Contents

Creating your jME interface using GBUI 3

1. Getting started with GBUI in Eclipse 3

1.1 Caveat Emptor 3

1.2 Tools 3

1.3 Getting the GBUI source 3

1.4 Installing the Groovy plug-in 5

1.5 Compiling the project 6

1.5.1 Troubleshooting 9

1.6 Running the tests/demos 11

1.7 Building a jar file 12

2. “GBUI 101” 14

2.1 Orientation 14

2.2 Component Hierarchy (or, "Why Understanding Swing Will Be Helpful") 15

2.3 Style Hierarchy (or, "Why Understanding CSS Will Be Helpful") 18

3. Creating your own “look-and-feel” 30

4. Layout Managers 31

5. Putting it all together: “A Technique” 32

5.1 Required Reading 32

5.1.1 Model-View-Controller/Model-View-Presenter/State-View-Controller 32

5.1.2 Presenter-First 32

5.1.3 Inversion-of-control/Direct-Injection 32

5.1.4 Game States 32

5.1.5 StandardGame 32

5.2 “A Technique” 33

5.3 What we’re going to create 33

5.4 How we’re going to do it 34

5.4.1 Main 35

5.4.2 ClassFactory 36

5.4.3 GameManager 38

5.4.4 Translator 39

5.4.5 Main Menu 40

5.4.6 Select Campaign 43

5.4.7 Start Server 45

5.4.8 InGame 46

5.4.9 InGamePause 47

5.4.10 Edit Settings 48

5.5 What’s Next? 50

Appendix A: References 51

Creating your jME interface using GBUI

1. Getting started with GBUI in Eclipse

1.1 Caveat Emptor

In the words of "standtrooper": "When BUI was written BSS was, as best I can tell a "streamlined" or thinner version of CSS. It never occurred to me, a web developer, that there would be Swing developers who don't know CSS (just as there are Swing developers who don't think that there are web developers that don't think like them  ."

I don't consider myself to be either a web developer, or a Swing developer. I've used CSS and am generally familiar with the concept, if not the implementation. I've also used Swing, but not much in the last 9 years, so my recent work with JMEDesktop was a crash-course getting reacquainted with that. So if you don't know CSS and you don't know Swing, don't worry: that knowledge will probably help, but I think I've got it figured out so it can't be that hard. 

1.2 Tools

As a starting point, I'm going to assume the following:

- you already have Eclipse installed and running (you can create, compile, and execute Java projects)
- you have a project named "jME" in your Eclipse workspace. This project is a compiled and running version of the jME 2.0 source code. If you have any problems getting to this point, reference some of the other available "Setting up Eclipse to build jME 2" tutorials.

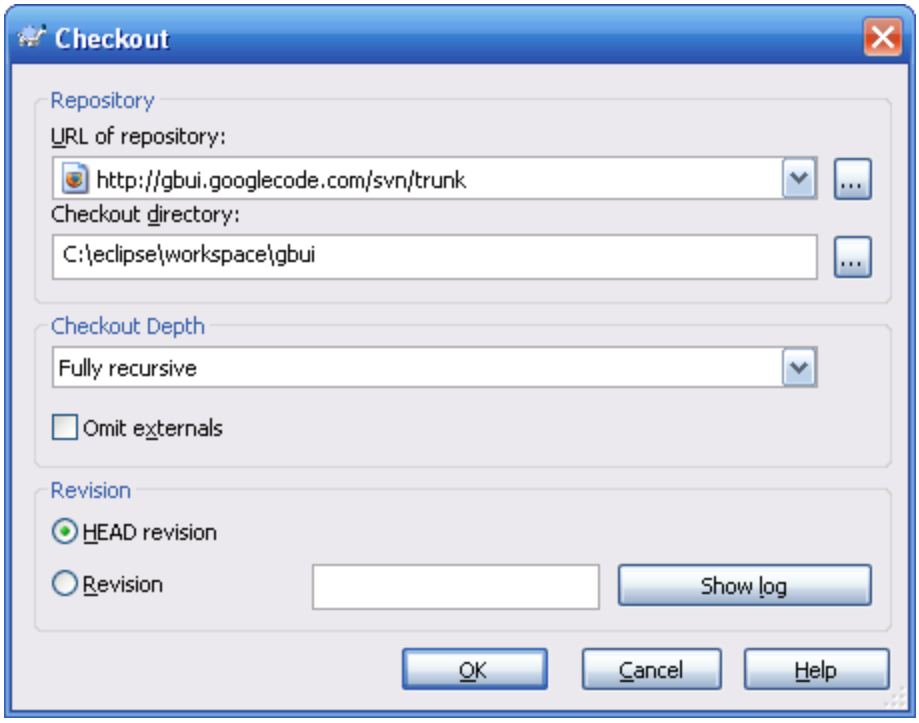
1.3 Getting the GBUI source

The Download tab of the GBUI site offers some pre-compiled jars you can download for GBUI. However, I want to work with the source code because:

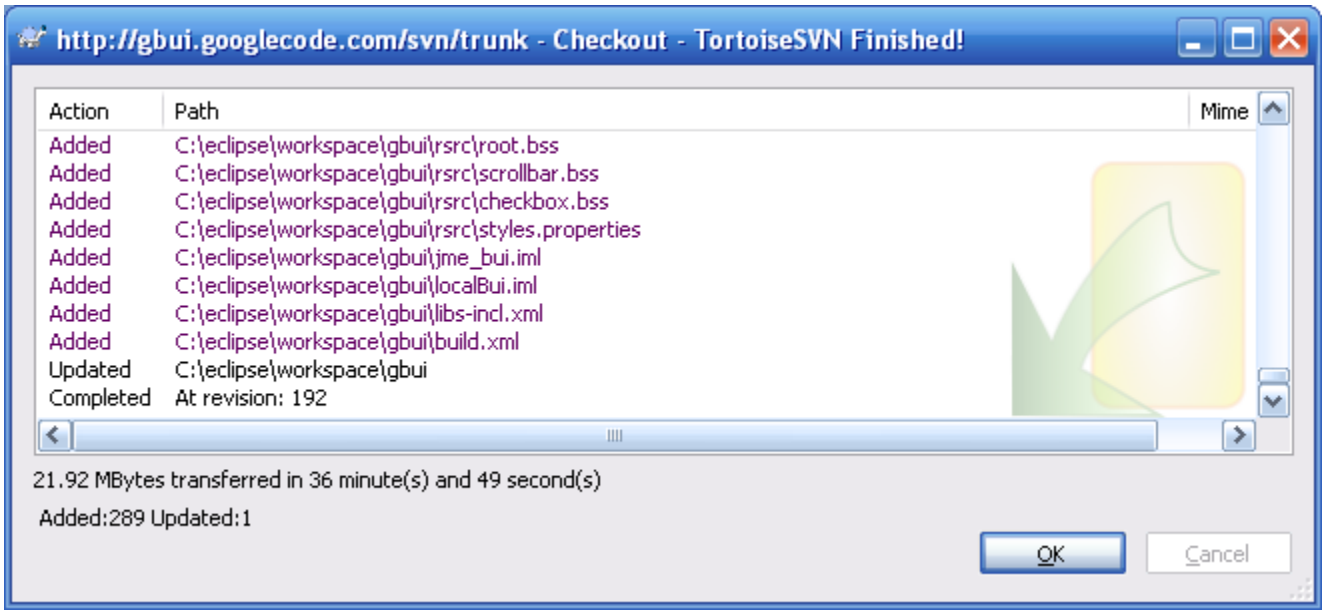
1. it's the most up-to-date
2. it contains the source code for the examples
3. when debugging, I can step into the source to help determine the root cause of any errors I encounter

When working with SVN repositories, I prefer to use TortoiseSVN. I know I've heard people rave about Subclipse, but I never had much luck with it, and I prefer the control that TortoiseSVN gives me. So if you haven't already done so, download and install TortoiseSVN. (Go ahead; I'll wait :)

After TortoiseSVN has been installed, use Windows Explorer to view your Eclipse workspace directory. Mine is C:\eclipse\workspace. Create a new folder there called "gbui". Right-click on that newly-created directory and select "SVN Checkout" (notice how TortoiseSVN nicely integrates itself?). In the "URL of repository" field of the Checkout dialog, enter "http://gbui.googlecode.com/svn/trunk" (without the quotes). The "Checkout Directory" should already be the directory you just created ("C:\eclipse\workspace\gbui" in my case), but correct it if it's not.



Click the “OK” button, then take a break while it downloads the source. The checkout status window will update as each file is downloaded so you can verify that something is happening. Wait until the last message is “Completed” and the “OK” button is enabled.

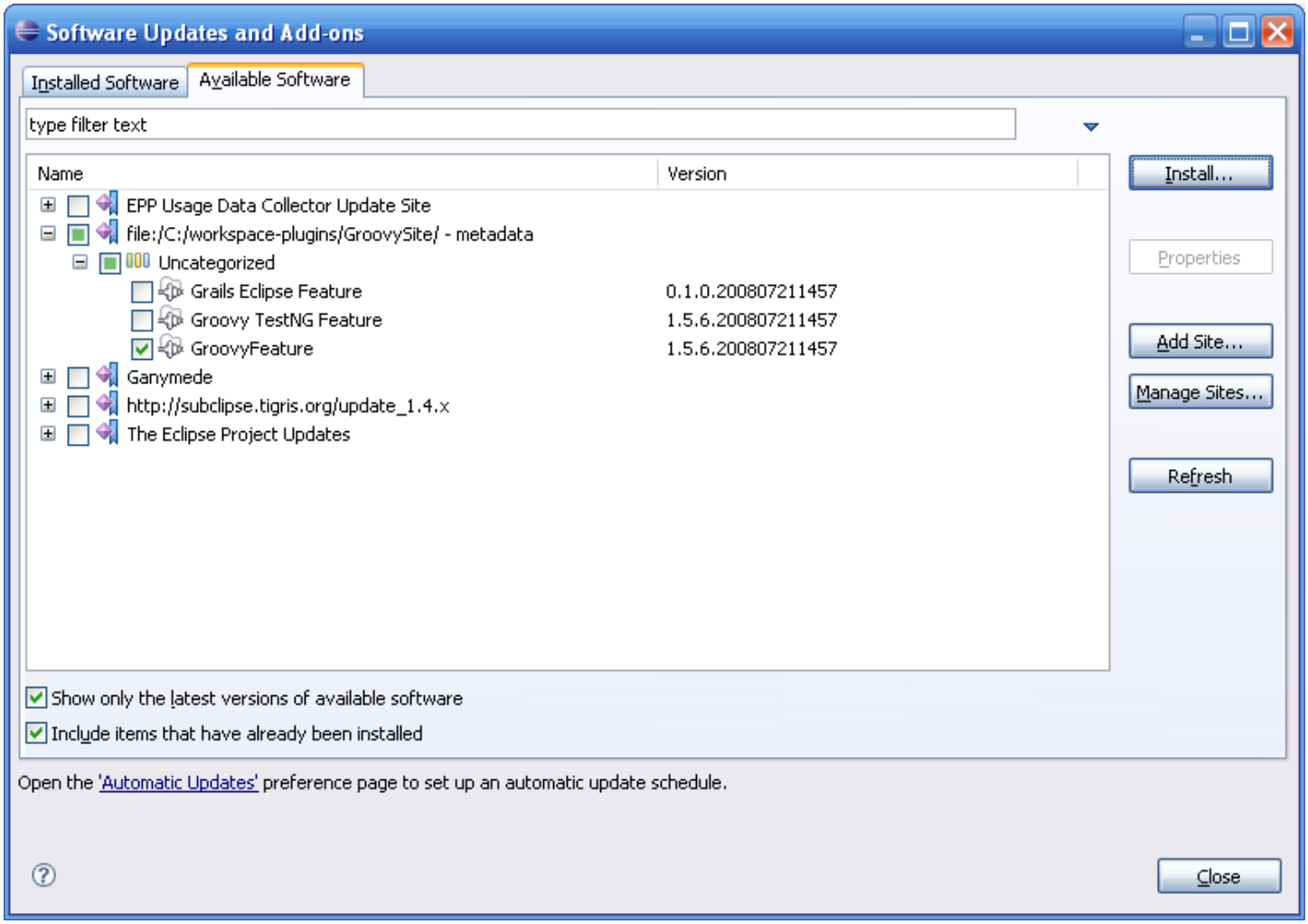


Click “OK” to close the dialog.

1.4 Installing the Groovy plug-in

Note: to save yourself some time, do this step while you're waiting for the gbui source to download.

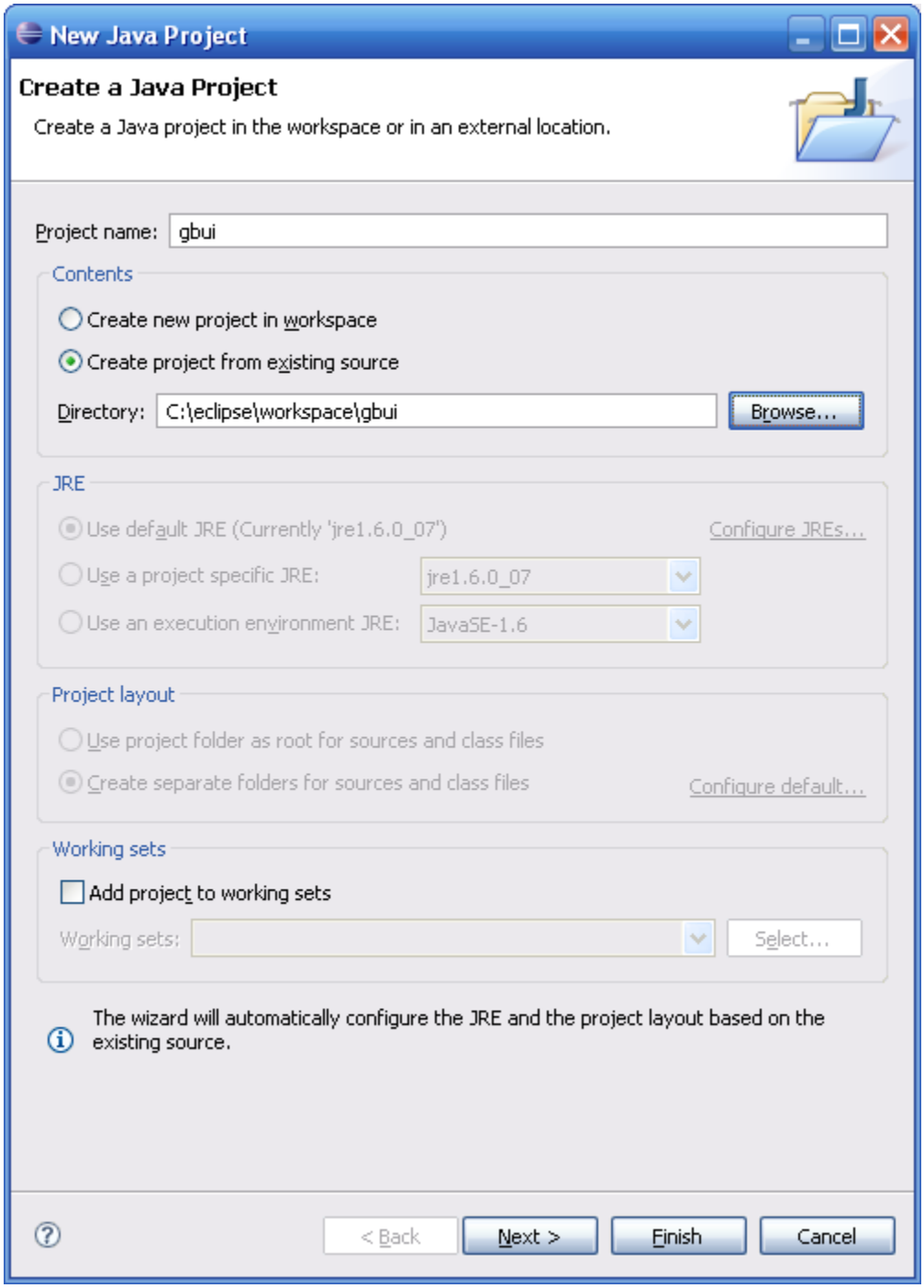
In Eclipse, navigate to Help | Software Updates. Select the "Available Software" tab at the top, then the "Add Site" button. Enter the following URL: http://dist.codehaus.org/groovy/distributions/update/ and the press "OK". Once the site has loaded, click on the plus sign to the left of the GroovySite to expand it, then expand the "Uncategorized" item. Check the box to the left of "GroovyFeature".



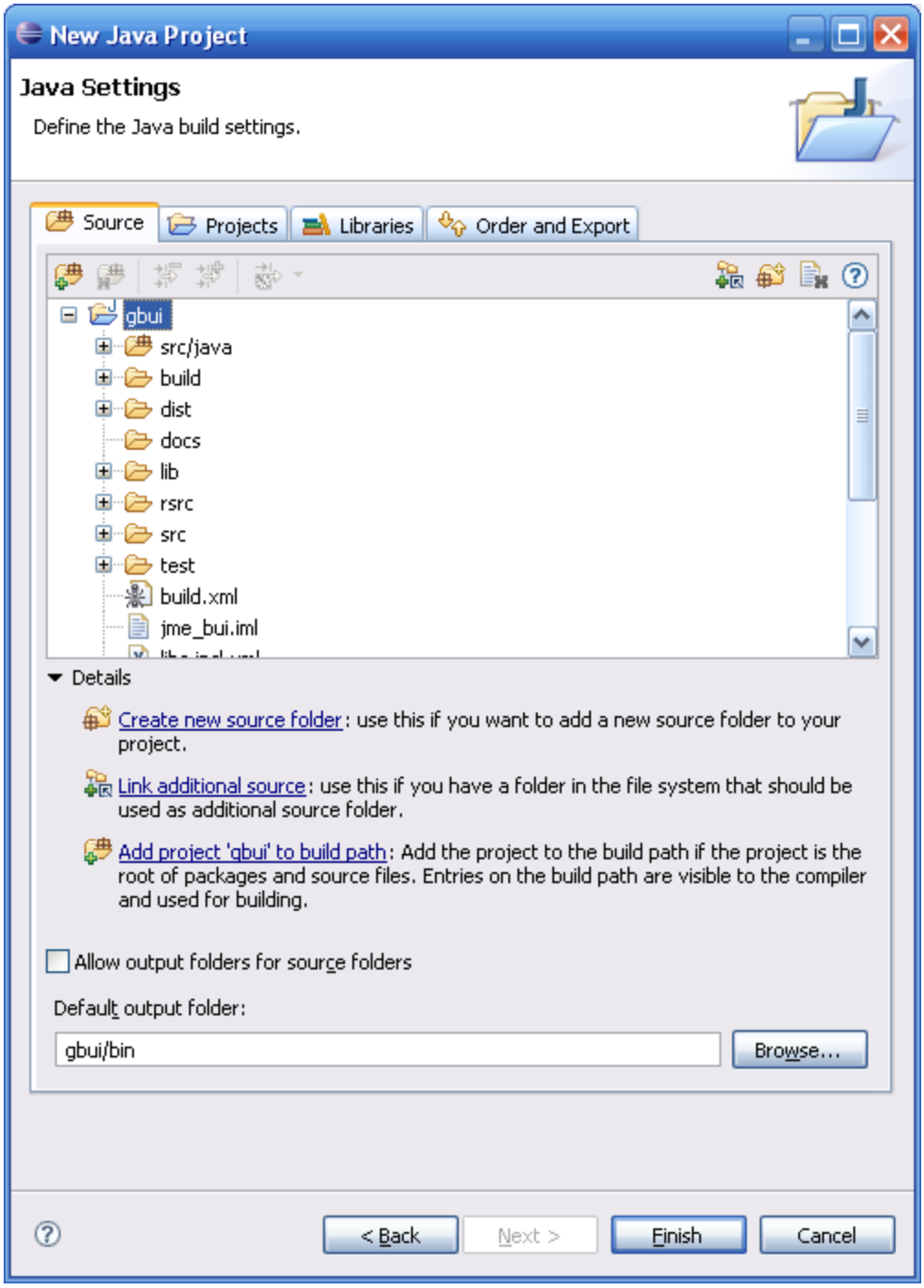
The "Install…" button in the upper-right of the window should become enabled. Press it and follow the instructions. This may require rebooting your computer, or at least Eclipse, after the installation.

1.5 Compiling the project

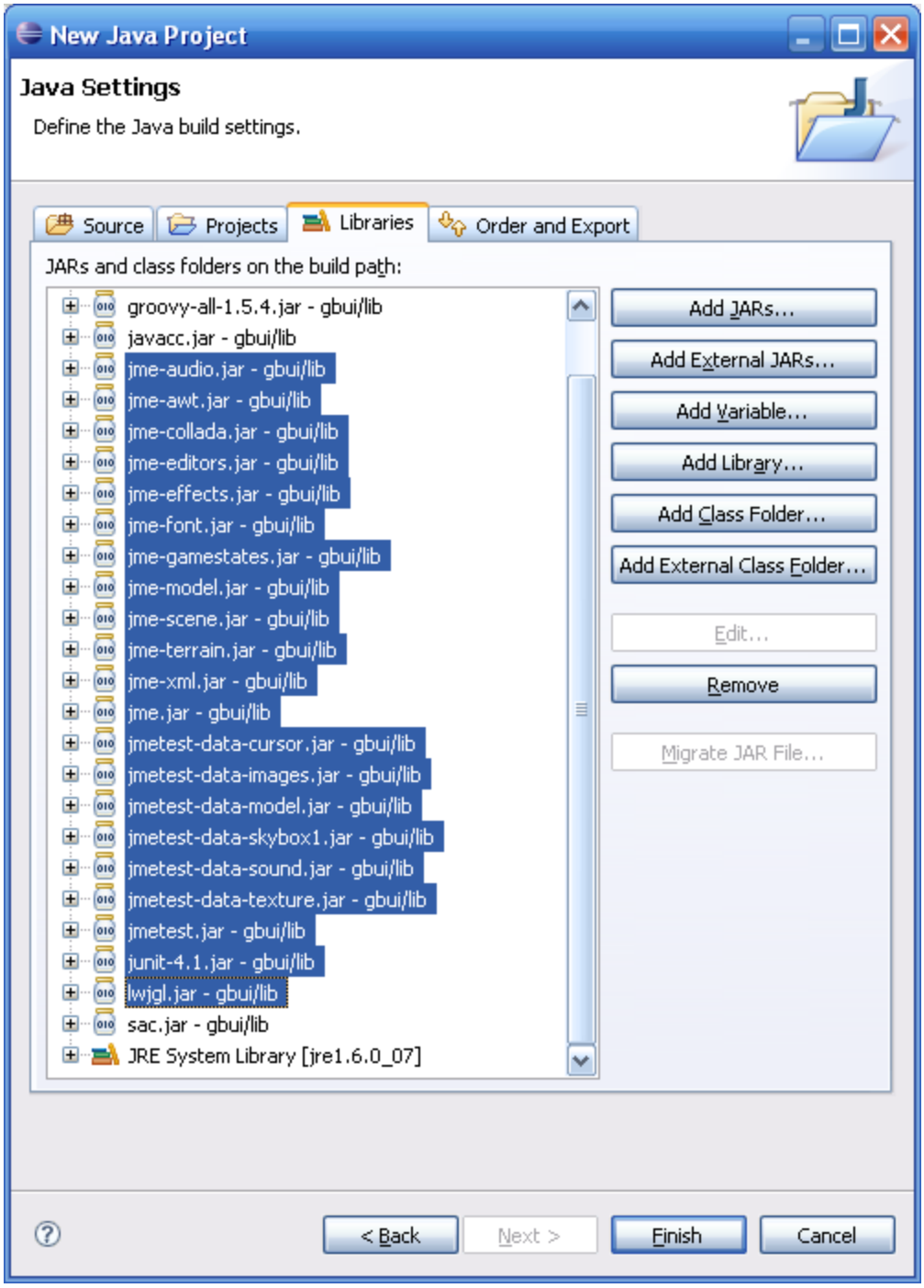
If you haven't already done it, open Eclipse. Create a new Java project. The project name must be the same as the name of the folder that you just downloaded the source code to. Case matters! Click the “Create project from existing source” option, and press the “Browse” button to navigate to the directory you created. Don't go any lower in the hierarchy (to “bin”, “lib”, “src”, etc). Press the “OK” button.



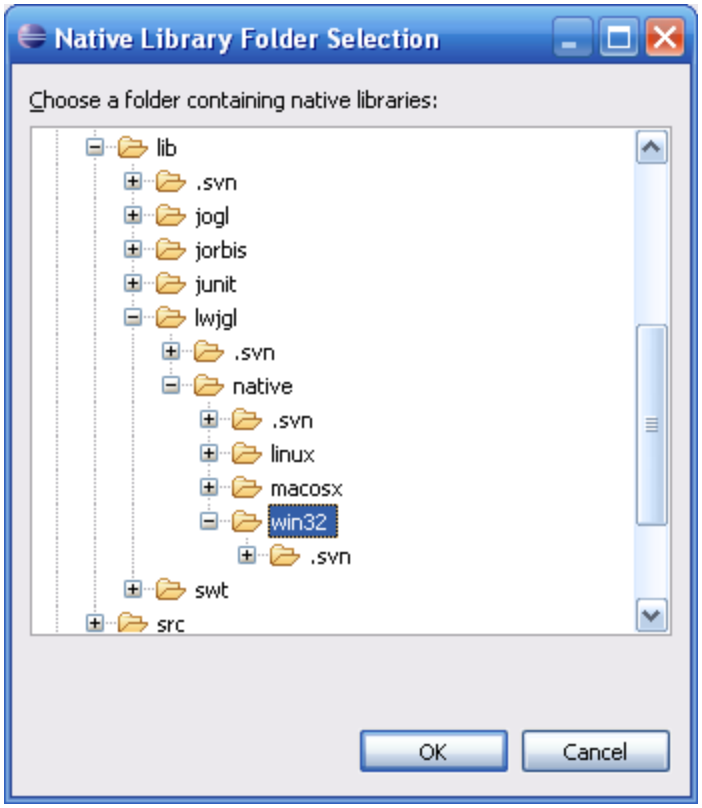
Press “Next”. Eclipse should then display all the sub-directories and files that are in the root directory. If you only see a single “src” folder, the name of the project you entered probably didn't match the directory the files are stored in, and Eclipse won't create the project properly.



Once everything looks right, select the “Projects” tab. Add the “jME” project that you already have in your workspace (remember the assumptions up above in the “Tools” section?) Select the “Libraries” tab and remove the reference to all the “jme*” files and the “lwgjl.jar”. This is so we can keep this project in sync with the same files as our jME project.



Press the “Add JARs” button and navigate to “jME/lib/lwjgl”. Select “lwjgl.jar” and press “OK”. Select the plus-sign to the left of the “lwjgl.jar”, then select “Native Library Location”. Select the “Edit” button, then the “Workspace” button. Select “jME/lib/lwjgl/native/win32”.

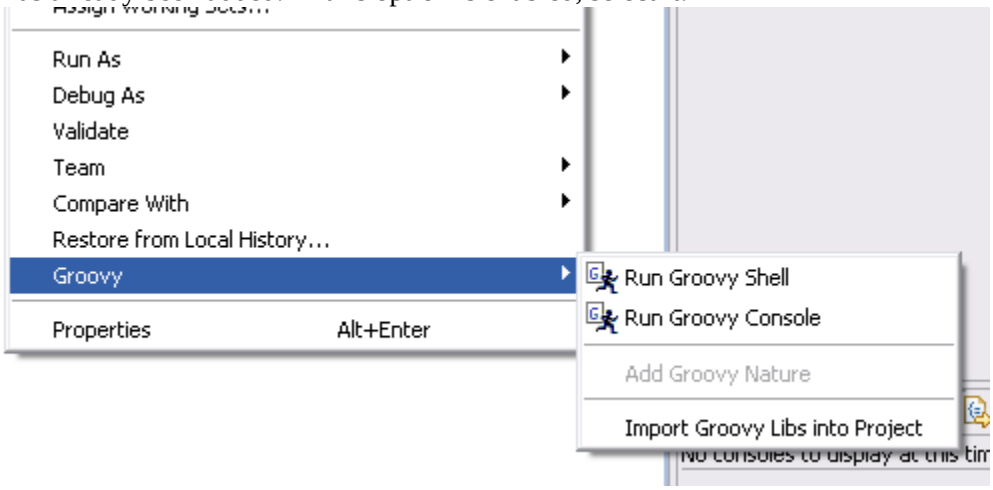


Press “OK” twice. Finally, press the “Finish” button and give Eclipse some time to compile the project (assuming you have auto-compile enabled). Check the “Problems” tab for any errors or warnings. Ignore any warnings, but there shouldn't be any errors.

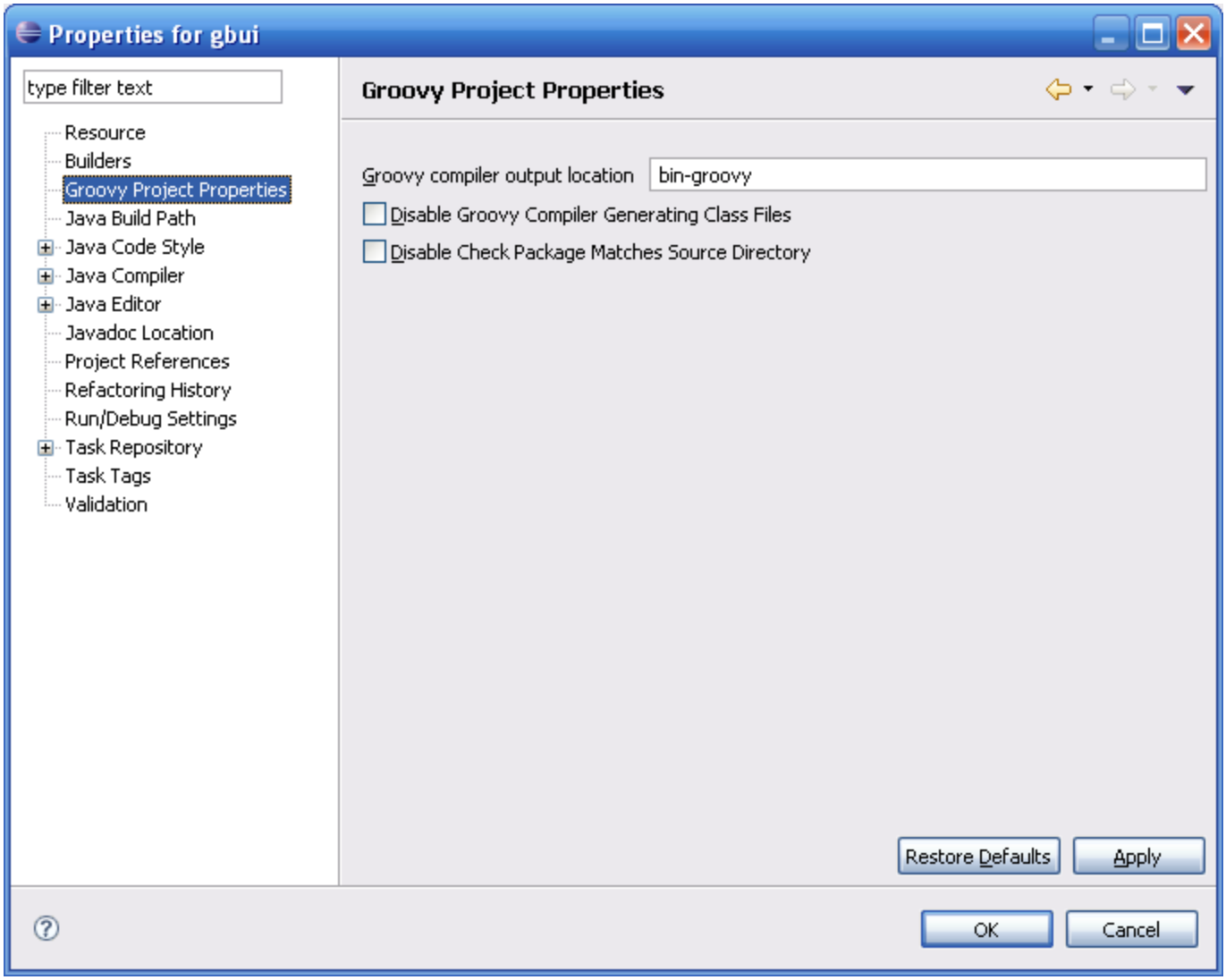
1.5.1 Troubleshooting

My experience was that Eclipse usually does a pretty good job of automatically detecting that the project uses Groovy, and automatically enables what is needed for a successful compile. However, I did run into two issues that may help. When the compile failed, it was because the BStyleSheet.groovy file was not being properly compiled, and all the classes that referenced BStyleSheet couldn’t find the definition. If that appears to be the error you’re seeing, try the following:

1) Right-click on the gbui project. You should see an option for “Groovy” right above “Properties” at the bottom. Select “Groovy”. You should see an option for “Add Groovy Nature”. This option should be disabled, indicating it has already been added. If this option is enabled, select it.



2) Right-click on the gbui project and select “Properties”. Select the “Groovy Project Options” line. Ensure the build path is set to “bin-groovy”.


Properties for gbui
type filter text
Resource
Builders
Groovy Project Properties
Java Build Path
Java Code Style
Java Compiler
Java Editor
Javadoc Location
Project References
Refactoring History
Run/Debug Settings
Task Repository
Task Tags
Validation
Groovy Project Properties
Groovy compiler output location
bin-groovy
Disable Groovy Compiler Generating Class Files
Disable Check Package Matches Source Directory
Restore Defaults
Apply
OK
Cancel

1.6 Running the tests/demos

I'm not a fan of full-screen applications (especially when trying to debug), so I wanted a chance to modify the default properties. In your “gbui” project, open the “src/java”, then the “com.jmex.bui.tests” package. Double-click on “AllDialogTests.java” to open it. Modify it's main method to the following:

```
public static void main(String[] args) {
    Logger.getLogger("com.jmex.bui").setLevel(Level.WARNING);

    AllDialogsTest test = new AllDialogsTest();
    test.setConfigShowMode(ConfigShowMode.AlwaysShow);
    test.start();
}
```

Save the changes.

This next step seems very kludgy, and I hope there's a more elegant solution that someone else can point me to, but for now I'm taking the path of pragmatism: it works for me, and I don't know a better way to do it.

1. right-click on the “rsrc” folder and “Copy”
2. right-click on the “src/java” folder and “Paste”
3. right-click on the “rsrc” folder and “Delete”. Yes, you're sure.

This is the only way I've found that will let you modify the GBUI styles inside Eclipse, and still use those changes files when running the tests.

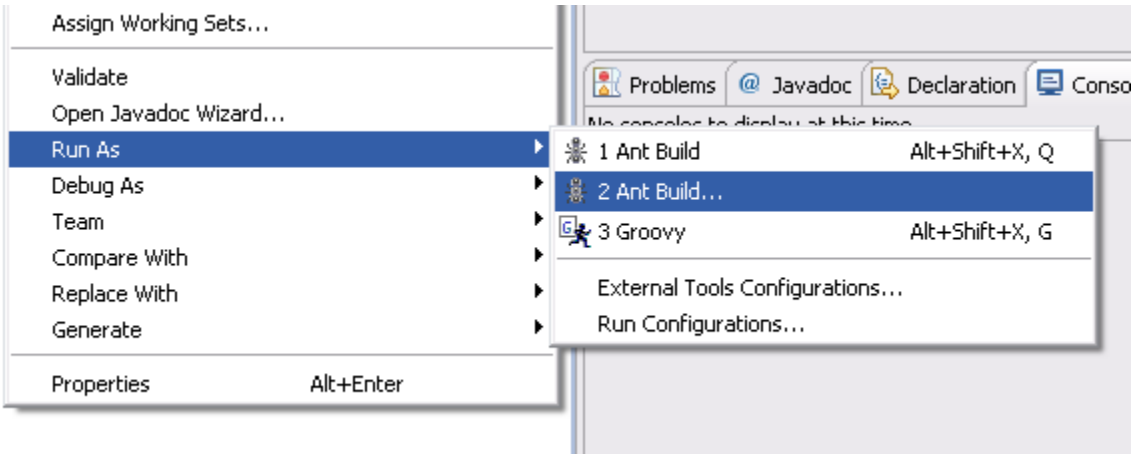
Finally, right-click on the “AllDialogsTest.java” and run as a Java application. You'll be presented with the default jME properties configuration. Un-select the “Fullscreen?” checkbox and press “OK”. You should be presented with a dialog in the middle of the window. Move the dialog to see more underneath it.

Congratulations! You've just gotten GBUI to work, and can now run all the tests to see different examples of what it can do.

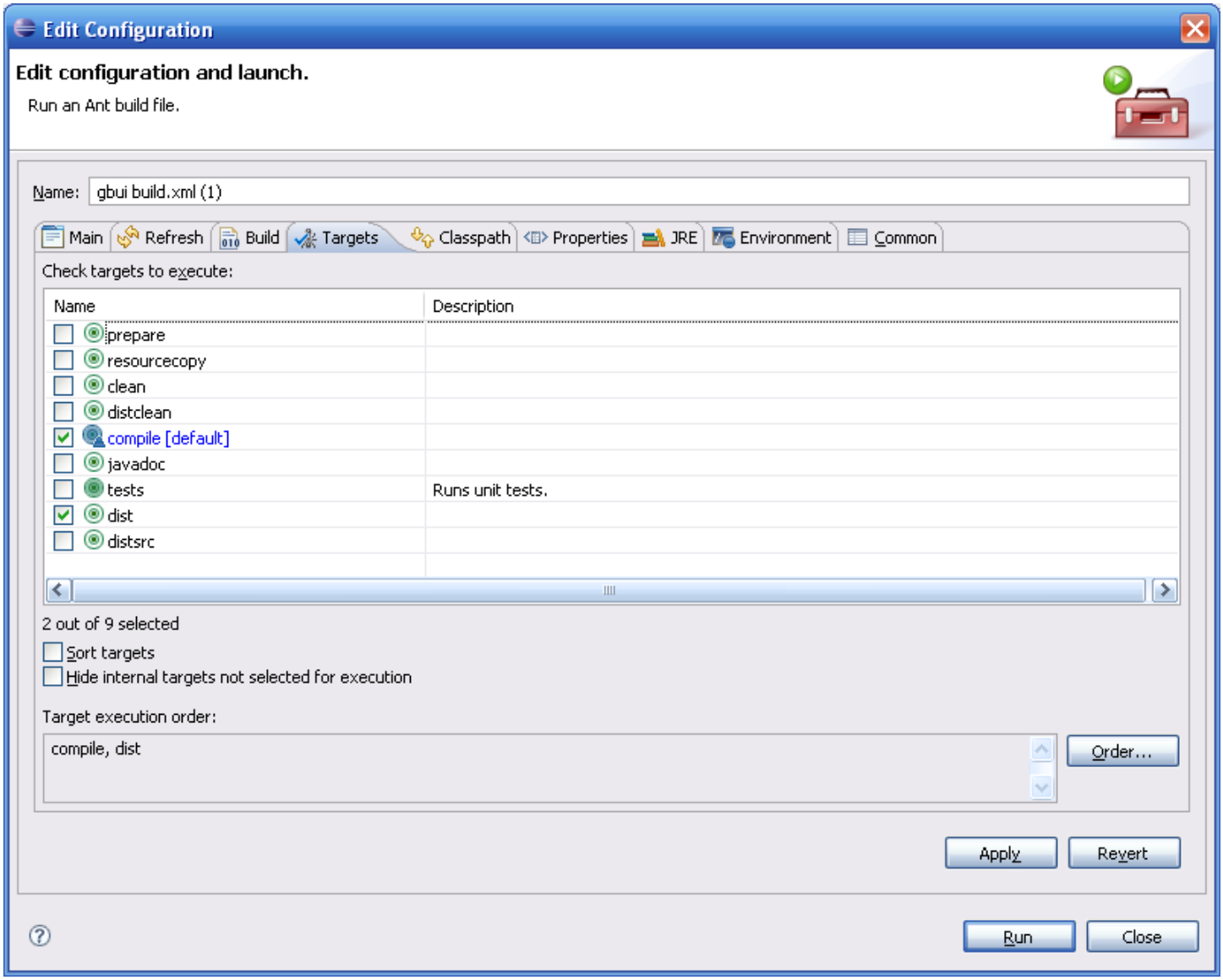
1.7 Building a jar file

As I mentioned at the beginning, I prefer to work with the latest version of the source, rather than a pre-built distribution. However, the reality is that once I have the source downloaded and working, I usually ignore that code while I work on whatever application I'm writing that uses the library. Everything works well until I run into a bug that's fixed in a later version than I have, or a feature is added, or whatever. So I update the source of the library, and then the compile fails. Now my "real" application is dead in the water. My solution to this is to download the source of the library, and if it compiles, build a jar that I use for developing against. Then, when I update the source of the library, I can continue developing against the jar if the latest source doesn't compile for some reason. If the updated code does compile, then I change the reference in my project from the jar of the library to the actual project containing the source of the library. If my "real" application still runs and behaves as expected, I build a new jar and develop against that. If any type of error or unexpected behavior happens, I can either dig into the library to see what changed, or I can revert to the jar.

To create a jar of the GBUI library, just right-click on the build.xml file in the root directory of the source code. "Run As…" "2 Ant Build…".



Select the "dist" option.



Press the “Run” button and watch the console. If the build is successful (which it should be), you will now have a jme-bui.jar file in your gbui\dist directory that you can reference from any other application without worrying that changes to the gbui source will break anything before you’ve had a chance to test it.

2. “GBUI 101”

2.1 Orientation

In the words of “standtrooper”: “When BUI was written BSS was, as best I can tell a “streamlined” or thinner version of CSS. It never occurred to me, a web developer, that there would be Swing developers who don't know CSS (just as there are Swing developers who don't think that there are web developers that don't think like them 😉.”

I don't consider myself to be either a web developer, or a Swing developer. I've used CSS and am generally familiar with the concept, if not the implementation. I've also used Swing, but not much in the last 9 years, so my recent work with JMEDesktop was a crash-course getting reacquainted with that. So if you don't know CSS and you don't know Swing, don't worry: that knowledge will probably help, but I think I've got it figured out so it can't be that hard.😀

2.2 Component Hierarchy (or, "Why Understanding Swing Will Be Helpful")

As a developer, when I started working with GBUI, my primary interest was in what the library can offer me; what widgets are available? What events can they respond to? How easy is it to use? From that perspective, I can say that the library offers an excellent selection of widgets, and the implementation reminds me of Swing. At the top is BComponent (think "JComponent"), from which everything derives.

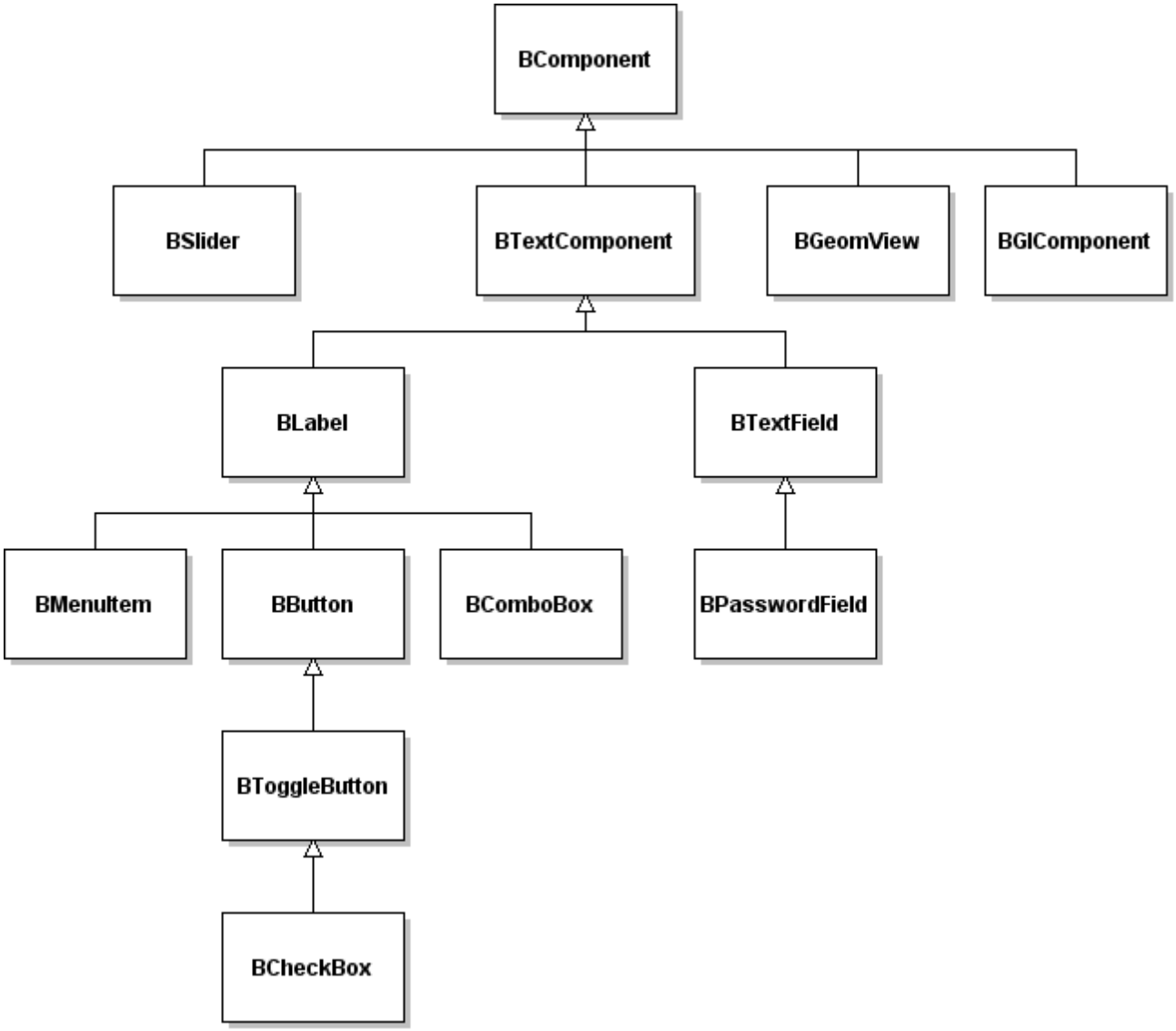


Figure 2.2.1: This shows BComponent down to some of the more common widgets such as text components, labels, buttons, menu items, combo box, etc.

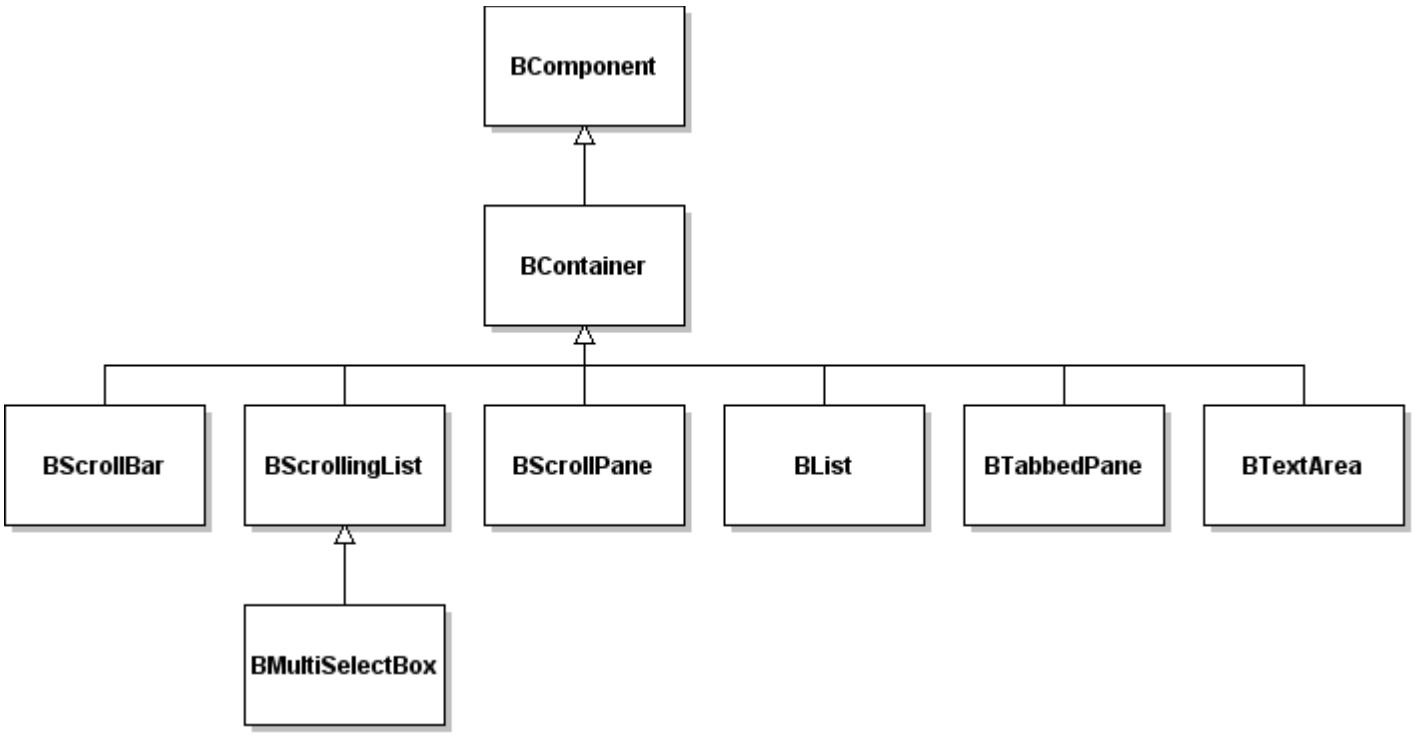


Figure 2.2.2: This shows BContainer deriving from BComponent, and then some of the more common container-type widgets. Again, think of parallels to Swing components with similar names.

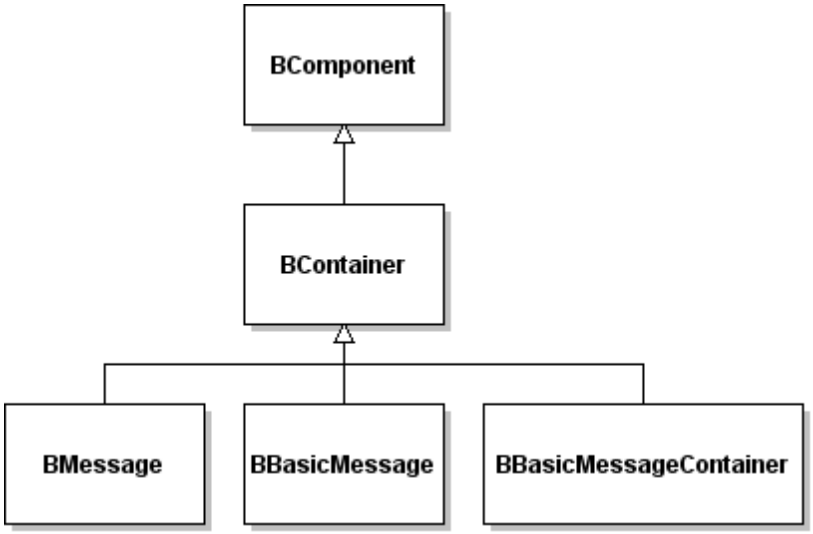


Figure 2.2.3:

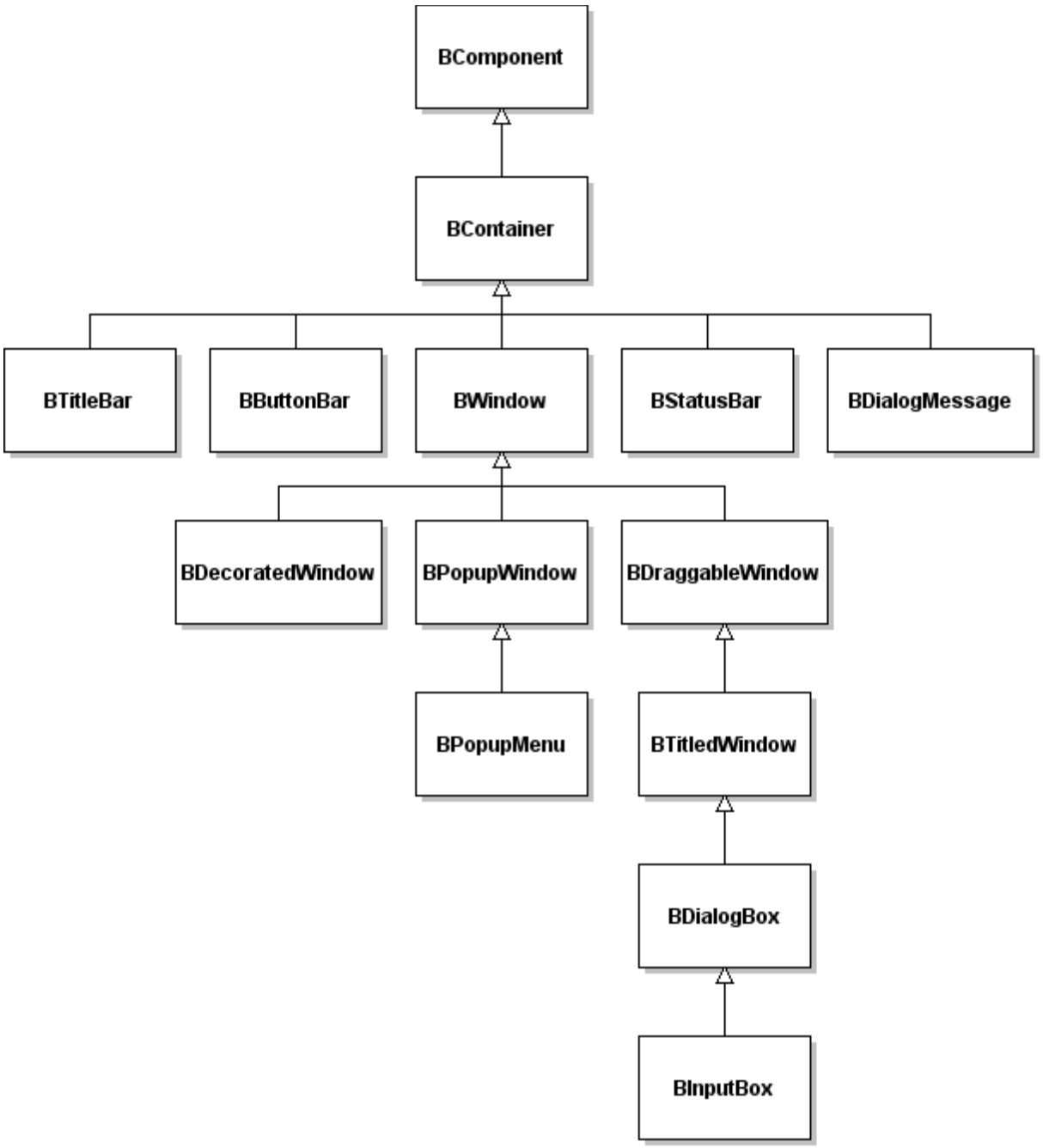


Figure 2.2.4: This shows the variety of Window and Dialog components that derive from BContainer.

Note: a BWindow may only be added to BRootNode; it may not be the child of any other component.

2.3 Style Hierarchy (or, "Why Understanding CSS Will Be Helpful")

If you've looked at the GBUI source at all, I'm sure you've seen the *.bss files that define how your GUI will look. You can see widget types, and some obvious properties that relate to appearance: font size, background image, etc. When GBUI renders each component on the screen, it decides how each widget should look based upon it's component type, and it's state.

2.3.1 States

"State" is an easy-to-understand concept that is used to describe what the component is doing: BComponent defines three states common to all widgets:

- Disabled: most components that are disabled have a darker than normal appearance used to indicate that the user can not interact with those components
- Hover: a component is in this state when the mouse cursor is within the bounds of the component. Often this is used to give a visual cue to the user that pressing the mouse button will do something
- Default: this state applies to any component that is not disabled, and the mouse is not hovering over it.

2.3.2 Styles

A component's "Style" is determined by the values of the properties for that component. The properties that can be set, their format and allowed values, and the defaults if that property is not explicitly set are:

Name: **background**
Description: This property defines what the main body of the component looks like. This may be a solid color, an image, or a blank (which basically means the component is invisible).
Format: type path/color [scale] [insets]
Valid values:

- type: image/solid/blank
- path/color:
- scale: centerxy, centerx, centery, scalexy, scalex, scaley, tilexy, tilex, tiley, framexy, framex, framey

Note:

- the second parameter is ignored if the type is "blank"
- the second parameter is a color only if the type is "solid"
- the second parameter is a path only if the type is "path"
- the third parameter is used only if the type is "path", and is optional. The default value is scalexy.
- the insets are used only if the third parameter is framexy. See **padding** for a description of the expected format.

Examples:

- **background: image "rsrc/textures/button_up.png" tilexy;**
- **background: image "rsrc/textures/button_up.png" framexy 1 2 3 4;**
- **background: solid #00000088;**

Default:

- n/a?

Name: **border, border-left, border-top, border-right, border-bottom**
Description:
Format: thickness type [color]
Valid values:

- thickness: (integer value)
- type: solid/blank
- color: #RRGGBB or #RRGGBBAA

Note:

- if a **border** class is defined, it will be used. If one is not defined, the border will be constructed by compiling the definitions of the individual sides.
- the **color** parameter will only be read if the **type** is "solid".

Examples:

- **border: 1 solid #FFFFFF;**

Default:

-

Name: **color**
Description:
Format: color
Valid values:

- color: #RRGGBB or #RRGGBBAA

Examples:

- **color #BBBBBB;**

Default:

- n/a?

Name: **cursor**
Description:
Format: name
Valid values:

-

Examples:

-

Default:

- n/a?

Name: **effect-color**
Description:
Format:
Valid values:

-

Examples:

- **effect-color: #442288;**

Default:

- n/a?

Name: **effect-size**
Description:
Format: size
Valid values:

- size: (integer value)

Examples:

-

Default:

- **effect-size: 1;**

Name: **font**
Description:
Format: family style size
Valid values:

- family: “Dialog”, “Helvetica”
- style: plain/bold/italic/bolditalic
- size: (integer value)

Examples:

- **font: “Dialog” plain 16;**

Default:

- n/a?

Name: **icon**
Description:
Format: type path/width [height]
Valid values:

- type: blank/image

Note:

- if the type is “blank”, the second parameter is the width and the third parameter is the height.
- if the type is “image”, the second parameter specifies the path to the image file. If any error is encountered loading the specified file, the icon will default to **blank 10 10**.

Examples:

- **icon: blank 7 7;**
- **icon: image “rsrc/textures/checkbox.png”;**

Default:

- **icon: blank 10 10;**

Name: **line-spacing**
Description: Amount of space to add or remove between lines.
Format: spacing
Valid values:

- spacing: (integer value)

Examples:

- **line-spacing: -2;**

Default:

- **line-spacing: 0;**

Name: **padding**
Description:
Format: top [right] [bottom] [left]
Valid values:

-

Examples:

- **padding: 5;** **// top = 5; right = top; bottom = top; left = top**
- **padding: 3 5;** **// top = 3; right = 5; bottom = top; left = right**
- **padding 4 6 2 4;**

Default:

- **padding 0;**

Name: **parent**
Description: Establishes an explicit inheritance. The class being extended must be defined before this one.
Format:
Valid values:

-

Examples:

- **parent: titlebutton;**

Default:

-

Name: **size**
Description: Overrides the component's preferred size.
Format: width height
Valid values:

-

Examples:

-

Default:

-

Name: **text-align**
Description: Specifies the horizontal alignment of text on the component.
Format: alignment
Valid values:

- alignment: left/right/center

Examples:

- **text-align: center;**

Default:

- **text-align: left;**

Name: **text-effect**
Description:
Format: effect
Valid values:

- Effect: none/shadow/outline/plain

Examples:

- **text-effect: outline;**

Default:

- **text-effect: none;**

Name: **tooltip**
Description: Used to define the style class for the tooltip.
Format: other_class
Valid values:

-

Examples:

-

Default:

-

Name: **vertical-align**
Description: Specifies the vertical alignment of text on the component.
Format: alignment
Valid values:

- alignment: center/top/bottom

Examples:

- **vertical-align: top;**

Default:

- **vertical-align: center;**

2.3.3 Determining a component's style

In "typical" (ie. VB-style) GUI programming, a widget will have a standard look, and then you can modify the appearance of each instance by modifying the text displayed, the back color, etc. The problem with this is that each application made with the same widgets tends to look very similar, since implementing a common look-and-feel across an entire application is no small task. With GBUI, just as with CSS, each widget class (button, window, etc) can be given the same look-and-feel with relatively little work, making it easy to implement an application-specific look-and-feel that is different from another application built with the same widgets. As an added bonus, the "cascading" feature means that if a particular component type does not define a value for a property, GBUI will traverse up the style hierachy looking for the next-lowest component that does define a value. By default, the style hierarchy is the same as the class hierarchy, but that can be modified by explicitly defining the style class a component should use, and/or by using the "parent" property. On top of everything, a "root" style class exists that will be checked after all other components in the hierarchy.

A component's style is resolved in the following manner:

- First by looking up the property using the component's stylesheet class (either the default, or the one explicitly set) and state
- For certain properties, the interface hierarchy is the climbed and each parent's stylesheet class is checked for the property in question. The properties for which this applies are: **color, font, text-align, vertical-align**
- Finally, the **root** stylesheet class is checked (for all properties, not just those for which we climb the interface hierarchy)

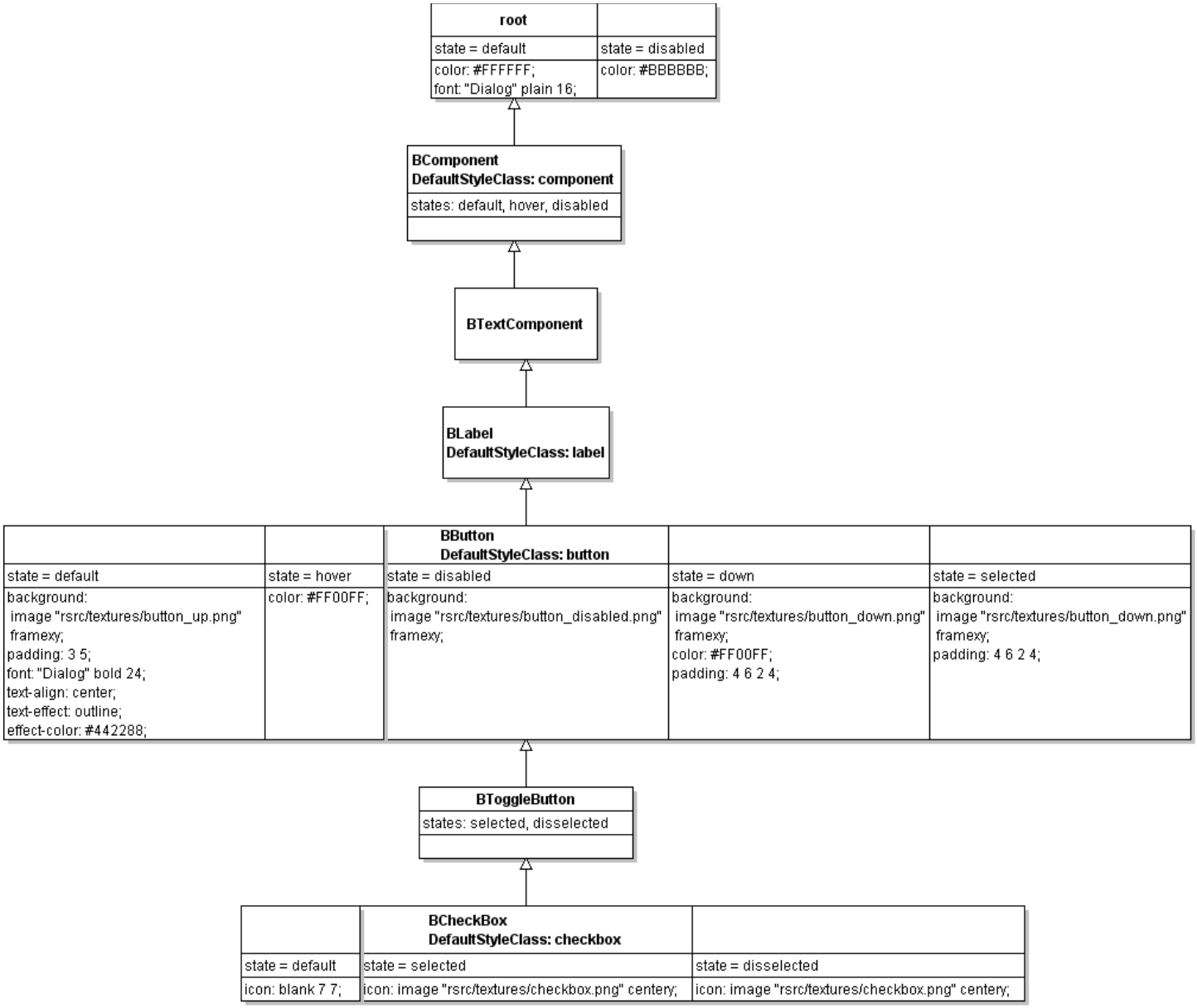

root
state = default
color: #FFFFFF;
font: "Dialog" plain 16;
state = disabled
color: #BBBBBB;
BComponent
DefaultStyleClass: component
states: default, hover, disabled
BTextComponent
BLabel
DefaultStyleClass: label
BButton
DefaultStyleClass: button
state = default
background:
image "rsrc/textures/button_up.png"
framexy;
padding: 3 5;
font: "Dialog" bold 24;
text-align: center;
text-effect: outline;
effect-color: #442288;
state = hover
color: #FF00FF;
state = disabled
background:
image "rsrc/textures/button_disabled.png"
framexy;
state = down
background:
image "rsrc/textures/button_down.png"
framexy;
color: #FF00FF;
padding: 4 6 2 4;
state = selected
background:
image "rsrc/textures/button_down.png"
framexy;
padding: 4 6 2 4;
BToggleButton
states: selected, disselected
BCheckBox
DefaultStyleClass: checkbox
state = default
icon: blank 7 7;
state = selected
icon: image "rsrc/textures/checkbox.png" centery;
state = disselected
icon: image "rsrc/textures/checkbox.png" centery;

Figure 2.3.3.1:

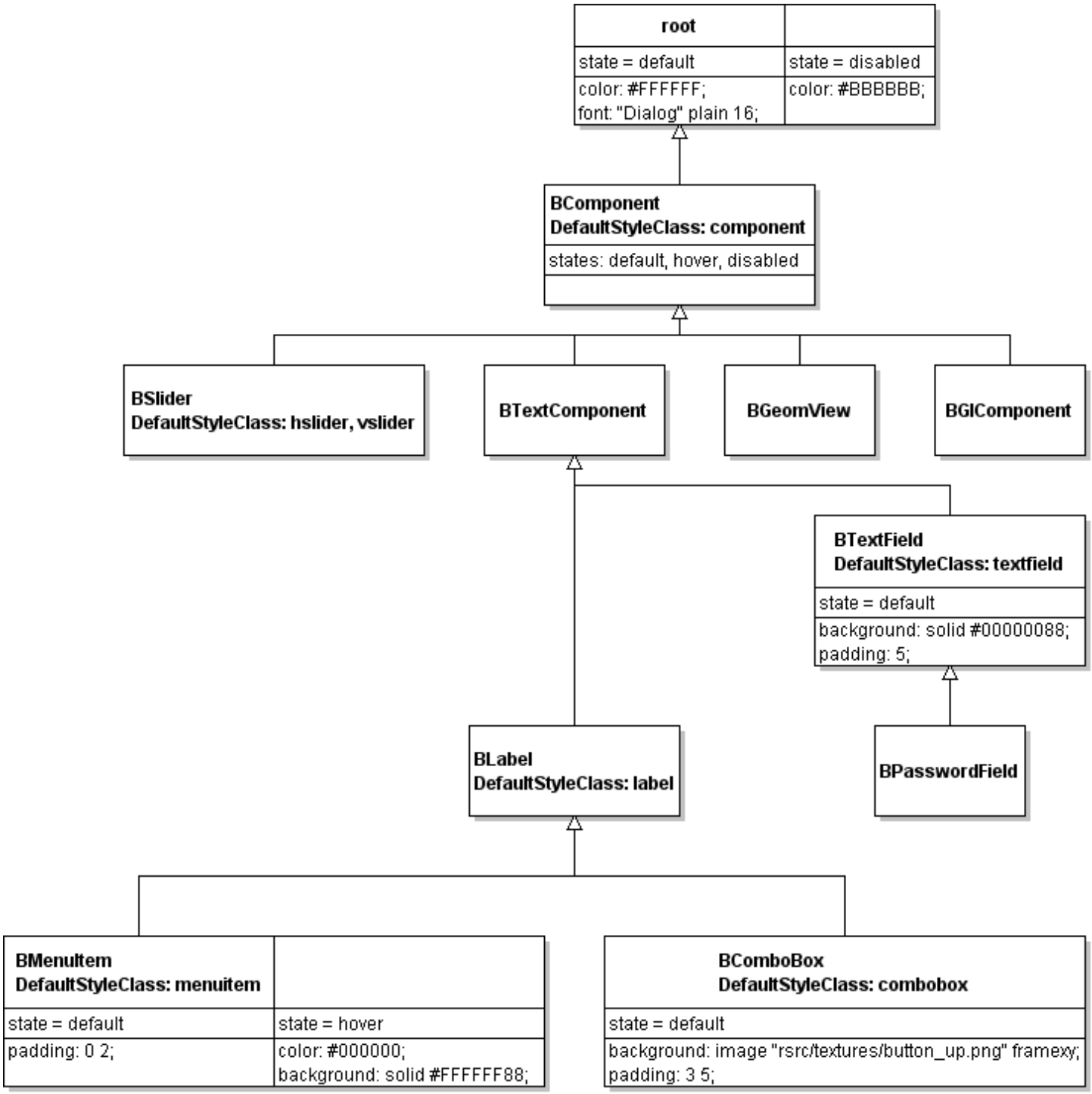


Figure 2.3.3.2:

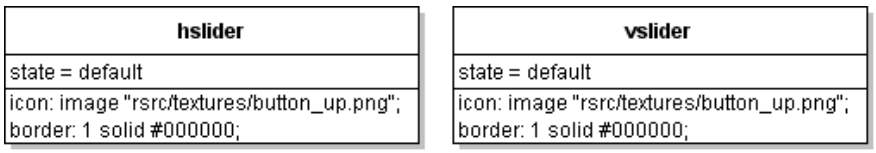


Figure 2.3.3.3:

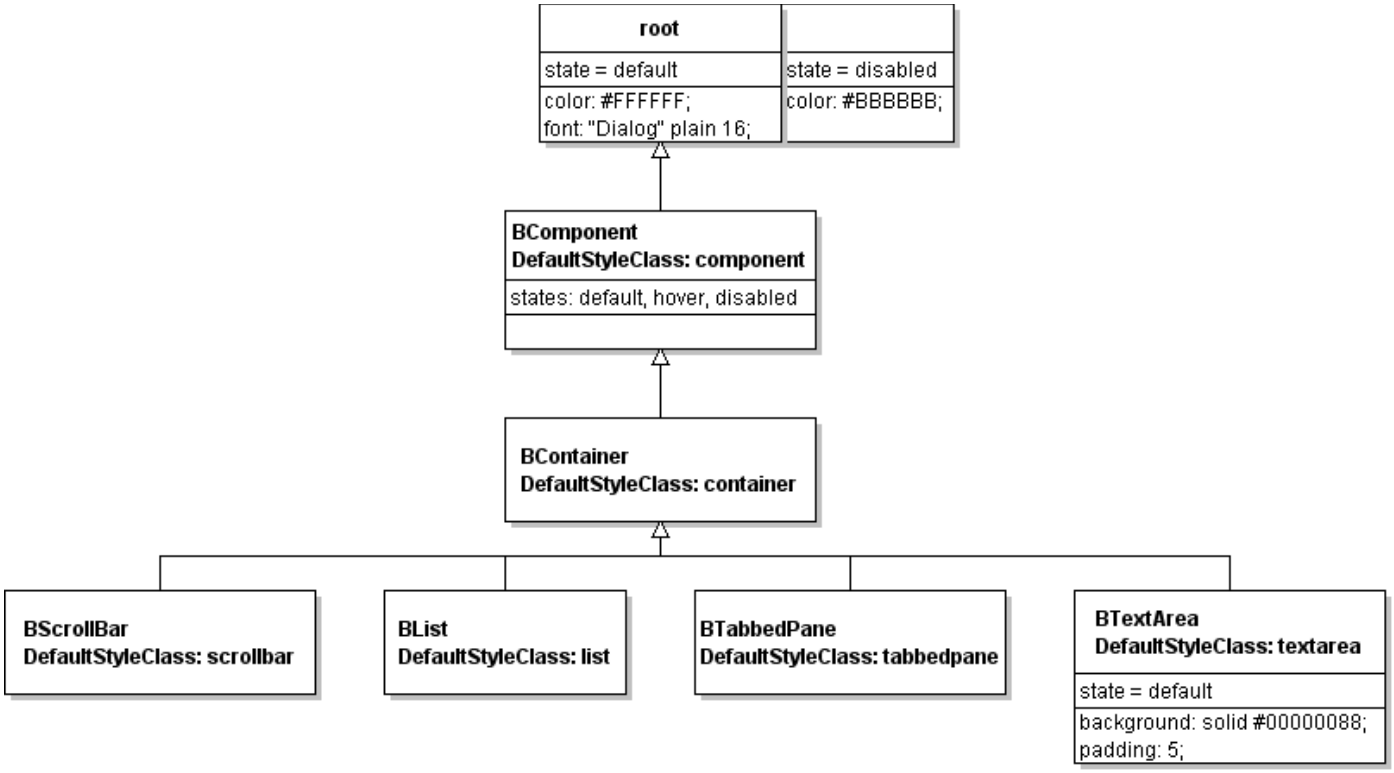


Figure 2.3.3.4:

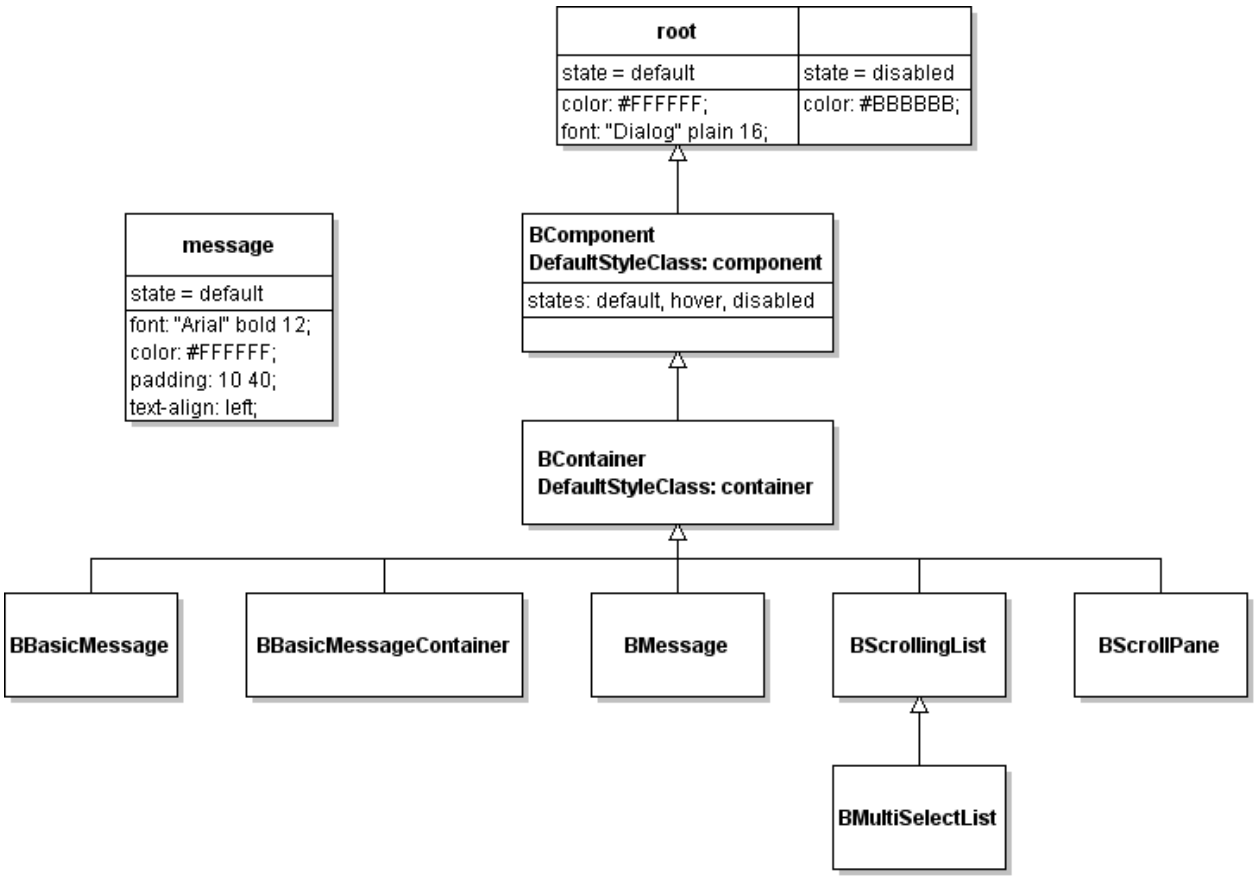


Figure 2.3.3.5:

2.3.4 Styles without classes

2.3.5 Overriding the default class style

2.3.6 Windows and their buttons

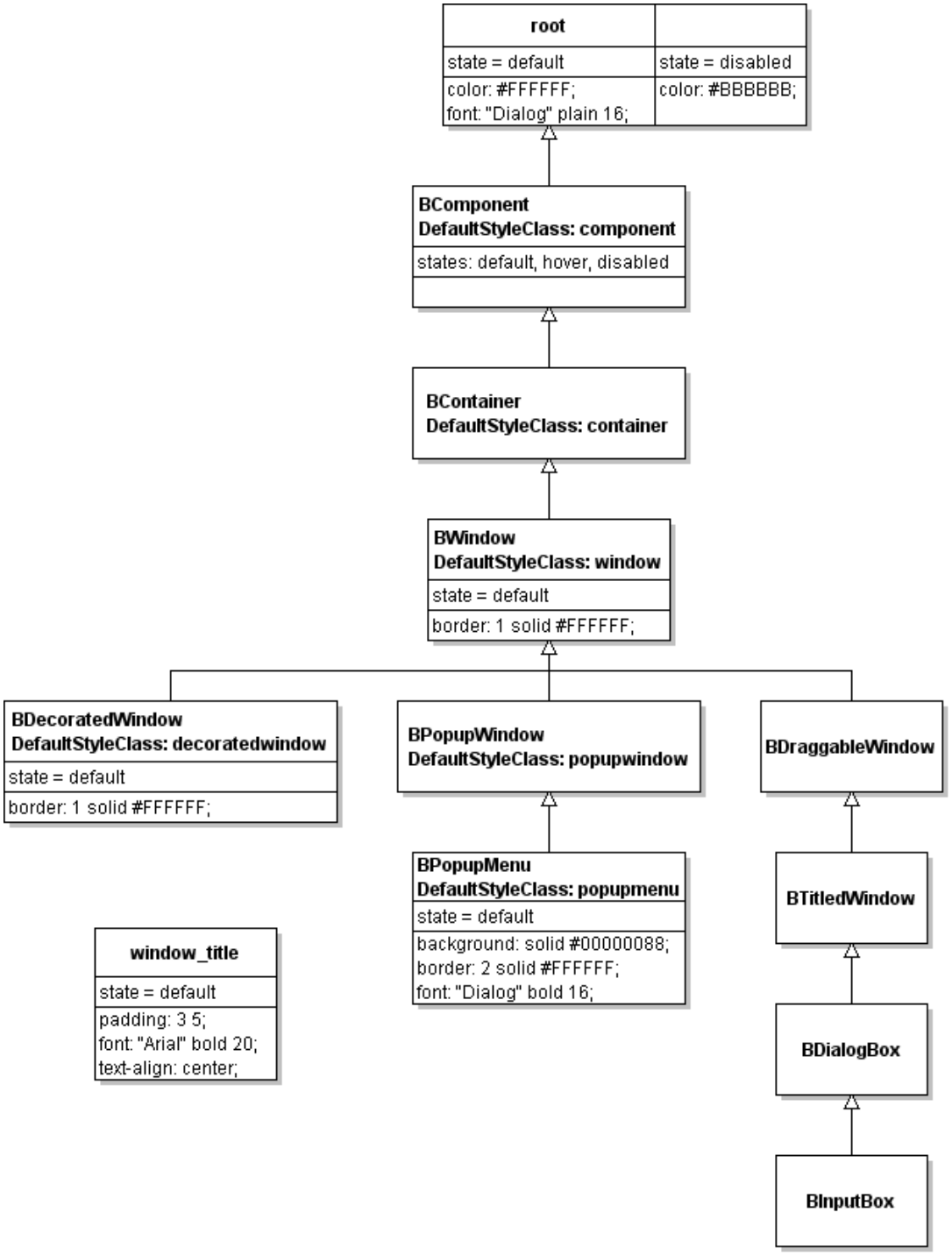


Figure 2.3.6.1:

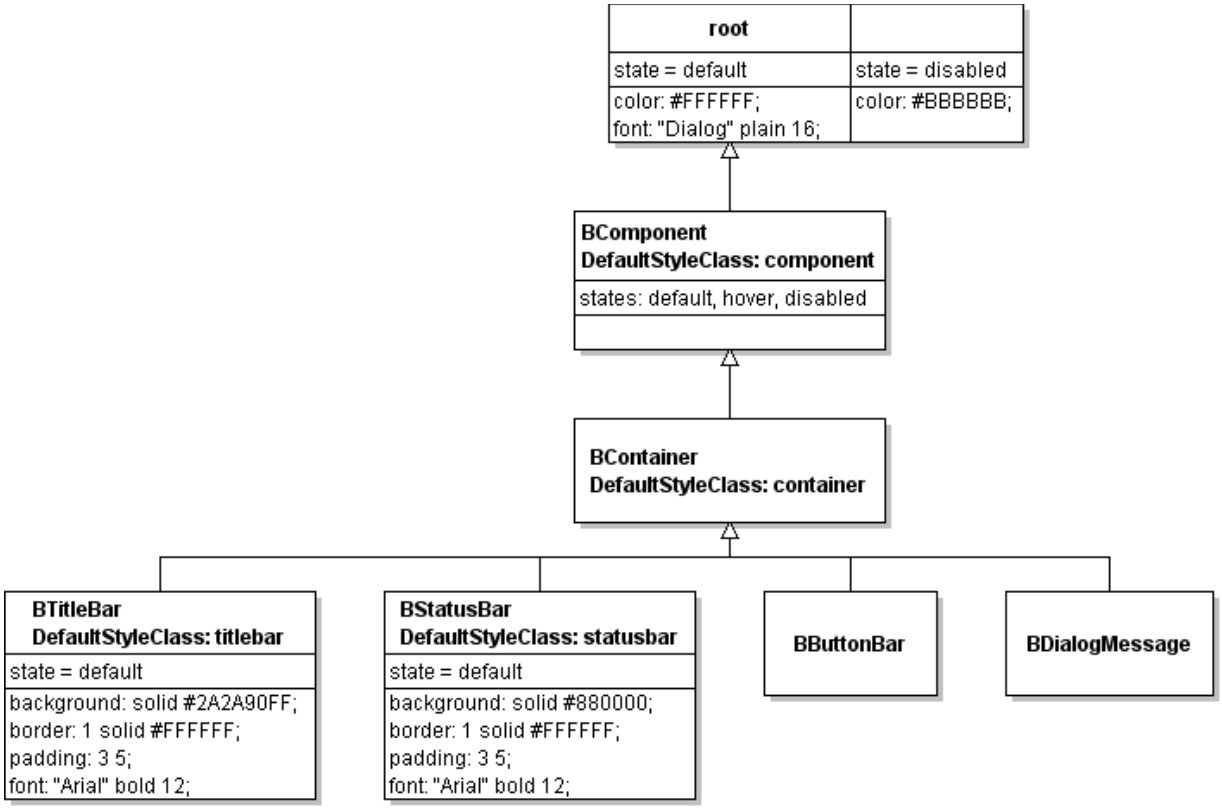


Figure 2.3.6.2:

		dialogbutton		
state = default	state = hover	state = disabled	state = down	state = selected
background: solid #000088; border: 1 solid #FFFFFF; padding: 3 5; font: "Arial" bold 12; text-align: center;	background: solid #2A2A90FF;	color: #999999; border: 1 solid #999999;	padding: 4 6 2 4;	padding: 4 6 2 4;

Figure 2.3.6.3:

		titlebutton		
state = default	state = hover	state = disabled	state = down	state = selected
padding: 1 1; font: "Arial" bold 12; text-align: center;	color: #E38833;	color: #999999; border: 1 solid #999999;	padding: 4 6 2 4;	padding: 4 6 2 4;

Figure 2.3.6.4:

closebutton	
state = default	state = hover
parent: titlebutton background: image "rsrc/textures/close.png" centerxy;	parent: titlebutton; background: image "rsrc/textures/close_hover.png" centerxy;

Figure 2.3.6.5:

minimizebutton	
state = default	state = hover
parent: titlebutton background: image "rsrc/textures/minimize.png" centerxy;	parent: titlebutton; background: image "rsrc/textures/minimize_hover.png" centerxy;

Figure 2.3.6.6:

maximizebutton	
state = default	state = hover
parent: titlebutton background: image "rsrc/textures/maximize.png" centerxy;	parent: titlebutton; background: image "rsrc/textures/maximize_hover.png" centerxy;

Figure 2.3.6.7:

titlemessage
state = default
font: "Arial" bold 12; color: #FFFFFF; padding: 5 5; text-align: left;

statusmessage
state = default
font: "Arial" bold 10; color: #FFFFFF; padding: 5 5; text-align: left;

greymessagebg
state = default
font: "Arial" bold 12; color: #FFFFFF; background: solid #999999; padding: 10 40; text-align: left;

Figure 2.3.6.8:

2.3.7 Custom style classes

2.3.8 Scrollbars

scrollbar_hthumb
state = default
background: solid #FFFFFF;

scrollbar_vthumb
state = default
background: solid #FFFFFF;

Figure 2.3.8.1:

scrollbar_lwell
state = default
background: image "rsrc/texture/button_up.png" framexy; padding: 3;

scrollbar_vwell
state = default
background: image "rsrc/textures/button_up.png" framexy; padding 3;

Figure 2.3.8.2:

scrollbar_hless
state = default
background: image "rsrc/texture/scroll_left.png";

scrollbar_vless
state = default
background: image "rsrc/textures/scroll_up.png";

Figure 2.3.8.3:

scrollbar_hmore
state = default
background: image "rsrc/texture/scroll_right.png";

scrollbar_vmore
state = default
background: image "rsrc/textures/scroll_down.png";

Figure 2.3.8.4:

3. Creating your own “look-and-feel”

- If you haven’t look at style2.bss yet, do so now
- Notice how button_up.png is used by many components; a good choice to change the look-and-feel of an application

4. Layout Managers

- AbsoluteLayout
- BorderLayout
- GroupLayout
- TableLayout

5. Putting it all together: "A Technique"

5.1 Required Reading

Many times, academic articles will be sprinkled with a long series of footnotes throughout the text, referring the user to other articles that pertain to what the author is saying. In this article, I'm going to reverse the article by listing reference works first, before I say anything that's at all original. The reason for this is simple: the work I've done is built on the foundations laid by many other people, and I believe understanding where I got my ideas from will help better explain what I've done.

5.1.1 Model-View-Controller/Model-View-Presenter/State-View-Controller

This is a software-engineering design pattern that goes by a variety of names, but what they have in common is a formal separation of data ("model" or "state"), the way that data is presented to the user ("view"), and the way the user interacts with that data ("controller" or "presenter"). It's such a widely covered topic that any Google search will turn up enough references to keep you busy reading for the next year, so I'm not going to include links to any specific articles. Besides, the next item takes this general pattern, and puts enough meat on the concept to make it very useable.

5.1.2 Presenter-First

This is a variation of the MVC/MVP/SVC pattern published by Atomic Object. Their include a great number of articles and examples explaining it on their site, and I definitely recommend you spend some time studying them. Just in case you don't, here's my quick summary: in Presenter-First, the presenter knows about the model and the view only through interfaces. The presenter is constructed with an instance of an object that implements the view interface, and an instance of an object that implements the model interface. The presenter interacts with the view and model by calling methods of their interfaces. Neither the view nor the model know anything about each other or the presenter; they can interact with the presenter only by firing events. This architecture makes it very simple to test the presenter (assumed to contain the bulk of the program's logic) by simply creating test mock objects that implement the appropriate interface.

5.1.3 Inversion-of-control/Direct-Injection

I'm not going to spend any time explaining either of these topics; I'm just going to direct you to what I consider to be ***the*** classic article on the topic: Inversion of Control Containers and the Dependency Injection pattern.

Once again, don't get too hung up on understanding everything in this article; if you do a quick skim to get the basic ideas, you'll be good to go.

5.1.4 Game States

When I started working with jMonkeyEngine, this was my understanding of game states: Managing Game States in C++. I was led to this article during my work with Ogre3D and this article: Managing Game States with Ogre. Yet again, don't obsess with the details, just understand that these articles use a different understanding of game states than jME. These articles use the more traditional software engineering definition of a states as being mutually exclusive of each other; only one state is active at a time.

5.1.5 StandardGame

Hopefully you're already familiar with StandardGame and it's definition of GameStates (that are different from the definition of GameState in the previous reference). If not, there are a number of tutorials on the jME Wiki:

- StandardGame, GameStates, and Multithreading (A New Way of Thinking)
- SimpleGame to StandardGame - An effort to gain mindshare by darkfrog

5.2 "A Technique"

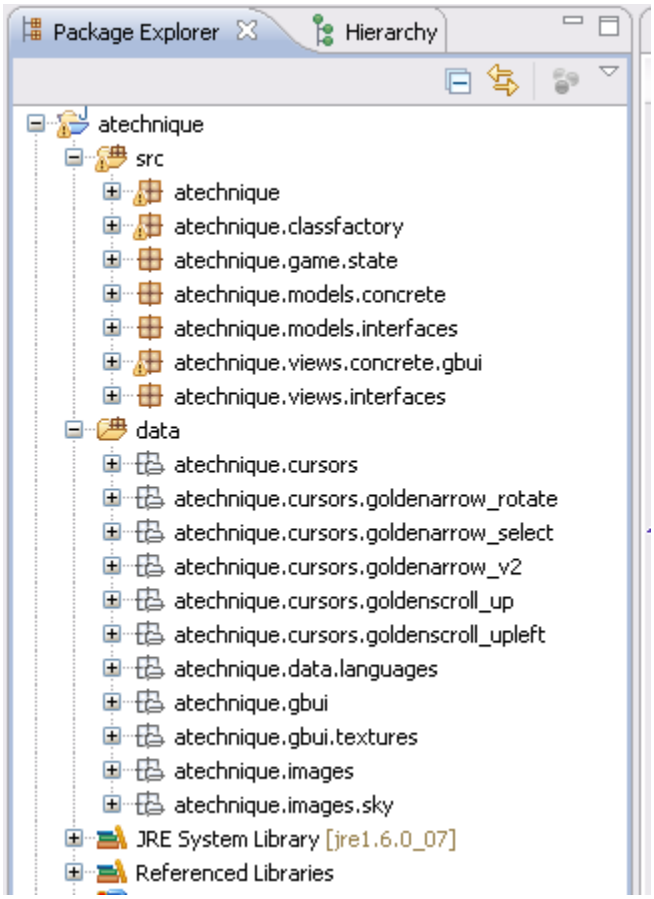
Why do I call this "A Technique"? Blame it on my military schooling and the instructors that say "Gentlemen, what I'm about to teach you is merely 'A Technique'". What it means is that I'm going to show you one way of doing something. I'm not going to claim that it's the "One True Way", or that it's the best way. I'm not even going to claim that it's a correct way. I will say that it "Works For Me". Try it for yourself and learn from it. I can almost guarantee there will be parts of it that you don't like. I hope there are parts you do like but it won't hurt my feelings if you don't. Take what you like, find your own way for the parts you don't, and move forward.

5.3 What we're going to create

Here's the premise of this sample application: we're going to create a 3D, online version of a tabletop miniatures wargame. One player will start the application and select a campaign. A campaign is a series of related battles or scenarios, the results of each one influencing the others. For this game, campaigns and their scenarios will either ship with the game, players can purchase them as expansion packs, or they can create their own (if an appropriate editor is provided). The player can either start a new campaign from the beginning, or resume one that they have already progressed in. After the campaign and scenario have been selected, the player will start a server connection. They will specify what their username will be, what port the server will listen on for incoming connections, and which faction (defined in the scenario) they will play in the upcoming battle. The game will then load and display the appropriate terrain and wait for incoming connections. Another player will start the game, connect to an existing game, and enter the server address and port, and their username. Once their client connects to the server, they will select which faction they will play. Now that everyone has connected and sides chosen, the players will play their game. At any point, they can save the game where they are, to resume at a later time.

5.4 How we're going to do it

I've separated my code and the media resources so I can hopefully update the code at a later date and save you the bandwidth of downloading the media again. I'm not going to go into a lot of detail on setting up the project here. Refer to the chapter on setting up GBUI and understand this project is practically a parallel to that. Once that's been done successfully, your Eclipse project should resemble the following:



5.4.1 Main

```
package atechnique;

import com.jme.renderer.ColorRGBA;
import com.jmex.game.StandardGame;

import atechnique.classfactory.ClassFactory;
import atechnique.game.state.GameManager;

public class Main {
      private static StandardGame _game;

      /**
       * @param args
       */
      public static void main(String[] args) throws InterruptedException {
            _game = new StandardGame("ATechnique", StandardGame.GameType.GRAPHICAL,
ClassFactory.getGameSettings());
            _game.setBackgroundColor(ColorRGBA.darkGray);
            _game.start();

            ClassFactory.initialize(_game.getCamera());

            GameManager.getInstance().start(ClassFactory.getMainMenuPresenter());
      }

    public static void exit() {
        _game.shutdown();
    }

    public static void changeResolution() {
      _game.recreateGraphicalContext();
    }
}
```

Let's look at what we have here: at first glance, this looks like a fairly standard (no pun intended) start of a StandardGame application. We declare a class-level StandardGame instance, then create it, set a background color, and start the game. But what's with the references to ClassFactory and GameManager? I'll go into more detail later, but for now let me just say that ClassFactory is my implementation of a Direct Injection container. It is a Singleton object that provides access to all of the shared objects that will be referenced throughout the application. More than just providing access to these objects, the ClassFactory is responsible for creating them. The GameManager is my implementation of my game state machine, as described in the above required reading. The call to the start method is initiating the state machine. The two public methods (`exit` and `changeResolution`) are here because it's the only way I could find to provide access to the instance of StandardGame. I don't really like it, it doesn't feel right, but it is what it is.

5.4.2 ClassFactory

For reasons of space, I'm not going to reprint the entire class here. Follow along in the source as I describe each item:

```
package atechnique.classfactory;
```

The package declaration. If you found the source file, you already knew this. If not, now you do! ☺
I don't think there's anything in the imports section or class declaration that needs to be explained. Just after the declaration for the class are the declarations for all the class member variables. You may notice that the vast majority are `private` and `static`. There is one that is `public`, but I generally believe that `public`, and even `protected`, member variables are a bad idea. When I violate this guideline, it's generally a shortcut to what I consider to be a "proper" solution, and I usually regret it later. This `public` variable is no exception: I haven't yet regretted it, but it's definitely a shortcut.

Before I go further, let me explain the principle behind this entire class: in the required reading section, I referred to Inversion-of-Control and Direct-Injection. This class is my implantation of a Direct Injection container; it's job is to construct objects and make them available to the application. It also lets me use the concepts of global variables and singleton objects, while also avoiding implementing the objects I need as singletons, so I can actually use automated unit tests on them. (In case you haven't already encountered the issue, singleton objects are notoriously difficult to test with automated unit test suites). I've used Spring.NET in a previous application, but ended up ripping it out and replacing it with a class similar to what I have here. Using Spring just required too much overhead, and caused problems during debugging. I prefer to have my mistakes pointed out to me by the compiler, rather than at run-time. When other developers (not familiar with Spring, or how to configure it) tried to work on the application, they just got confused and stumped. So maybe I'm just not smart enough to see the benefit of a container that puts together objects at run-time based on a configuration file, even though I know exactly how I want the objects constructed at compile time and I won't ever change that, but it wasn't worth the hassle. I'm still not really happy with the name of this class but I haven't found a better one yet, so it stands.

Now to continue with the code: with this type of container, I have some control over exactly when and how I create all the objects I need, but eventually they do need to be created before they can be used. One simple technique would be to create all the objects in the constructor of the class. But this doesn't work so well with a bunch of `static` members. One way to work around this is to include an `Initialize` method that can be called to do all necessary creation. This gives us explicit control over when we do that creation. But here was the problem I ran into: I knew that the game settings was going to be an object that I wanted to have access to through this container. I also knew I wanted to use the StrategicHandler in my main game state. StrategicHandler requires an instance of the camera when it is created. The StandardGame creates the camera, but requires a reference to the game settings when it is created. So if the `initialize` method was going to create the StrategicHandler object, it would need a reference to the camera created by the StandardGame. We should call `initialize` after we create the StandardGame. But since StandardGame requires the game settings, now we need to call `initialize` before we create the StandardGame. Since we can't call `initialize` both before and after we create the StandardGame, we have a problem. The solution I chose was to use a static initializor to create the game settings so we could use that when we create the StandardGame, and we then use the camera created to call `ClassFactory.initialize()`. This explain the static initializor you see in the ClassFactory declaration, as well as most of the Main method.

In the `initialize` method, you can see a lot of reference to the mouse and cursors. The reason for this is that I wanted to control the appearance of the mouse cursor consistently throughout the entire application so I don't have to recreate the same logic all through the application.

Finally, we get to some real meat. The rest of the `initialize` method creates the actual states (in the StandardGame sense) we will use in our application. Note that for each triplet, the declarations of the view and

model are for interfaces, but here we specify concrete objects that implement those interfaces. We then pass those to the constructor of the presenter. Finally, we add the new state to the GameStateManager, but this is typical for any StandardGame application. Why do I specify the complete path for the view object when an `imports` statement could have done the same thing? Because this is where I have changed the actual view class. When learning jME GUIs, I started with FengGUI, moved to JMEDesktop, then to a custom hud implementation, and finally GBUI. By creating different packages, I could (and did!) have a view class for each of the above libraries, all implementing the same interface. As I moved from one library to another, I could do it one view at a time simply by modifying the class path to the view I wanted.

The rest of the class is fairly straight-forward: public accessors to the private member variables. The notable exception is at the end of the class:

```
public static InGamePausePresenter getInGamePausePresenter() {
      return _inGamePausePresenter;
}

public static InGameState getInGameState() {
      if (_inGameState == null) {
        _inGameState = new InGameState(_camera);
        GameStateManager.getInstance().attachChild(_inGameState);
      }

      return _inGameState;
}
```

You'll notice that `getInGamePausePresenter` merely returns the class that we created in the `initialize` method. On the other hand, the `_inGameState` object is not created in the `initialize` method: rather, the `getInGameState` method checks if the `_inGameState` method has been created yet, and creates it if it has not. I think of this as "lazy" instantiation, deferring the creation of the object until it is first needed. The trade-off of lazy instantiation versus the `initialize` method is merely a matter of when we will make the user wait for the object(s) to be created: at the game's startup, or during play of the game. I started with lazy instantiation for all objects, but didn't like the delay during menu navigation, so I create the GUI states when the game starts up. Since I expect the `_inGameState` to have to load data specific to the selected scenario, which won't be known until the user selects it, I use lazy instantiation for that object.

5.4.3 GameManager

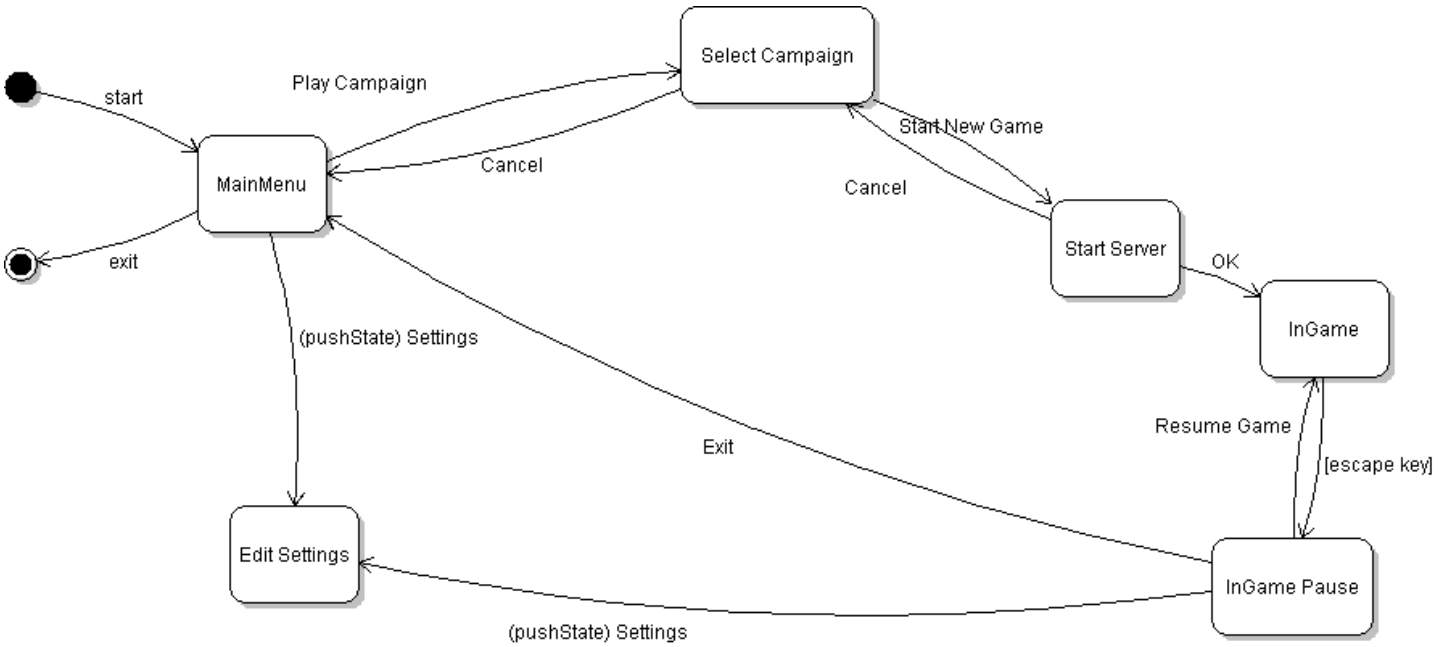
This class is the implementation of the state machine, using the concept of mutually-exclusive states. I know the multiple definitions of "Game State" is confusing, and I apologize for it, but I'll try to make it clear which type I mean when I can. This class is meant to be a singleton object and is implemented as such; hence the `getInstance` method. I'm sure that simple static methods would work just as well, but this is a remnant of this class's original implementation in C++. Also, if you're wondering why this class is implemented as a singleton, rather than being created by the ClassFactory, my stock answer is going to be that it serves as an example of the two different techniques. Yeah, yeah, that's the ticket…

Anyway, we have created an interface called `ATechniqueGameState` as follows:

```
public interface ATechniqueGameState {
	void enter();
	void exit();
	void pause();
	void resume();
}
```

If you need any more explanation for this interface, refer back to the "Managing Game States using C++" article.

The GameManager class declares a stack of objects that implement the `ATechniqueGameState` interface. The object at the top of the stack represents the current state (in the State Machine sense) of the application. This class lets us specify a new state for the application, and it will automatically call the appropriate methods of the new and old states to ensure all transitions are done properly. `pushState` lets us change to a new state, and then we can call `popState` to return to the original state. We can (and will) use this feature when we have a state that we will change to from more than one other state. This is probably a good time to describe the different game states we'll be using during this game:



Note that if the current state is MainMenu, and we `pushState` to the EditSettings state, `popState` will return us to MainMenu. If the current state is InGamePause and we `pushState` to the EditSettings state, `popState` will return us to InGamePause.

5.4.4 Translator

Maybe I'm getting ahead of myself, but I'd like to think that my game will be so popular that it will be played be people all over the world, including those that don't speak English. Since I know that localization of strings can be a significant effort when done as an afterthought, I want to include the capability from the beginning. Accordingly, I'm going to create an object that will be responsible for providing me with the correct text, without me needing to know what the current language is.

```
package atechnique.views.interfaces;

import java.util.ArrayList;

public interface ITranslator {
	ArrayList<String> getTranslatedPhrases(ArrayList<String> translationTags);
}
```

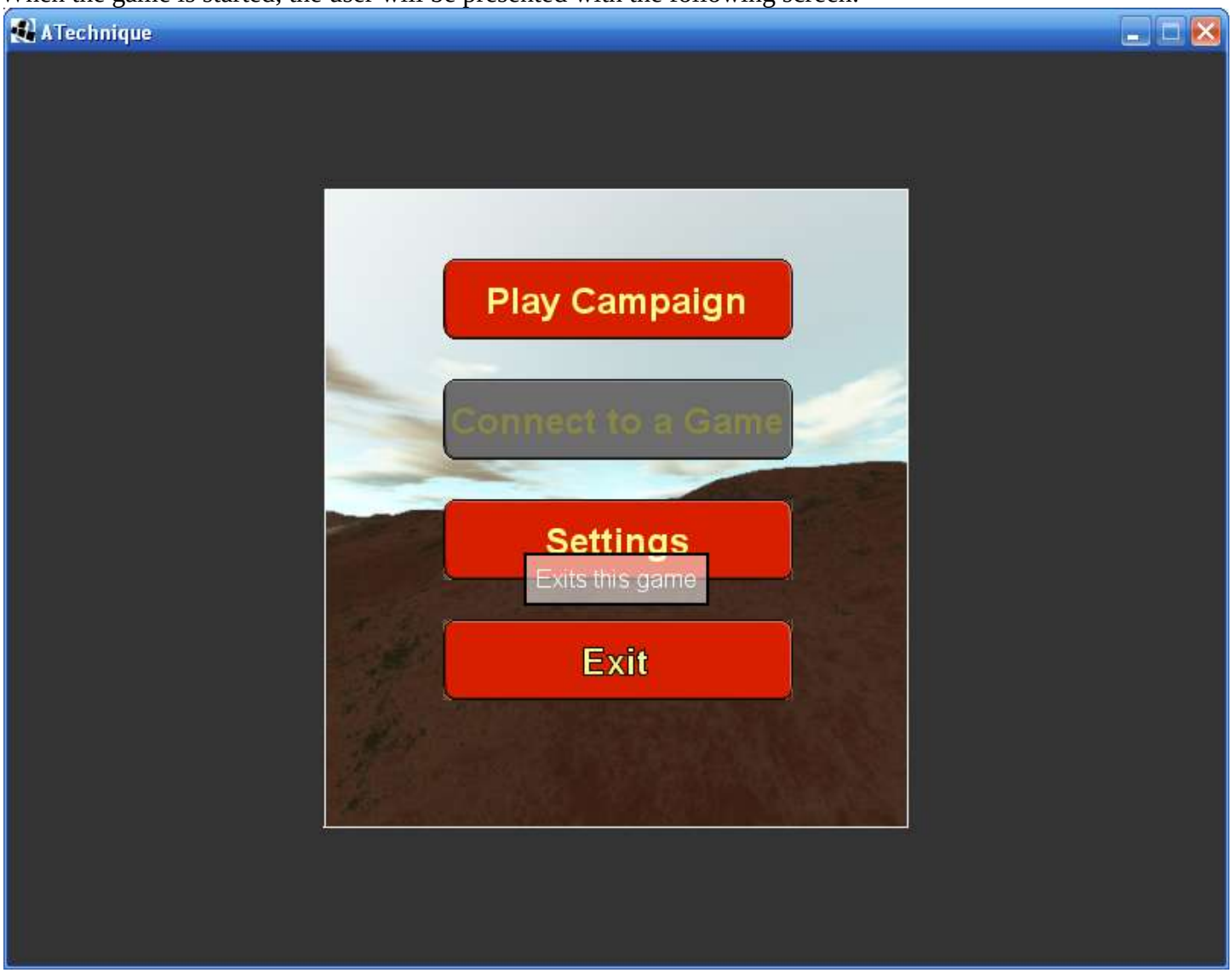
This interface defines a single method. This method expect to receive an array of strings that represent the tags to be translated. This method will return an array of phrases in the appropriate language. My current implementation is as follows:

```
package atechnique.classfactory;

import java.util.ArrayList;
import java.util.Locale;
import java.util.MissingResourceException;
import java.util.ResourceBundle;

import atechnique.views.interfaces.ITranslator;

public class Translator implements ITranslator {
	private Locale _currentLocale;
	private ResourceBundle _messages;

	public Translator(String language, String country) {
	  _currentLocale = new Locale(language, country);
	  _messages = ResourceBundle.getBundle("atechnique/data/languages/MessagesBundle",
_currentLocale);
	}

	@Override
	public ArrayList<String> getTranslatedPhrases(ArrayList<String> translationTags) {
		ArrayList<String> translatedPhrases = new ArrayList<String>();

		if (translationTags != null) {
			for (int idx = 0; idx < translationTags.size(); idx++) {
				try {

	translatedPhrases.add(_messages.getString(translationTags.get(idx)));
				} catch (MissingResourceException e) {
					// Just add the tag surrounded by <>
					translatedPhrases.add("<" + translationTags.get(idx) +
">");
				}
			}
		}

		return translatedPhrases;
	}
}
```

This uses the ResourceBundle feature of Java. Another implementation might use a database, or even a web service. The implementation really isn't important. When testing this, I found that if I tried to fetch a phrase that didn't exist in my current bundle, and exception would be thrown. I didn't like that, so I modified it to return the tag surrounded by less-than/greater-than symbols so I could know I needed to add the tag and phrase to my bundle without needing to do so immediately.

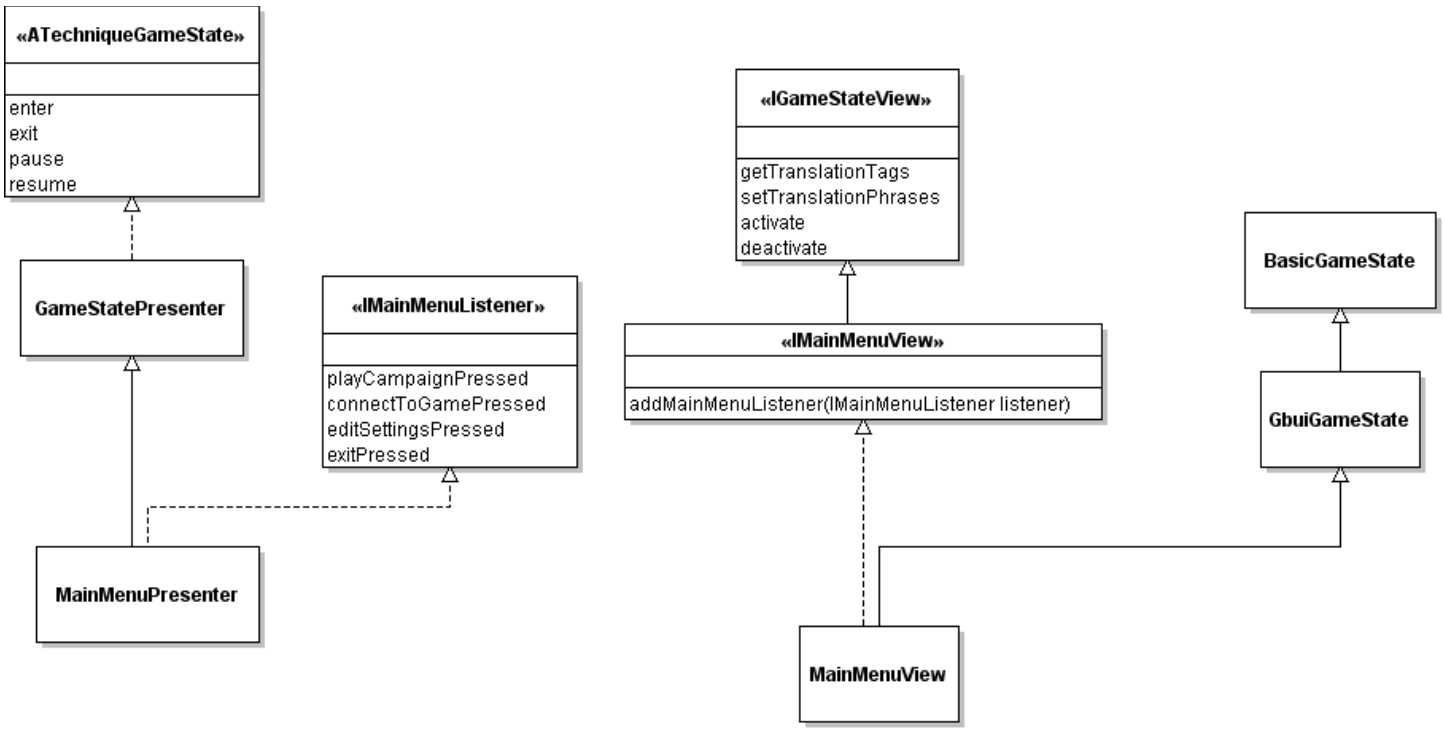
5.4.5 Main Menu

When the game is started, the user will be presented with the following screen:



Unfortunately I can't get the mouse cursor to appear in this screen-shot, but it is hovering over the "Exit" button, causing the text to be outlined, and a tooltip to be displayed. Note that the "Connect to a Game" button is disabled (because I haven't implemented that yet).

In order to explain how this is created, let me display the following class diagram:



Note the following items:

- GameStatePresenter provides a default implementation of the ATechniqueGameState interface
- MainMenuPresenter inherits from GameStatePresenter, making it a GameState (in the State Machine sense) that can be transitioned by the GameManager
- GameStatePresenter has a reference to IGameStateView, and MainMenuPresenter has a reference to IMainMenuView. When the MainMenuPresenter is created, it is given a reference to a MainMenuView object
- Since GbuiGameState inherits from BasicGameState, MainMenuView is a GameState (in the StandardGame sense) that can be activated and deactivated by the StandardGame GameStateManager
- In the required reading, I mentioned that a presenter will call methods on the view's interface, and the view will fire events that will be caught by the presenter. The IMainMenuListener represents "events" that are "fired" by the view, since I don't know how to do a true event/handler implementation in Java as I do in C#. The presenter implements the IMainMenuListener interfaces and calls addMainMenuListener on the view to hand a reference to itself

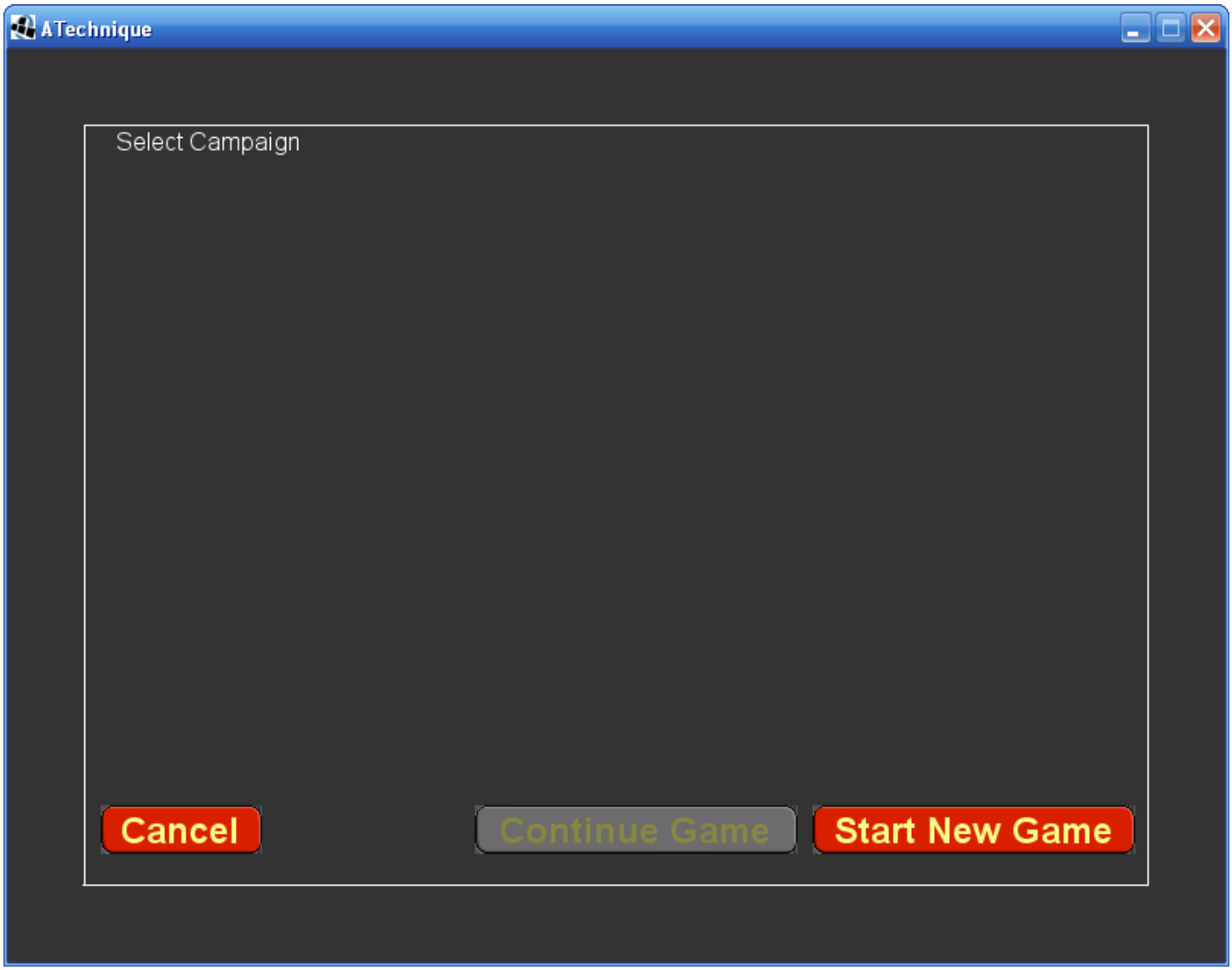
When we change to a new GameState (in the StateMachine sense), the `enter` method (defined by the `ATechniqueGameState` interface) of that state is called. The default implementation by `GameStatePresenter` is to call the `activate` method of the IGameStateView interface. GbuiGameState provides a default implementation of `activate` by calling `setActive(true)`, a GameState (in the StandardGame sense) method. By putting this function in GbuiGameState, we can create each view without worrying about these details.

If you look at the code in the constructor of `MainMenuView`, you'll see it creates a GBUI window and controls, and displays them. It responds to input events by calling methods of the `IMainMenuListener` interface, thus keeping all game and control logic in the presenter and out of the view. When this view implements the `getTranslationTags` and `setTranslationPhrases` methods of the `IGameStateView` interface, the code merely needs to return a list of tags to be translated, and then display the returned text on the appropriate control, respectively. I believe the code is fairly self-explanatory, but then me know if you have any questions about it.

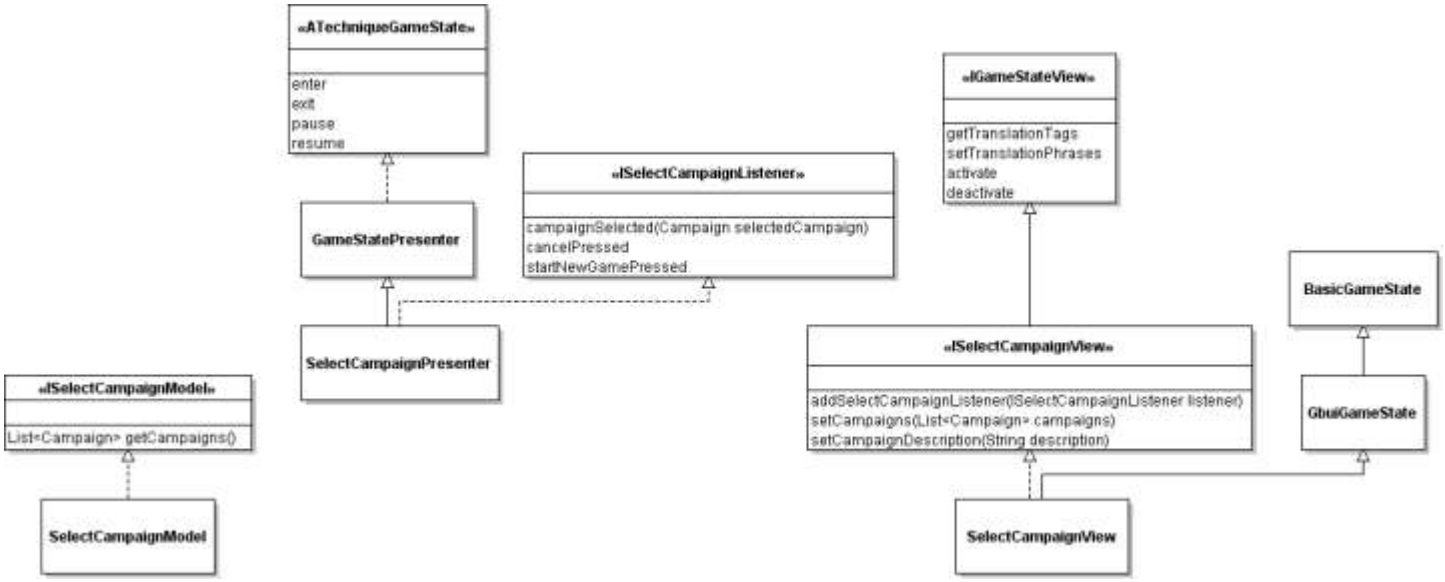
A note about the background image: I know I could have defined it in the style class definition. However, I didn't want the image in every window I created, and I was too lazy to define a custom style class. Since that is probably a "better" solution, it will probably end up there, but it's not there now.

You may be wondering where the Model portion of the MVP triad is: simply put, it ended up being an empty interface and I didn't anticipate it ever having anything to do. Rather than dogmatically adhering to the architecture, I decided I would be better off just removing the interface and it's implementation.

5.4.6 Select Campaign



I know what you're thinking: it looks a little incomplete. That's because the implementation is very specific to my game, but it's not critical for the architectural concepts I'm trying to explain but it might have distracted and confused you. So just understand that on this screen, I'm going to present a scrolling list of all the campaigns in the appropriate directory. When the user clicks on a campaign, I'll present a narrative description, and display the "progression tree" of all the scenarios in the campaign, and how they are connected. Each campaign will have a flag stored indicating if they're new, or in-progress. If new, the "Start New Game" button will be enabled. Oterwise, the "Continue Game" button will be enabled. For the purposes of this tutorial, this is just another state (in the State Machine sense) that the user can view and interact with.

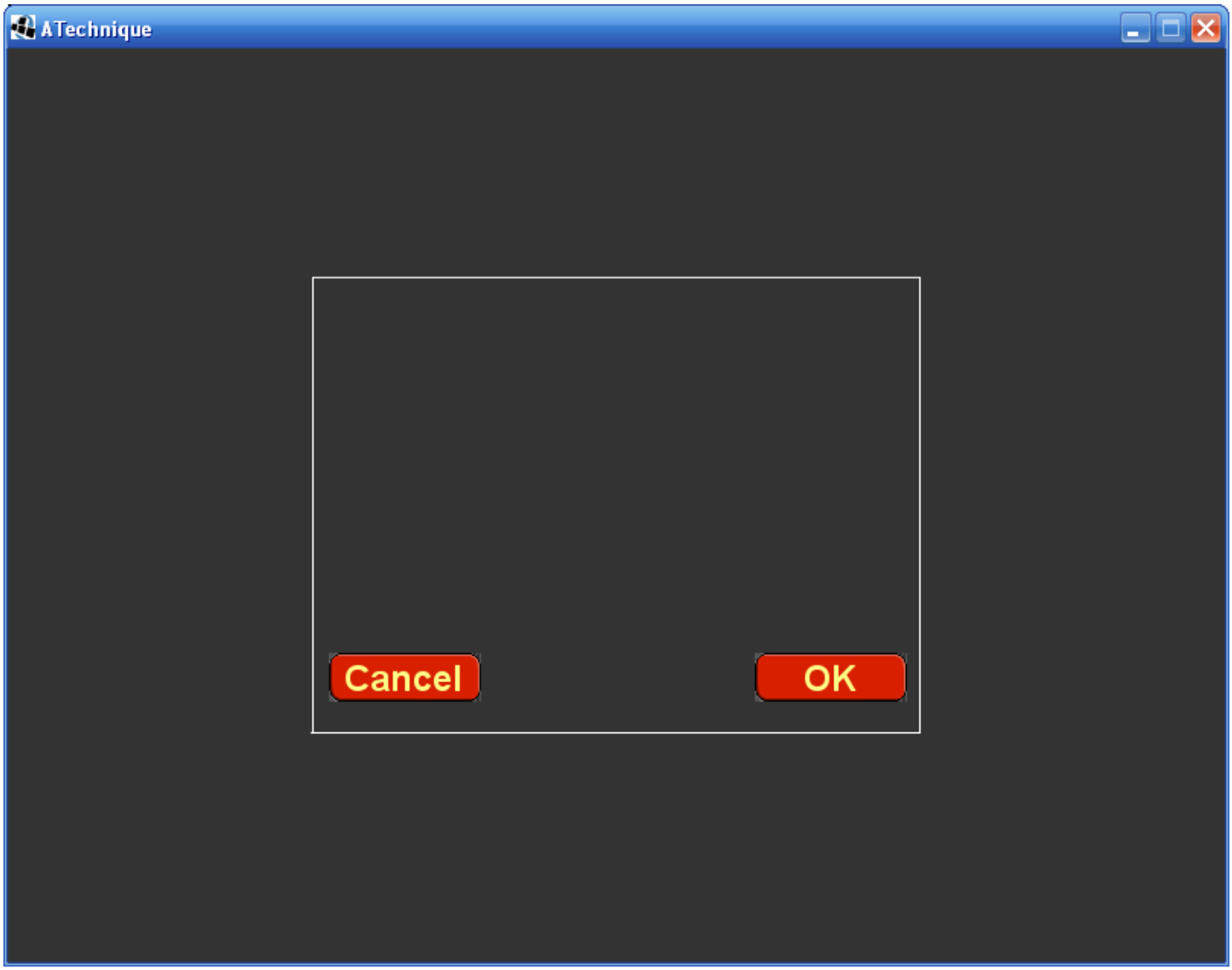


Here's how all these classes and interfaces work together:

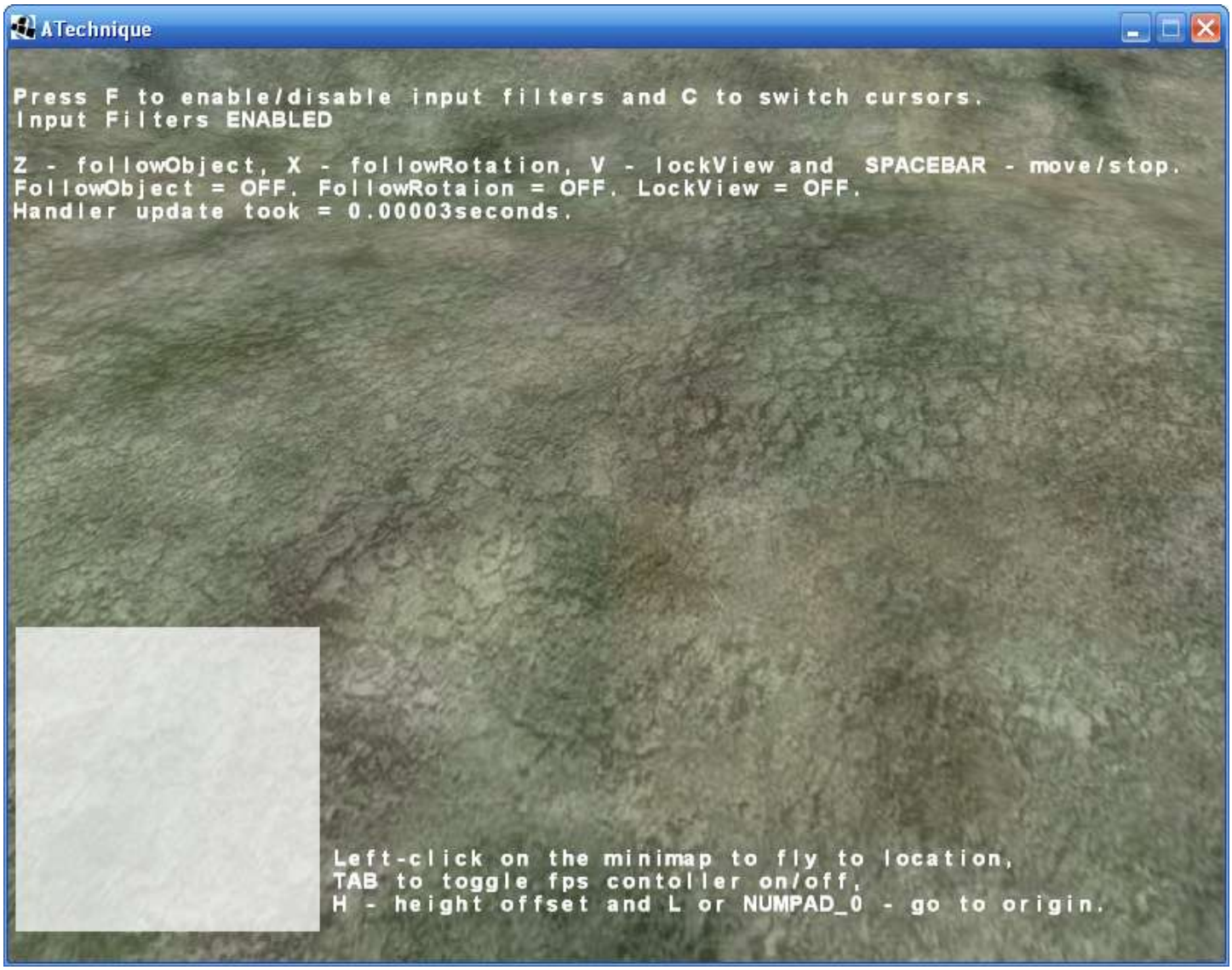
- ISelectCampaignListener again represents the events that the view would fire and the presenter would handle. Instead, these are methods of the presenter that the view will call
- When the SelectCampaignModel is created, it will automatically compile a collection of the available campaigns, and all their details
- When the Presenter is created, it will get the collection of campaign from the model, and push them to the view for display
- When the user selects a campaign in the list, the view will call the `campaignSelected` method of the presenter
- The presenter will get then push the description to the view for display

I'm expecting that when you first saw the class model for the Main Menu, the number of interfaces and states seemed somewhat daunting. Hopefully now you can see that many of them can essentially be ignored once you understand what they are doing, and you can concentrate on the specific implementation of your game.

5.4.7 Start Server

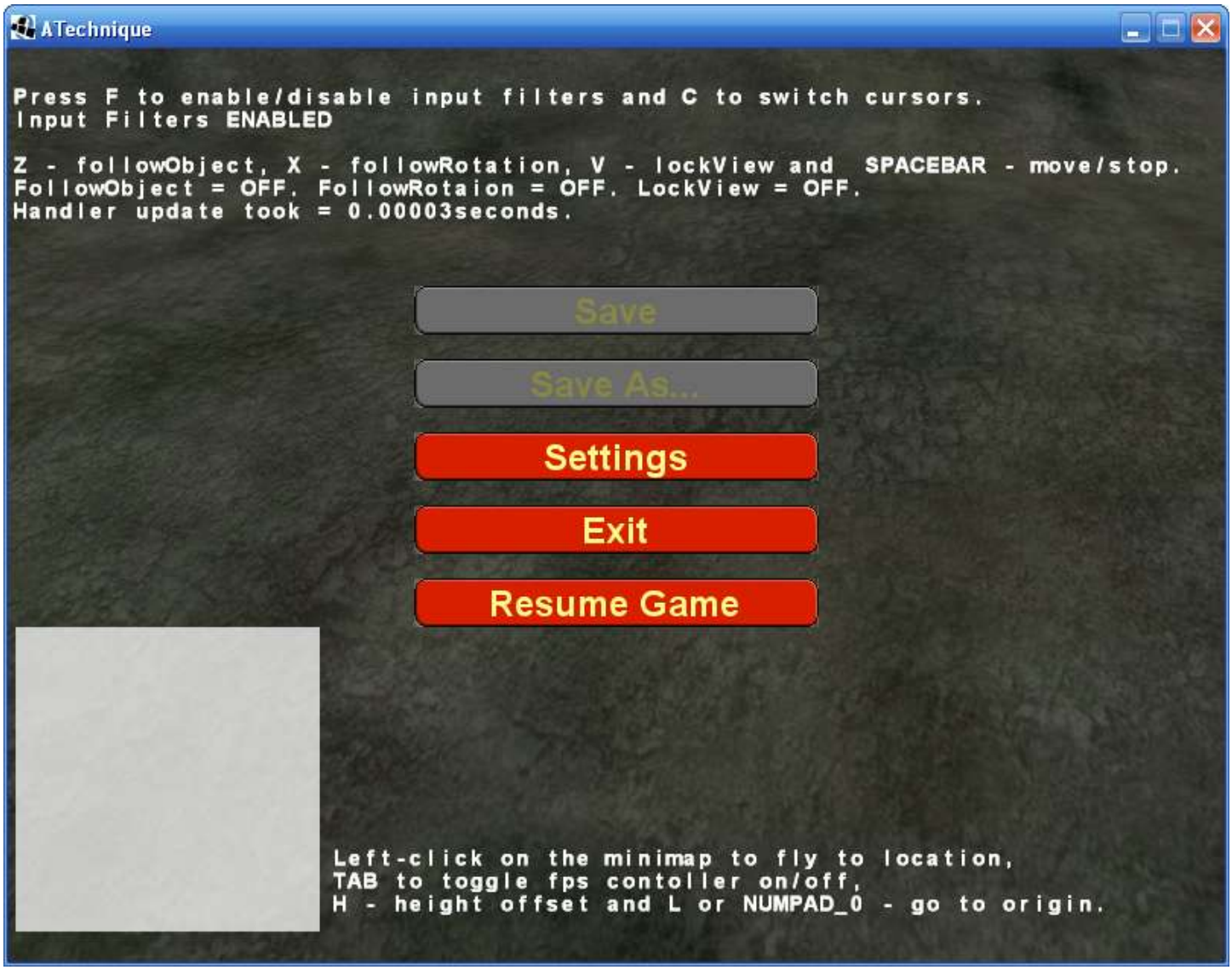


5.4.8 InGame



This state is essentially the StrategicHandler demo, converted to a StandardGame GameState. I don't have the mouse cursor working perfectly, but most of the functionality is here, or at least enough to showcase jME.

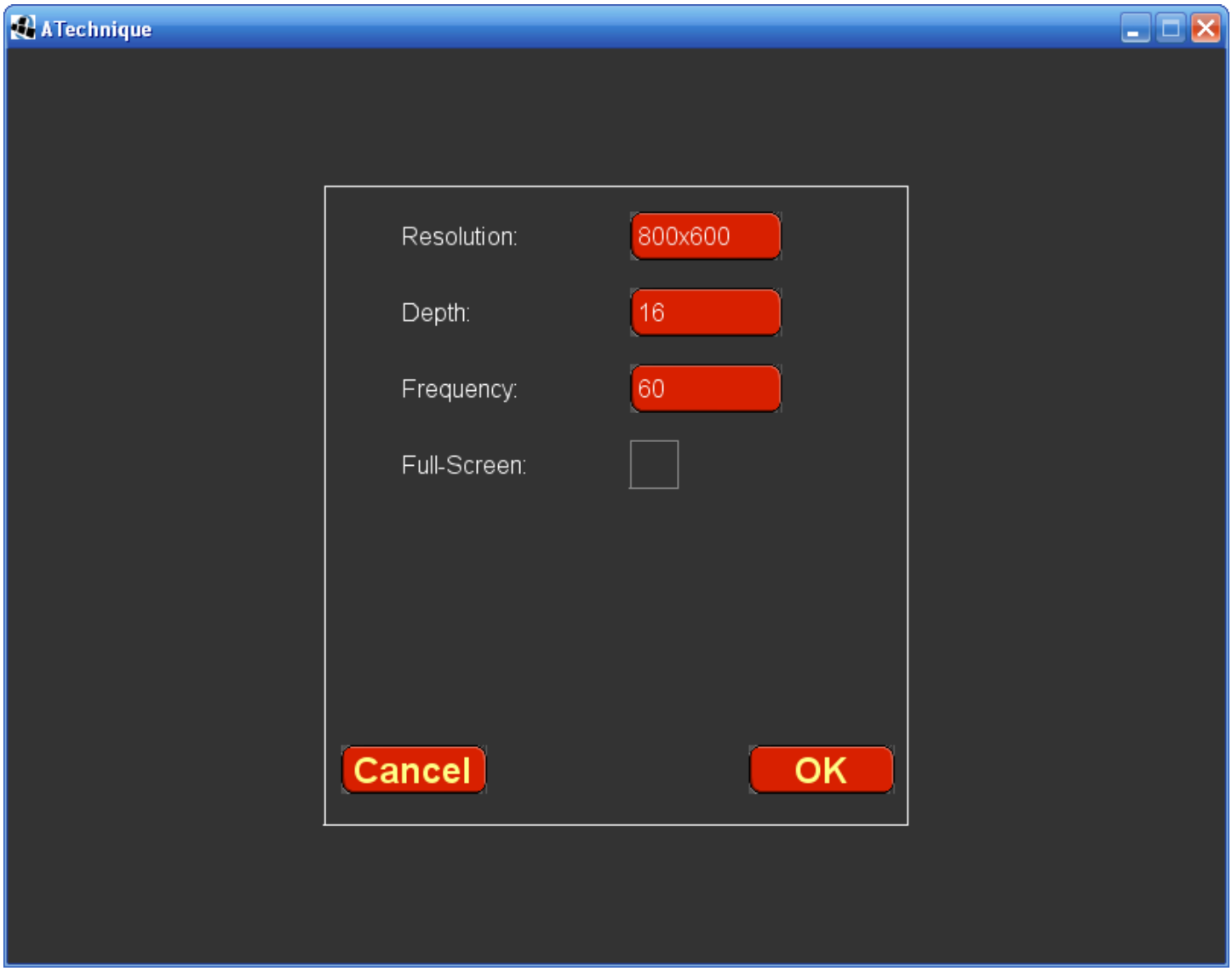
5.4.9 InGamePause



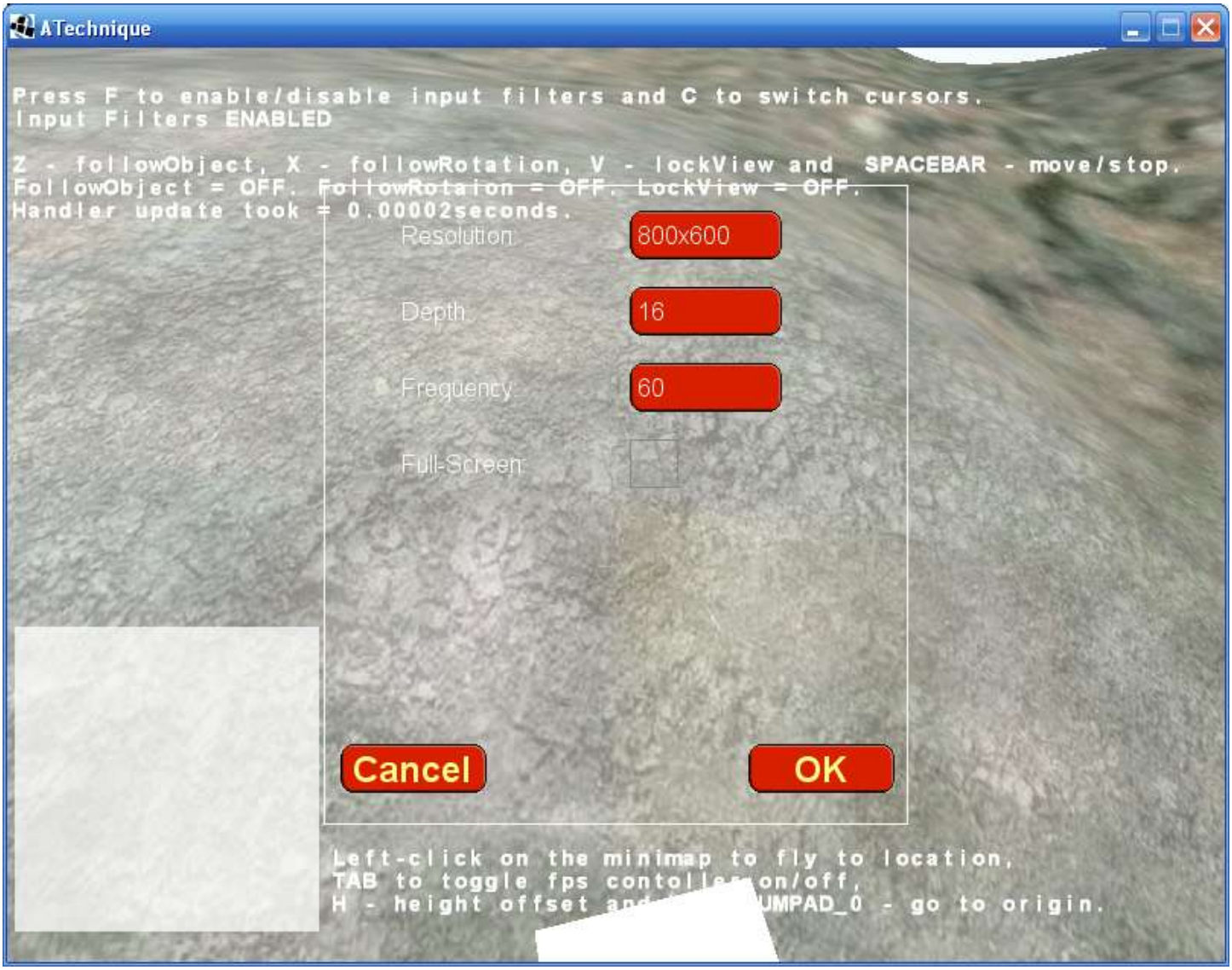
When the player hits the Escape key, I display this menu. I wanted to somehow make it obvious to the user that the game is paused and disabled, and I did this by setting the GBUI window to the size of the entire display and then applying an alpha state so it looks faded.

5.4.10 Edit Settings

Called from Main Menu:



Called from InGame:



The significance of this state (in the State Machine sense) is that it gets called from two other states, and returns to the state it was called from when closed. Because the background is transparent, we get a different appearance depending on where it is called from. Personally, I don't think I like that, but you might. Also, notice that when called from the InGame state, we haven't set the alpha blending on the rest of the display. Now we've confused the player. Oh well, we'll just put it on the "issues" list and move on. ☺

5.5 What's Next?

Here's what's on my list of next steps, for this tutorial and for my real project:

- Networking: you may have notices that even my limited example only covered the side of the player creating the server, but not the player connecting to a server. I want to introduce real networking and a "chat" application. This will also introduce the use of GBUI components at the same time as "regular" jME objects. I believe this is going to required at least some slight modification of my architecture, but time will tell.
- Try this: notice that the windows are centered on the display. Now use the Edit Settings screen to change the resolution. Notice the windows aren't centered any more. This is because when the screens/views are created, they use the display setting current at that time. Many views are sized to be proportional to the display resolution. Currently we have no mechanism to rebuild them when the resolution changes. I think the application would be a little more "well behaved" if it did this, but this is a relatively low priority for me since it is non-critical eye-candy and there is a workaround: restart the app.
- Fix the mouse cursor problem in the InGame state
- Put real meat on the application: load real campaigns with real scenarios and let players play real games

Good luck!

Appendix A: References

- Bang! Howdy – created by Three Rings Design
- BUI – the “original” BUI (Banana User Interface) library, used by Bang! Howdy
- StrategicHandler