. The purpose of a plugin ID is to be able to identify the plugin from the lab’s XML file. You should modify this line to have a unique id, which is usually a CamelCase name of the apparatus, without spaces or delimiter symbols. For more on apparatus naming see Section 2.1 Each apparatus has a major and a minor version number.
- Line 5 says that this plugin requires the core package. This package contains all the functionality provided to an apparatus by the LoCuS framework, as well as abstract classes that an apparatus must extend (discussed in more detail below). This line should not be modified.
- In lines 8–10 we tell the plugin system that all classes plugin’s classes are visible to LoCuS platform.
- Line 11 sets the resources folder location to the resources directory. Do not call it anything else.
- Line 13 says that this plugin extends the core plugin: the one which is used to communicate with LoCuS environment. point-id attribute is the name of extension point in the core plugin. The value of both plugin-id and point-id attributes should not be modified.

The id attribute, on the other hand, must be modified. It specifies the ID of the current extension and must be equal to the plugin’s ID, as required by the LoCuS platform.

Child nodes of this element specify:
i the main class of the plugin (should be modified to point to your class),
ii plugin's name (should be modified)
iii plugin's icon (should not be modified, but the icon itself should be changed).

2.2.2 The Ant Script

The build.xml file is an Ant script, which is used to build the project. It compiles the plugin and automatically places it into LoCuS' plugin directory for you.

This file is quite long, but only one line that should be changed, the first one.

```
<project name="HelloWorld" basedir="." default="main">
```

You should change the name attribute to your apparatus' name.

2.2.3 Directories

The src folder is required for every apparatus and contains the source code for the apparatus. We will discuss the structure of the source code in the next section.

The dist folder is where you can find the compiled jar file.

The docs directory is where the Ant script places javadoc files.

The lib directory is where the libraries (external .jar files) for your apparatus should go. For the HelloWorld apparatus, this folder is empty because it doesn’t use any external libraries. The SocialNet apparatus uses the prefuse.jar library.

The resources is where the resources specific to the apparatus should go. This folder should always have a default options file and a logger options file as described in subsections 2.2.4 and 2.2.5 respectively.

2.2.4 The Default Options File

The resources folder of an apparatus must contain the file defaultoptions.xml. This file is used when the apparatus is invoked as a standalone or when no options are given in the lab XML where the apparatus is designated. When testing an apparatus as a standalone the options in this file should be altered in order to test all options that may be needed within a LoCuS lab.

Below is an example of the defaultoptions.xml for the HelloWorld Apparatus. This apparatus only uses a single option that takes an integer and is used to determine the number of exclamation points that are added when a student clicks a button within the apparatus.

```
<options>
  <numExclamations>3</numExclamations>
</options>
```

2.2.5 Log Properties File

The resources folder of an apparatus must also contain a log4j.properties file. This file should be the exact same for every apparatus in order to maintain consistency. To insert this file into a new apparatus you can copy the existing file from the HelloWorld apparatus into your own resources folder.
2.2.6 Other Apparatus Resources

The resources is also where the resources specific to the apparatus should go. Note that only the resources that are static should be placed here. In other words only resources where the same version will be used across all labs. Some examples of resources, other than the ones mentioned above, that should go into this folder are as follows.

- A picture of a check mark for Accept button
- A texture for the apparatus main form’s background

Resources that should not go into this folder include the following.

- A picture of a dinosaur for coloring apparatus.
- A file describing relations between people for socialnet apparatus.

2.3 The Primary Apparatus Class

Remember the class that is pointed to by Line 14 of HelloWorld’s plugin.xml. This class is called the primary apparatus class. Recommended implementation suggests that the primary apparatus class should not be referenced by any other class in the apparatus itself, as it is intended purely to initialize the apparatus. Let’s go through the primary apparatus class of HelloWorld in detail. We start with the import portion.

```java
package edu.arizona.cs.locus.plugins.apparatus.helloworld;
import java.awt.Dimension;
import java.util.Scanner;
import edu.arizona.cs.locus.core.Logger;
import edu.arizona.cs.locus.plugins.apparatus.core.*;
import edu.arizona.cs.locus.plugins.apparatus.helloworld.model.HWDetailModel;
import edu.arizona.cs.locus.plugins.apparatus.helloworld.model.HWMainModel;
import edu.arizona.cs.locus.plugins.apparatus.helloworld.view.HWDetailView;
import edu.arizona.cs.locus.plugins.apparatus.helloworld.view.HWMainView;
```

Listing 2: HelloWorld apparatus: the import section

2.3.1 Logging

At Line 5, we import the Logger class, which is a wrapper class around the log4j Logger class. It is important to use this class to output debug and error messages, rather than using java.util.Logger or System.out.println because with log4j you can specify where you want to output debug messages. Right now there is a new feature in development, which will send error messages to the web to give developers more insight on how LoCuS behaves on different client machines and how to improve robustness of the application. Using log4j will ensure that the team receives such error messages. Logging is discussed in more detail in the Tau Getting Started Guide.

2.3.2 Core Classes

Line 8 imports required classes from the core plugin. Apparatus class is the one that the platform uses to initialize, display and manage the apparatus. The primary apparatus class must extend the Apparatus superclass. This should be done via a primary class named for the apparatus as a whole (e.g., ColorGame for Color Game, LocalityAnalyzer for the Locality Analyzer, cf. Section 2.1).

Although there are many methods in the Apparatus class (see the javadoc), implementors of subclasses need pay attention to only those shown in the UML given in the Appendix.
2.3.3 Model-View-Controller

Lines 10–13 indicate that the apparatus implements the model-view-controller design pattern. It is recommended apparatuses are written using this pattern. The controller can (and in most cases, should) be integrated into the view, but the model should be separate from both the primary apparatus class and the view.

Maintaining a model-view-controller design may seem like a daunting task, but this requirement goes a long way toward ensuring a solid, flexible design. The HelloWorld apparatus is over-designed, but larger apparatuses would do well to follow its example. While a walkthrough follows, it is highly recommended that you explore the coreplugin and HelloWorld projects on your own and make sure you understand the flow of information and control. Note that there is a model class for each view, as exemplified by the two model classes in HelloWorld.

A brief summary of the design pattern is as follows:

• Models contain information and send that information to views.
• Views receive information from models, interpret that information, and present it to the user. Optionally, views may send messages from the user to a controller.
• Controllers receive messages from views and interpret these messages to pass on to the models. Within the apparatus architecture, controllers are implemented within the views, usually as an internal Listener class.

The apparatus, e.g. HelloWorld, is expected to have, at minimum, the following three packages (all specified in lower case).

• edu.arizona.locus.plugins.apparatus.helloworld
  This contains the main class for the plugin, which generally implements the main() method, as discussed shortly.

• edu.arizona.locus.plugins.apparatus.helloworld.model
  This contains all of the classes required for the model, storing the state with getters and setters.

• edu.arizona.locus.plugins.apparatus.helloworld.view
  This contains all of the classes required for the view, generally extending JInternalFrame, as discussed in Section 2.7

2.4 Implementing an Apparatus Subclass

At line 14 we define the primary apparatus class which, as mentioned above, extends Apparatus class. The primary apparatus class, in this case, HelloWorld.java, usually has two methods: main() and initialize().

```java
14 public class HelloWorld extends Apparatus {
15     private HWMainModel hwm;
16
18     public static void main(String args[]) {
19         HelloWorld hw = new HelloWorld();
20         hw.startApparatusStandalone(args);
21     }
```

The main() method at line 18 starts the apparatus as a standalone application. While the ability for standalone operation is not required, testing and debugging are much easier if the main() method is implemented. This method should exist in the Apparatus’ subclass, but the actual standalone functionality is built into the superclass. The main() method should only call the constructor for the subclass, and call the startApparatusStandalone() method.
The `startApparatusStandalone()` method does the majority of the work in terms of creating windows, etc. Sequentially, this method calls `setOptions()`, followed by your implementation of `initialize()`, to be described next.

When LoCuS launches an apparatus, it constructs the apparatus, and then calls the `initialize()` method, described next. An apparatus should *not* have a constructor.

### 2.4.1 Abstract Methods

There is only one abstract method in the `Apparatus` superclass: the `initialize()` method.

```java
public void initialize() {

Listing 3: HelloWorld’s implementation of `initialize()` method
```

This open-ended `void` method is where the code begins. Subclasses of `Apparatus` should not have a constructor, as the `initialize()` method functions as one, as well as a `main()` method (except when running standalone). The primary responsibilities of this method are as follows.

1. Instantiate the logger. Name your apparatus in the string.

   ```java
   // initialize Logger
   ```

2. Construct your model-view-controller objects.

   ```java
   // initialize Model and View
   HWMainModel hwm = new HWMainModel();
   HWDetailModel hwd = new HWDetailModel();
   HWMainView hwmv = new HWMainView(hwm, hwd);
   HWDetailView hwdv = new HWDetailView();
   hwm.addObserver(hwmv);
   hwd.addObserver(hwdv);
   ```

3. Parse the XML options given. XML options are discussed in detail in Section ??.

   ```java
   // parse options
   int numExclamations = 0; // Default of 0.
   try {
     numExclamations = Integer.parseInt(options.getText("numExclamations"));
   } catch (Exception e) {
     logger.error(e, "Hello World option \"numExclamations\" missing or malformed.");
   }
   ```

4. Set options using public calls.

   ```java
   // set options using public setters
   hwm.setExclamations(numExclamations);
   ```
5. Initialize sockets. The socket framework is discussed in detail in Section 3.1.

6. Add your views to the apparatus frame.

2.4.2 Requestable

At lines 52–53 you might have noticed the requester object. This is how apparatuses communicate with the main LoCuS application. requester is defined in the Apparatus class, core package. This object is of type Requestable, which is an interface implemented by the LoCuS framework. Please refer to the UML in Appendix A for details on the structure, and to the Javadoc for a listing of methods available in the Requestable interface.

2.4.3 Apparatus Utility Methods

The Apparatus class provides a number of apparatus utility methods, some of which are available to all classes, whereas others are only available to its subclasses. The majority of these should not be overridden unless very specific functionality is desired. Even then, there may be another way to implement said functionality, which would be preferable.

The HelloWorld apparatus overrides several utility methods, which in general will be overridden by all apparatuses.

Listing 4: Apparatus utility methods in HelloWorld
The `getAppSize()` method sets the size of the apparatus frame. `getName()` provides the name of the apparatus to the LoCuS platform. Two other methods are used to define expected options.

### 2.5 Options for an Apparatus

When launched by a lab, an apparatus is provided an XML document which defines options which the apparatus should utilize. For the `HelloWorld` apparatus the options appear as they do in Figure 2.5 within the XML for the lab.

Note that Figure 2.5 also shows the attributes required in the apparatus tag. The `name` attribute must be given the name of the apparatus, which is the same as the name of the jar file for that apparatus. Note that all apparatus names should contain only lowercase letters and no spaces. In addition the `caption` attribute is the text that will appear on the button in the lab to launch the apparatus. This attribute can be any string.

```
<launchApparatus>
  <apparatus name="helloworld" caption="Hello World">
    <options>
      <numExclamations>3</numExclamations>
    </options>
  </apparatus>
</launchApparatus>
```

Listing 5: The XML that is used to specify a version of the `HelloWorld` apparatus that will add 3 exclamation points to the text.

All of the options are contained within the `options` element. Various types of options are handled in different ways. Options can have attributes. Some of the options can have data, e.g. a string for the apparatus to display or process. The options an apparatus expects are documented in each Apparatus Specification (see TD–17 for the “`HelloWorld` Apparatus Specification”).

#### 2.5.1 Defining Expected Options

Primarily to facilitate debugging and to allow for some kind of validation across apparatuses, implementations of subclasses of the `Apparatus` class are expected to define the set of options that it can expect to process. An apparatus should not crash at run-time if it is given inadequate or inappropriate options. An apparatus will supply default values for options that are undefined (defaults are explained in the apparatus specification) and will silently ignore options that are not relevant. The primary reason for defining expected options is to facilitate debugging and to make it obvious when there are inconsistencies between lab XML files and source code.

Note that in case any option is left out by a lab author all options should have either a default value, or an appropriate error message should be displayed when attempting to open the apparatus. Due to the fact that LoCuS runs as a full screen application this message should be displayed as a pop up so that when testing this error can easily be recognized.

The options that are defined within an apparatus can vary between apparatuses and should be clearly defined within a technical document called an Apparatus Specification. As an example see the “`HelloWorld` Apparatus Specification”.

This task entails overriding two methods from the superclass: `getRequiredOptionElements()` and `getOptionalOptionElements()`. These methods describe to the structure what options are relevant, and which aren’t by returning `String` arrays, each representing a list of potential option names. As above, do not use instance variables in these methods; simply return array literals. Required options are options that must exist each time the apparatus is launched. Optional options are those that may exist and should be used by your apparatus if they are present. These lists do not need to overlap. Options not on these lists will be ignored at launch.
protected String[] getOptionalOptionElements() {
    return new String[] {
        "allCaps",
        "file"
    };
}

protected String[] getRequiredOptionElements() {
    return new String[] {
        "numExclamations"
    };
}

Listing 6: Defining expected options in HelloWorld

2.5.2 Defining Default Options

The UML class diagram in the appendix does not reflect how options are given during the standalone run. Given no arguments, your apparatus will search for a file named \texttt{defaultoptions.xml} in your resource path. If a valid option node is within this file, this node will be used at launch. Alternatively, if a command-line argument is given, the first argument will be interpreted as a file name from which the launch options should be read. The \texttt{HelloWorld} project comes with several option files—try running the apparatus specifying one of these other filenames and see what happens (the LoCuS Getting Started Guide describes how to set this filename in Eclipse).

\begin{Verbatim}
<options>
  <numExclamations>3</numExclamations>
</options>
\end{Verbatim}

Listing 7: HelloWorld's \texttt{defaultoptions.xml}

2.5.3 Defining Attributes

XML attributes are used to further describe the apparatus. Note that any item that is used as an attribute could be alternately listed as an option instead. When considering the use of an attribute it is important to note that attribute cannot contain multiple or nested elements. This makes it more difficult to expand or alter an attribute for future use.

In LoCuS attributes are commonly used to provide names for an apparatus or other bookwidth. It is unlikely that the name element will need to list multiple elements or later be changed making the use of an attribute appropriate.

Note that the attribute values must always appear in quotes. Either single or double quotes can be used. The example below demonstrates how the name attribute might appear in an apparatus.

\begin{Verbatim}
<LaunchApparatus name="helloworldexample">
  <options>
  </options>
</LaunchApparatus>
\end{Verbatim}

Listing 8: Example of the name attribute

2.6 Creating HTML Tabs

If you want any informational tabs present in your apparatus, you can write HTML documents to this effect. Creating these is up to you, but displaying these documents in your apparatus is easy. As above, there is one method which needs to be implemented: \texttt{getHTMLPanelNames()}. Important to note is that the name of the file is what gets used as the title of the tab after dropping \texttt{.html}. The files themselves are expected to be found in \texttt{html} subdirectory of the \texttt{resources} folder.
Listing 9: HelloWorld’s implementation of `getHtmlPanelNames` method

These tabs get made at construction and do not require any additional effort on your part. The rationale for defining HTML files in the `getHtmlPanelNames()` method and not just creating tabs for everything in the `html` directory is that you may want to have tabs link to other HTML files which do not require their own tabs.

### 2.7 The Apparatus View

Listing 10: HelloWorld `HWMainView` class declaration.

The Apparatus View consists of at least a single `MainView` which extends `javax.swing.JInternalFrame` and implements `Observer`. When views receive `Observables` via the `update()` method, they are expected to cast them to the appropriate model-interface (not the class itself), and thus only use the model’s accessors. Views are expected to take controllers in at construction.

Controllers should (usually) be implemented as internal classes of the views. They are to receive messages from views (similar to `Observer’s` `update()`). Controllers are assumed to contain direct references to relevant models, ideally taken in at construction. Controllers may mutate Models any way they see fit. How the internals of the view are implemented is completely up to the implementer.
Chapter 3

Additional Functionality

The core package provides many utilities for you to use in your own apparatus. It is important for you to be familiar with these so that you don’t duplicate code. These utilities can be classified as involving sockets (Section 3.1), controlling the chalkboard (Section 3.2) and resources (Section 3.3).

3.1 The Socket Framework

Apparatuses communicate to each other by the means of sockets. The Requestable interface, available to apparatuses via the Apparatus superclass, provides two methods to create sockets.

ApparatusInSocket registerInSocket(Apparatus, String, Type) : This method registers an input socket and returns a reference to an input socket object.

ApparatusOutSocket registerOutSocket(Apparatus, String, Type) : Registers an output socket and returns a reference to it.

Interfaces ApparatusInSocket and ApparatusOutSocket are discussed in more detail below. Both of these methods take the same arguments, which are as follows.

1. Apparatus appRef takes a reference to the object of primary apparatus class.

2. String socketName is a name of a socket. The name usually describes the data that will be sent via the socket. This name is used when designing connections in the lab XML file.

3. Type type is a type of data that will be transmitted through the socket. The type can be anything, including primitive types (e.g. boolean, which will be auto boxed into a class type, e.g. Boolean), class types, and arrays (which also get autoboxed).

In the Listing 11, the HelloWorld apparatus declares two sockets.

HelloWorld class:

```java
try {
    hwm.setBooleanOut(requester.registerOutSocket(this, "helloworldbooleanin", boolean.class));
    hwm.setBooleanIn(requester.registerInSocket(this, "helloworldbooleanout", boolean.class));
} catch (NullPointerException e) {
    HelloWorld.logger.error(e, "Requester is not available");
}
```

Listing 11: HelloWorld registering input and output sockets.

The setInSocket and setOutSocket methods should be implemented in the model class, so that the model can send and receive messages over sockets. These two methods are very simple.
HWMainModel class:

```java
private ApparatusInSocket inSocket;
private ApparatusOutSocket outSocket;

public void setBooleanIn(ApparatusInSocket socket) {
inSocket = socket;
inSocket.addObserver(this);
}

public void setBooleanOut(ApparatusOutSocket socket) {
outSocket = socket;
}
```

Listing 12: HelloWorld registering input and output sockets

It is important to note that all connections functionality is handled entirely by LoCuS platform and thus is not available to apparatuses launched in standalone mode.

3.1.1 Reading From a Socket

Interface `ApparatusInSocket` is used by apparatuses to receive data via connections mechanism. To instantiate a socket, one must use the `registerInSocket()` method of apparatus’ `requester` object.

This interface extends `Observable`. Whenever there is new data available at the input socket, the `update()` method gets invoked. The data can then be read by using a `read()` method. This method returns an object which was sent to a connected output socket. Although the return type of `read()` method is `Object`, the model always knows what type of object is read from the socket: the type is explicitly specified by the `registerInSocket()` method. The input socket will not accept the data of a different type, thus explicit conversion from `Object` to the required type will always succeed.

```java
@Override
public void update(Observable o, Object arg) {
    threadPool.execute(new Runnable() {
        @Override
        public void run() {
            try {
                if ((Boolean)inSocket.read())
                    setExclamations(numExclamations() + 1);
            } catch (Exception e) {
                HelloWorld.logger.error(e);
            }
        }
    });
}
```

Listing 13: HWMainModel class. Reading from the socket.

3.1.2 Writing to a Socket

Interface `ApparatusOutSocket` is used by apparatuses to send data via connections mechanism. Just like with `ApparatusInSocket` interface, output sockets must be created by `registerOutSocket()` method of apparatus’ `requestable` object.

This class implements a `write(Object obj)` method, where `obj` is the object that should be sent via the output socket.
HWMainModel.java:

```java
70 public void sendExclamation() {
71     try {
72         if (outSocket != null)
73             outSocket.write(true);
74     } catch (Exception e) {
75         HelloWorld.logger.error(e);
76     }
77 }
```

Listing 14: Writing to a socket.

### 3.1.3 Socket Implementation Guidelines

It is suggested that sockets are registered at the launch of the apparatus (in the `initialize()` method of the primary class). References to registered sockets can be stored either in the primary class or in the model. It is also suggested that the data coming from the socket is received by apparatus’ model. For this, the model should be registered as an observer for the `ApparatusInSocket` object (see Listing 8).

Although the LoCuS platform ensures that the `read()` method doesn’t block when attempting to read from an empty input socket, one should consider reading the data in a separate thread for a better reliability of the system, as well as increased performance (see Listing 9).

### 3.2 Controlling the Chalkboard

Apparatuses can control the contents displayed on the LoCuS chalkboard by utilizing the following methods of the `Apparatus` class: `appendToChalkboard(String)` and `overwriteChalkboard(String)`. These methods will not cause an exception crash when called in a standalone mode.

### 3.3 Accessing Resources

There are two types of resources that an apparatus can have access to.

1. Local resources that are located in apparatus’ `resources` folder. These resources must be read only by using modifications of one of the two lines: `Apparatus.readJarFile` or `Apparatus.requester.readJarFile`.

2. Resources in a lab’s `.zip` file. These resources should be accessed by using `Apparatus.readLabFile()` or `Apparatus.requester.readLabFile()` methods.

You should not be using any other method for accessing external files.
Chapter 4

Testing

Testing your apparatus should be done in two distinct phases.

4.1 Standalone Testing

The first phase should be to run your apparatus as a standalone application as discussed in Section 2.4. This executes the main method in the subclass of Apparatus. Testing in this way will ensure that your apparatus behaves as expected in and of itself. It should test all GUI aspects and underlying behavior. This form of testing will not be able to test interactions with other apparatuses (such as sending to/receiving from another apparatus via a socket). Interactions with the chalkboard and LoCuS lab environment are also not supported.

When conducting this first phase of testing defaultoptions.xml should be created within the resources folder and used to test all possible combinations of available options. Note that the default options should be tested and all available options including the defaults should be well documented in the specification for the apparatus.

4.2 Testing Within a Lab

The second phase of testing is integration with LoCuS. At this point, you should run your apparatus through the testing lab as described in the LoCuS Getting Started Technical Document. To do this, create a minimal testing lab with the following code. You will need to provide apparatus-spec that includes all available options and their default values.
Listing 15: XML that is used to display the HelloWorld Apparatus.

This will allow you to test your apparatus’ ability to receive directions from the lab, its ability to interact with the chalkboard, and its ability to send/receive data to/from another apparatus. Listing 4.2 should a minimal lab with the XML for the HelloWorld apparatus is contained in the lab. Note that there is currently a minimal lab, HelloWorld.xml that can be used for this purpose.

4.3 Interactions with WATCHER

The WATCHER feature is implemented in LoCuS but not in individual apparatuses. Upon opening WATCHER will send information to the LoCuS database to indicate that the apparatus has been opened.

For more information on how WATCHER works see the “Watcher Feature Guide” Technical Document. Note that the functionality of WATCHER could be extended to send back specific information about the results of a particular apparatus.
Appendices
Appendix A

HelloWorld UML