Populus Guide for Developers

Lars Roe

May 13, 2015

Contents

Part I
Overview

This document was written as a primer for anyone interested in developing or modifying the Populus source code. This file is written in roughly in the order that you would want to understand the code, but feel free to jump around as needed.

Part II
Setting Up New Development Machine

1 Development Software

1.1 Java Development Kit

Download and install Java SE (Standard Edition) from oracle.com. Please use JDK 1.7.

1.2 Git

Git on the command line should be default for OS X and Linux. You don't have to install more, but `http://git-scm.com/downloads/guis` has some nice GUIs. I used GitX-dev (rowanj) for OS X, which seems good.

For Windows, I like git for Windows: `http://msysgit.github.io/`, which includes the command line tools and GUI.

1.3 Eclipse

Download the Eclipse Standard from `https://www.eclipse.org/downloads/`. (You can actually install any version that has Java support.) The "installer" is just a zip file that you extract somewhere. You'll run it by running the executable in there.

1.4 TeX

We don't use LaTeX for any externally-facing file, but it is used for modifying this document. I use MiKTeX for Windows. MacTeX and livetex are recommended for OS X and Linux respectively.

1.5 Photoshop

Use Photoshop to make the pictures for, say, the Web page. There are saved `.psd` files around that contain the source image to work from with its Layers. You don't really need to use it as a developer. In the past, Don created this.

2 Populus-specific Setup

2.1 Files

Here's how you can check out the files from git. Assume that the `.git` directory is at `C:/TEMP/pop.git` and you want to put the code into `workspace/pop` relative to your current directory.

```
mkdir workspace
cd workspace
git clone file:///C:/TEMP/pop.git pop
```

The "`pop.git`" directory I'm using as an example should be a directory that contains the `.git` directory. I assume we'll have a real central repository soon, and then this guide can be updated to that location.

2.2 Running Eclipse

Now run Eclipse. For the workspace, choose the `pop` directory, or whatever you used to extract the files from git in the previous step.

Be sure that you are using an installed JDK for the workspace (`Windows` ⇒ `Preferences` ⇒ `Java` ⇒ `Installed JREs`)

Go to `File` ⇒ `New` ⇒ `Java Project`

For the project name, choose `PopulusE`. Eclipse should know that this is an existing project, and don't set any more options.

Click the green run button. You want to run this as a Java Application. The main class is `PopRun` (`edu.umn.ecology.populus.core.PopRun`).

2.3 WindowBuilder

You'll want to install the WindowBuilder plugin to Eclipse if you plan to edit any of the screens. Go to `http://www.eclipse.org/windowbuilder/download.php` for instructions.

To use WindowBuilder, right click on a Panel file in the Package Explorer, then choose Open With... and select WindowBuilder Editor.

Part III
How to add a new model

3 Example of a new model: Fibonacci rabbits

We'll look at a simple model idea and the steps needed to incorporate it into a model.

3.1 Description of the model

Fibonacci once posed the following question:

> Suppose a newly-born pair of rabbits, one male, one female, are put in a field. Rabbits are able to mate at the age of one month so that at the end of its second month a female can produce another pair of rabbits. Suppose that our rabbits never die and that the female always produces one new pair (one male, one female) every month from the second month on. How many pairs will there be in one year? (from `http://fibonacci.uni-bayreuth.de/project/fibonacci-and-the-rabbits/the-story.html`)

Now, let's code!

3.2 Create package

From the Explorer window, select `File` ⇒ `New` ⇒ `Package`. Use `edu.umn.ecology.populus.mod` for the package. By convention, models are in a package/directory just under `edu.umn.ecology.populus.model`.

3.3 FRParamInfo

I think it's easier to think of what data will be taken from input screen. In this case, we just need the number of months, or generations, to run.

Right-click on the new package and select `New` ⇒ `Class`. Type in `FRInfo` for the name Add in the `Interface` `edu.umn.ecology.populus.plot.BasicPlot`. Press `Finish`.

Create a constructor that takes as input the number of generations. You should implement code here that creates a new `BasicPlotInfo` as a field.

Implement `getBasicPlotInfo()`, which will return a `BasicPlotInfo` object.

If you are creating a more-complicated model, you will want to create a `FRData` class that aggregates the data that you need to pass from the panel.

Your code should look something like this:

```
package edu.umn.ecology.populus.model.fibrabbits;

import edu.umn.ecology.populus.plot.BasicPlot;
import edu.umn.ecology.populus.plot.BasicPlotInfo;

public class FRInfo implements BasicPlot {
    private BasicPlotInfo bpi;

    public FRInfo(int maxGens) {
        bpi = new BasicPlotInfo();
        bpi.setMainCaption("Fibonacci Rabbits");
        bpi.setXCaption("Generation");
        bpi.setYCaption("Pairs of Rabbits");
        bpi.setIsDiscrete(true);

        //Generate Data
        double data[][][] = new double[1][2][maxGens+1]; //1 line with 2 variables
        double newbornPairs = 1.0;
        double maturePairs = 0.0;
        for(int gen = 0; gen <= maxGens; gen++) {
            data[0][0][gen] = (double) gen;
            data[0][1][gen] = newbornPairs + maturePairs;
            double prevNewbornPairs = newbornPairs;
```

```
            newbornPairs = maturePairs;
            maturePairs += prevNewbornPairs;
        }
        bpi.setData(data);
    }

    @Override
    public BasicPlotInfo getBasicPlotInfo() {
        return bpi;
    }
}
```

3.4 FRPanel

This is the input screen. Right-click on the new package and select `New` ⇒ `Class`. Type in `FRPanel` for the name Type in `edu.umn.ecology.populus.plot.BasicPlotInputP` for the Superclass (or use the browse button) Press `Finish`

Close the tab, then re-open it with WindowBuilder. You don't have to use WindowBuilder, but it definitely makes it easier. Click on the `Design` tab for the WYSIWYG designer of the window. We'll want to use a `PopulusParameterField` here for selecting values. If it is not yet in the WindowBuilder field, right-click on your menu of choice in the Palette and select `Add Component...`. Choose a name of your choice (I use PPField) and use `edu.umn.ecology.populus.visual.ppfield.Populu` for the Component.

Now click on the `PPField` in the `Palette` then click into the panel to insert it there.

In the properties window, set the `currentValue` and `defaultValue` to `10.0`. Set `helpText` to a long description, like `Total number of months for rabbits to grow` (this is the hover text). Set `integersOnly` to `true`, since we only want to allow an integer value (even though the underlying model uses floating point). Set `parameterName` to `months`. Set `minValue` and `maxValue` to something reasonable like `1.0` and `200.0`, respectively.

Now switch back to the Source view tab.

At the end of the constructor, add the following line so that user inpuut events will trigger plot updates:

```
this.registerChildren(this);
```

Now implement `getPlotParamInfo()`, which should return an object of type `FRInfo`.

Implement `getOutputGraphName()`, which will return a string for the main title of the output window.

Your code should like this:

```
package edu.umn.ecology.populus.model.fibrabbits;

import edu.umn.ecology.populus.plot.BasicPlot;
import edu.umn.ecology.populus.plot.BasicPlotInputPanel;
import edu.umn.ecology.populus.visual.ppfield.PopulusParameterField;

public class FRPanel extends BasicPlotInputPanel {
    private static final long serialVersionUID = -982727645471238633L;

    private PopulusParameterField maxGenerations;

    public FRPanel() {
        maxGenerations = new PopulusParameterField();
        maxGenerations.setMinValue(1.0);
        maxGenerations.setMaxValue(200.0);
        maxGenerations.setHelpText("Total number of months for rabbits to grow");
        maxGenerations.setParameterName("months");
        maxGenerations.setIntegersOnly(true);
        maxGenerations.setDefaultValue(10.0);
        maxGenerations.setCurrentValue(10.0);
        add(maxGenerations);
        this.registerChildren(this);
    }

    @Override
    public BasicPlot getPlotParamInfo() {
        return new FRInfo(maxGenerations.getInt());
    }

    @Override
    public String getOutputGraphName() {
        return "Fibonacci Rabbits";
    }
}
```

3.5 FRModel

Now create `FRModel`. Its Superclass is `edu.umn.ecology.populus.plot.BasicPlotModel`.

Implement `FRModel()` to set the modelInput to a new `FRPanel`.

Implement `getModelName()` to return the model name.

Don't worry about implementing `getModelHelpText()` and `getHelpId()` at this stage. These functions are so that users looking for help will go to the context-specific section of the help pdf.

Your code should look like this:

```
package edu.umn.ecology.populus.model.fibrabbits;
import edu.umn.ecology.populus.plot.BasicPlotModel;

public class FRModel extends BasicPlotModel {
    public FRModel() {
        this.setModelInput(new FRPanel());
    }

    public static String getModelName() {
        return ("Fibonacci Rabbits");
    }
}
```

3.6 Res

You may want to create a Res file that should be used for storing all of the String resources. See how other models use it.

3.7 Add model to the menu

If you can shoehorn this into an existing menu group, it's quite easy. Just go to `PopPreferences::initializeMenuPackets()` and add a single line with the new `ModelPacket` accordingly. If you wanted this near in the single-species dynamics menu, add this in the initialization list of `singleModels`:

```
new ModelPacket( edu.umn.ecology.populus.model.fibrabbits.FRModel.class ),
```

If you have to create a new `ModelPacket` array, you'll need to also add code to `DesktopWindow`.

Part IV
Java Source Code

4 Models

4.1 Files

By convention, each end model (not meant to be inherited from) should be in the package `edu.umn.ecology.populus.model.`*`ModelName`*.

4.1.1 Model

A `Model` holds together the basic parts of a model.

4.1.2 ModelPacket

A `ModelPacket` is a simple wrapper for a model so we can refer to one class at a time, and used in making the menus. The menus are created in `initializeMenuPackets()`, and this is manually updated to add or remove models.

4.1.3 ModelPanel

The `ModelPanel` (input window) base files are in `edu.umn.ecology.populus.edwin` (short for editor window, from the Pascal DOS program's naming conventions).

`registerChildren()` looks at all of the components, and sets event listeners where appropriate. Read Events for more information.

4.1.4 ModelOutputPanel

The `OutputPanel` (output window) base files are in `edu.umn.ecology.populus.resultwindow`

4.2 Events

When changes in the input panel occur, events - or messages - are sent to the output. The `ModelPanel` will call `fireModelPanelEvent()` whenever a change occurs, with a constant such as `CHANGE_PLOT`. If this warrants a new

output, ModelPanel will be queried for, in the case of Basic Plot, new plot info.

Do not assume that getPlotInfo() will be called whenever you call fireModelPanelEvent. For example, if changing the value of a radio button should disable another parameter, that should be done separately from getPlotInfo(). See the method modelPanelChanged() to see which events are ignored and which events create a new plot.

Inherited models should not have to worry about when to show the output screen. registerChildren() is called after the initialize of the front panel, and this routine looks at all of the components and adds listeners to the ones that should through events. There is a setting in the Preferences so that users can change when to automatically update the output and making decisions on a model-by-model basis will not work with this. See the Preferences note for this.

4.2.1 Events: An Example

Let's imagine a case where a user loads a simple model, namely Density Independent Growth, and then changes the r parameter.

When Populus starts, a DesktopWindow::MenuAction menu item, which has a reference to the class DIGModel, is loaded in DesktopWindow::loadMenu().

When the user selects Density Independent Growth in the menu, we call DesktopWindow::loadModel(), where it creates a new instance of a DIGModel. (We know that it is a DIGModel from the MenuAction object.)

In the DIGModel constructor, it creates a DIGPanel object, and calls registerChildren() to set up events from the input panel. Then it calls setModelInput(), which creates the model input frame and starts listening to ModelPanelEvents with modelPanelChanged().

In ModelPanel::registerChildren(), we add listeners to all of the UI components. These will call Model::modelPanelChanged(ModelPanelEvent e) when they receive events.

Now the user changes the r value. The UI component fires an event to ModelPanel, which calls Model::modelPanelChanged(). In modelPanelChanged, it sees that this is worthy of an update, and so calls BasicPlotModel::simpleUpdateOutput() via Model::updateOutput().

In BasicPlotModel::simpleUpdateOutput(), it takes a DIGData object from the input using DIGModel::getPlotParamInfo(), and passes it to the output. Since we haven't already, we will also create a ModelOutputFrame,

which contains a `BasicPlotOutputPanel`, which contains a `BasicPlotCanvas`. Most of the real code for output is in `BasicPlotCanvas`, although that code is really the same for all 2D graphs.

On subsequent changes to `r`, the code path is similar, except we don't need to create some of the output objects again. Instead, we just call `BasicPlotCanvas::setBPI()`.

4.3 Adding a Model to the Menus

To add a model to the menu, add a ModelPacket in PopPreferences.

I dreamed of one day being able to dynamically modify these models. Maybe we could load a file `Model` class on the fly and it would be included in the top-level menu for that session. Or even store it in the preferences. But we haven't had much of a need, and Don would've preferred the simpler one-size-fits-all approach.

4.4 Basic Model

Most models will derive from `BasicPlotModel`, in the plot directory.

4.5 Common Variants

Most models extend from `edu.umn.ecology.populus.plot.BasicPlotModel`, which does basic graphing. But you don't have to do this. See `Woozleology` for an example of one that does not extend from this.

5 Main

`main` is found in `edu.umn.ecology.populus.core.PopRun`. This is where the application starts. The `DesktopWindow` is the primary GUI background to the application.

6 Help

6.1 How Help Events are Triggered

When we click the Help button on a model or the main DesktopWindow, we call `HelpUtilities::displayHelp()`.

When we click on the Help button within a model, it's very similar, but we use the `getHelpId()` from the model to get a Named Destination into the PDF file. Named Destinations are like HTML anchors, except that a lot of PDF viewers ignore them.

TODO - I suspect that getModelHelpText() doesn't really do anything these days. Maybe we can gut that.

6.2 Displaying help

The help system was changed dramatically in 5.5. There are several options for how we will invoke a PDF reader, either internal APIs (via JNLP) or by using the system's "open" command, or a custom call that users can set. Look at `displayHelp` for the list of ways.

In addition, languages by modifying the local help file to use the language specified by the user's configuration. It's hard to tell the external PDF viewer to look at a file in an archive, so we extract the help file PDF into the user's home directory before invoking the PDF viewer.

Look at the README file for more information.

7 Logging

Logging normally goes both to stderr and a file in the user's home directory.

Logging should be handled by `Logging.java`. On the TODO list are changing references to `System.out.println` and `System.err.println` to use `Logging` instead; add a preferences option to change the logging level for stderr and the output file; and use Java's standard logging system.

8 Preferences

8.1 PreferencesFile

The file for keeping state is stored as `userpref.po` in the user's home directory (as of Populus 5.4). It is loaded during initialization. By default, it is in the user's home directory – not in Populus's – because we aren't guaranteed write permission for all systems. This can be overrided by the startup command - see `README.config`.

Almost all of the code is in `PopPreferences.java`.

8.2 Preferences GUI

The GUI behind it is in `PreferencesDialog.java`.

A lot is also explained in the README section following here.

9 README

There is a file in the doc directory, `README.config.txt` that is distributed with Populus. The target audience is for people in charge of installing or administering Populus.

10 Installer

`Installer.java` was meant to be used with JNLP install, but never made it to full production.

11 Custom GUI Widgets

Custom widgets are in the `edu.umn.ecology.populus.visual` package (or underneath it).

11.1 ParameterField

The ParameterField was originally concocted as a spinner. But then we added the variable name, and variable information to the parameter. I like

to use this with WindowBuilder (more details about this in the section on creating a new model).

11.2 HTMLLabel

This widget was created in the day when Java didn't have good options for HTML-like tags, especially superscript and subscript. It's basically a JLabel that knows about some HTML, and can also rotate text. Nowadays we don't use too much of it, but there is a bean so you can use it in WindowBuilder.

See Library dependencies for third-party dependencies.

12 Javadoc

I wish the code were better documented. But you can still use `javadoc` to generate documentation for the files.

13 Library dependencies

Populus relies on several libraries (JAR files). These are packaged into `PopulusAll.jar` in the build.

Modifications to the dependencies will require changes to `fullbuild.xml` (and anything else to get it to run) as well as `AboutPopulusDialog.java`.

13.1 Jama

Jama is an free-to-distribute library for Java math. We use it in the structured population growth models for eigenvalue decomposition. See http://math.nist.gov/javanumer
for details.

13.2 JClass

JClass includes the chart software for Java that we use. The Manifest file in the JAR file they included has some bogus `dependson` lines that give warnings when you try to run. I manually deleted these, and just keep this new version around.

JClass keeps switching companies. We have an old version of their product, and I don't have any reason for upgrading. We may update to something more common today, like JFreeChart.

13.3 acrobat

The acrobat JAR was added as a PDF viewer that's built into Populus. It didn't work as well as hoped, and Adobe doesn't seem to support it anyway, so it was dropped. If the need to use it again arises, look at `OpenPDFWithAdobeBean.java`.

Part V
Installer

Mostly, we just distribute the `PopulusAll.jar`. (The "All" part of the name means that it includes its dependent libraries with it.) The Ant task to run this is `create_run_jar`.

14 Populus Splash Screen

We have a file called Populus*.*.psd which is a photoshop file describing the title screen. For a new release, we probably want a change in version number, so make a new .psd file with the new version, and then export it to gif format (calling it `PopulusSplashScreen.gif`) and replace the one in edu/.../core/ with the new gif.

15 README

See the previous section of the README. This is currently separate from the install, but put on the install page.

16 OS X Build

For OS X, please run the `bundle_populus` option. This will create an OS X package in the `out` directory. Manually run iDMG to turn it into a `.dmg` file for easy transportation. (Using iDMG is simple: Just drag the application file into it, and it will create the .dmg file in the same directory.)

17 JNLP (Obsolete?)

For a while, we considered using JNLP (aka Java Web Start) so that we could install via the Web. Most of the files still exist as a proof of concept, but you may need to do some polishing to get it to work.

There are a few references to using JNLP in the code, namely in the Help. This does not imply that Populus is run via JNLP; It just uses the JNLP library.

Part VI
Web Page

This should all be handled by the UMN Web team these days. They now use Drupal (a content management system). For 5.5, I just gave them a new JAR file.

Part VII
Test and Verification

18 Release Checklist

Test that the output for each model agrees with what is in the PDF. Math is cross-platform, so I don't see much of a problem testing this on all of the platforms.

19 Platform

It's a good idea to test on different platforms. They will look different from platform to platform, and that's fine.

It is necessary to test that you test the following on each platform:

1. Populus installs and launches.
2. You can open a basic model.
3. When you click Help for a model, the PDF should open, hopefully on the correct location for the model. Opening to the correct Named Destination is a fickle beast (see Help section), so might not be a new bug by itself.
4. Preferences are stored correctly between successive launches of Populus. Make a change in Preferences, save it, then exit and restart Populus to see if the change was saved correctly.
5. Save and load models, especially between a restart. It would be nice to save a model on one platform and have it open correctly on another platform, but we aren't there yet.

19.1 Linux

LiveCD SLAX can boot up Linux on an otherwise Windows computer. There are other options now too. I personally use Ubuntu.

19.2 Mac OS X

You really just need a Mac for this. The UofM computer team have testers to help with this.

I don't really know of anyone that uses Linux, but I like to make sure this works just on principle.

We've had a lot of sizing bugs on Mac. It's best to open several models (mainly the input windows) and make sure the screen's layout looks okay.

Ideally, we'd also like to test that it will install a proper version of Java if it is not installed already. This is not yet implemented. See the note in `fullbuild.txt`

Permissions are a bit trickier now, with the addition of Gatekeeper. We still haven't figured out the best way to do this, but make sure that we can still run it as a

Part VIII
Conclusion

If anything is confusing in this document, please update or expand it as needed.