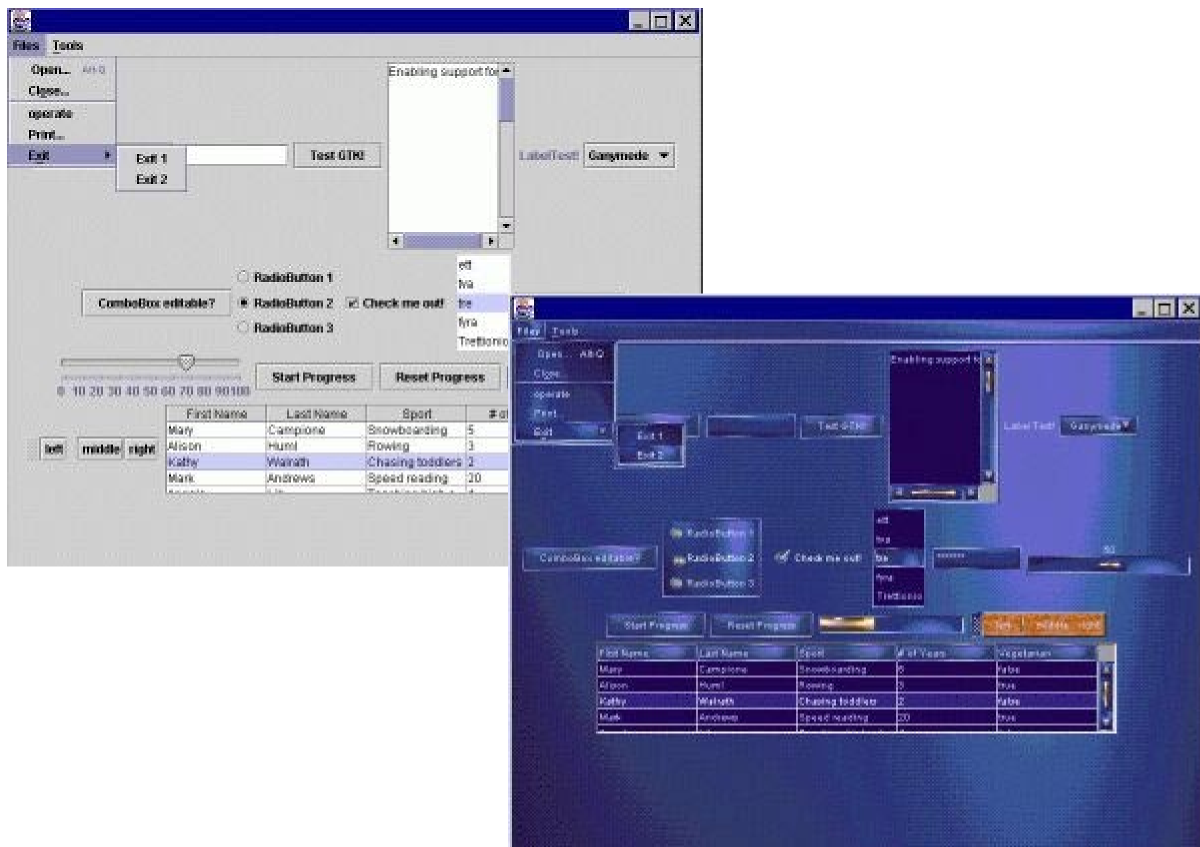
Friday, August 11, 2000

Creating a Swing

LookAndFeel for GTK themes



***A**uthor Fredrik Lagerblad*

Abstract

This Master's Thesis is written as a result of my degree project at Sun Microsystems Inc., CA, USA. The goal of the degree project was to create a Pluggable LookAndFeel package for Java's Swing component set that incorporates the existing GTK theme standard

To provide the ability to change the appearance, Look And Feel, of an application is becoming more important. Most new web-based applications give the user that option, but in Java, though possible, it has been very hard to do. By creating a new Swing LAF that incorporates the existing GTK theme standard with its hundreds of already-made themes, a new world is opened to Java users on all platforms. The Linux users especially benefit from this as their Java Swing applications now can seamlessly fit in with their existing desktop and other applications.

The GTK theme standard is based on a textfile and a number of image files. The textfile dictates how and where the images should be used, but does this in a very special format that was initially very hard to grasp. The images, which are of the PNG format, must all be stretched to fit the components. This stretching is quite complex and time-consuming as the images contain their border in the original image, and must therefore be removed and stretched separately.
To avoid having to "invent-the-wheel-again" the BasicLookAndFeel was inherited, which provided most of the common functionality of a theme. The GTK LAF package design was after a prototype had been built set to be in three layers and would use scale and paint on-the-fly architecture.
After testing and evaluating the package with a probing application, OptimizeIT, the performance could be improved significantly and the package is now fully comparable with the existing system Swing LAFs as Metal, Windows and Motif.
The next progression of the package could be to add other theme standards, perhaps the coming XML User interface Language, XUL.

Preface

This Master's thesis presents my Degree project that was done at Sun Microsystems Inc., Cupertino, CA, USA. It will conclude my Master of Science in Electronic Engineering at the Royal Institute of Engineering (KTH) in Stockholm, Sweden.
Sun Microsystems Inc. is a major hardware and software company based in Palo Alto, California, USA, but has offices all over the world. They are the inventors and copyright owners of the Java programming language. This project took place in Cupertino, CA, in conjunction with the Swing Team during the time period October 1999 to March 2000.

I'd like to thank Georges Saab, my advisor at Sun, Vlad Vlassov, my advisor at KTH, and the whole Swing Team for their indispensable help and input to the project.

[Abstract](#) 2

[Preface](#) 3

1. Introduction 6

1.1 Specification of the project 6
1.1.1 Explicit requirements for the finished package 6

1.2 Where the project was done 6

1.3 Structure of the thesis 7

2. Background 7

2.1 Motivation for the project 7

2.2 Java and the Swing package 8
2.2.1 The Java programming language 8
2.2.2 Swing 8
2.2.3 Swing's Pluggable Look And Feel (PLAF) architecture 12

2.3 Linux and the GTK themes 18
2.3.1 Linux 18
2.3.2 The GTK package and its themes 18

2.4 The combination of PLAF and GTK Themes 21

3. Design of the package's architecture 21

3.1 Initial and basic design goals 21

3.2 Building a prototype 22

3.3 Rethinking the design after code and prototype review 24

3.4 Actual design used 25

4. Implementation 26

4.1 System setup and tools used 26

4.2 Implementation of the design goals 27
4.2.1 The ImageData, StyleData and ThemeData classes 27
4.2.2 The parser 29
4.2.3 The image matching algorithm 30
4.2.3 The GTKMapper class 32
4.2.4 The GTKLookAndFeel convenience methods 34
4.2.5 The painting methods in the GTKUtils 36
4.2.6 Painting methods in the UI delegates 37
4.2.7 Other important classes and methods 39

4.3 Optimizations and improvements 40

4.4 Problems encountered 41

5. Testing and evaluation 42

5.1 What tests and evaluation used and why 42

5.2 Performance tests and results 42
5.2.1 Caching all images versus scale-and-paint-on-the-fly 42
5.2.2 ImageData caching in the GTKLookAndFeel class 44
5.2.3 Caching and pre-upsizing of tiled images 45
5.2.4 Optimization of the GTKParser.readFile() method 46

5.2.5 Delays for decoding the fonts in the parser 46

5.3 Example of usage **46**

5.4 Conclusion from the tests **48**

6. Conclusion and future work ***48***

References ***50***

Appendixes ***51***

I. Code **51**

com.sun.java.swing.plaf.gtk Class GTKParser 51
com.sun.java.swing.plaf.gtk Class ImageData 53
com.sun.java.swing.plaf.gtk Class StyleData 55
com.sun.java.swing.plaf.gtk Class ThemeData 57
com.sun.java.swing.plaf.gtk Class GTKMapper 58
com.sun.java.swing.plaf.gtk Class ImageDataDesc 59
com.sun.java.swing.plaf.gtk Class GTKLookAndFeel 60
com.sun.java.swing.plaf.gtk Class GTKUtils 63
com.sun.java.swing.plaf.gtk Class GTKIconFactory.CheckBoxMenuItemIcon 65
com.sun.java.swing.plaf.gtk Class GTKBorders.NewFocusBorder 66
com.sun.java.swing.plaf.gtk Class GTKButtonUI 67

II. Code for the test program **70**

III. Glossary **74**

IV. GTK Color Representation **75**

1. Introduction

1.1 Specification of the project

The ultimate goal of the project is to create a software package in the Java programming language that will enable and incorporate the existing Linux GTK standard for using themes. To use themes, or nowadays also sometimes referred to as skins, is a way to change the way a computer application looks and behaves. In Swing, an all-Java graphical component package to Java, the possibility to at runtime change the Look And Feel (LAF) has been made possible by its Model-View-Controller architecture. So far, the only different LAFs available to developers have been LAFs that copy the appearance and functionality of other operating systems as Windows, Macintosh and Unix and a special Java-only LAF called Metal. Developers have been able to develop their own custom LAFs, but it takes an experienced programmer, a lot of in-depth knowledge of Swing and quite some time. Other languages and component frameworks, e.g. the GNU Toolkit (GTK) on Linux, also have this theme changing possibility, where they have made it more easy for users to develop their own themes. This has sparked many users to create their own themes and exchange them among each other on the Internet (http://gtk.themes.org).
The additional package to Swing that will be created in this project will enable users and developers to take advantage of all the existing themes created for the GTK in their Java applications written using Swing and more easier create their own new ones.

1.1.1 Explicit requirements for the finished package

- **Performance and memory footprint** – the performance must not differ too much from the existing Swing LAFs. The memory footprint will probably be larger due to the many images used in the themes. It is important to find a good balance between speed performance and the memory usage.
- **Ease of use –** the package should be as easy to use as the existing LAFs. Just a few lines of code should be enough to make use of it.

1.2 Where the project was done

This project took place at Sun Microsystems Inc., Cupertino, California, USA. I worked with the Swing Team, the creators of the Swing package, at the Java language developing part of Sun. The project started October 11 1999 and continued until March 17 2000.
My advisor at Sun is Georges Saab, senior software engineer, who has been working in the Swing Team since the start, 1996.
My advisor at my school, Kungliga Tekniska Högskolan, is acting associate professor Vladimir Vlassov <vlad@it.kth.se> at the Information Technology Institution.
The examinator of the exjobb is Professor Seif Haradi <seif@it.kth.se>.

1.3 Structure of the thesis

The structure of this thesis is as follows:
This first section discussed what the goal of the project was, when and where it took place, and what people that were engaged in it.
Section 2 will provide the background to the project, explain how the Java Swing package and the Linux GTK themes function and how they can be used together.
Section 3 describes the design stage; discussion about the design of the architecture, building a prototype and eventually deciding for a final design.
Section 4 describes how the implementation of the design was done, and in more depth discusses the key classes and their function. It also describes the improvements and optimizations done.
Section 5 discusses the tests performed on the package, and what conclusions that can be drawn from them.
Section 6 concludes the findings of the thesis and discusses what the future might bring.
Last in the thesis are the references and appendixes.

2. Background

2.1 Motivation for the project

This project, i.e. this product, is needed primarily for two reasons, to offer the booming Linux platform and its users a more seamless integration with Java, and to offer all users of Java and Swing, on all platforms, an easy way to use and create themes for their applications.

Linux has in the last two years risen from being a computer enthusiasts' operating system to becoming a broadly accepted e-commerce platform and cheap home user operating system, by many seen as a real threat towards Microsoft Windows. Sun Microsystems also recognizes Linux as an important computer platform and tries to offer most of its product for it. With this integration of the existing and widely used GTK themes and Java, Linux users can now seamlessly use Java applications with other native applications. The way that the GTK themes are constructed enables them to be used not only for the Linux platform but also for virtually every platform with Java applications.

The ability to change how your application looks has within the last three years become almost a standard, at least for web centric applications. It started with the MP3 playing application Winamp, with gave its users the ability to design their own graphical interface. This became enormously popular and soon caught on with other developers and their applications. The Internet was the ideal place to exchange these themes or skins. Most users now expect themeing, which means that the developers in turn expect support for it in the language. So for Java to keep up with the trend and keep its developers, it must provide its developers this feature. This was partly done with Swing's Pluggable LookAndFeel architecture, but for most developers it was too hard and took too much time to create their own LookAndFeels.

The reason that this project was chosen for me to do and not one of the Swing core members was because the Swing Team during this time was very busy with the release of Java 2 version 1.3, a very important bug and performance update.

2.2 Java and the Swing package

2.2.1 The Java programming language

Java is an object oriented programming language that was created by under the supervision of James Gosling and Bill Joy during 1993 and 1994 by Sun Microsystems Inc. and released in May 1995. It immediately created a buzz within the computer world. It was one of the first language that was completely platform independent and its strong network support made it perfectly suited for the then evolving Internet. People could now add interactivity to their static web pages through the use of applets (an Internet browser embedded application which Java provided). Soon the initial hype about the applets (and Java) settled down, and Java started to mature into a "real" programming language that was highly suitable for creating distributed applications. As Java also is relatively easy to learn and use, but still very versatile, and is a schoolbook example of an object oriented programming language it became very popular at universities and other computer teaching institutions.
With the release of Java 2 Enterprise Edition (J2EE) in 1999 Java more or less has become a de facto standard for server side enterprise programming. Backed by most of the industry J2EE has in a short time revolutionized the formerly complex area of business-critical server-side programming by hiding complicated issues as transactions, persistence and security, letting the developer focus only on the business logic.
As Java became more accepted as not only a Internet programming language but also as a serious language for developing commercial applications more demands were put on its Graphical User Interface (GUI) building abilities. From the start the way to build GUIs was by using the Abstract Window Toolkit (AWT), but soon issues started to arise which could not be solved with AWT. (These issues will be discussed in the next section) To answer the demand of the developers and to strengthen Java's platform Sun created a new group with the some of the people that used to work with AWT, their task was to create a new set of pure-Java GUI components for Java or as their mission statement says it:

"*To build a set of extensible GUI components to enable developers to more rapidly develop powerful Java front ends for commercial applications.*" [Ref 1]

The team members called this project 'Project Swing' (by the way named by Georges Saab, my advisor); a name that soon caught on and later was set as the official name.

2.2.2 Swing

Background

As stated above Swing is a pure Java GUI component package that was released as a part of the Java Foundation Classes (JFC) with the Java Development Kit (JDK) 1.1 in the spring of 1997. The JFC incorporated many features from Netscape's Internet Foundation Classes and some

design aspects from IBM's Taligent division and Lighthouse Design[Ref 2]. The JFC consists of five APIs: AWT, Java 2D, Accessibility, Drag and Drop and Swing.
The Swing project was initiated to resolve some of the issues that had been discovered with the AWT package. The AWT relies on peer components; i.e. an AWT button creates a Windows button on a Windows operating system and a Mac button on a MacOS system. This means the AWT components can behave and look different on different platforms. This means that it can be very hard to design the component, a list component can behave differently on different platforms. To manage and hide these differences can be very difficult and cumbersome. Also for the application developer the visual design can be difficult when the components slightly differ on different platforms. The components that could be provided also had to be "the-least-common-denominator" of the platforms supported. To overcome these shortcomings Swing was developed as a complement, as not necessarily as substitute, for AWT. The Swing Team set some design goals that Swing would: [Ref 1]

1. *Be implemented entirely in Java to promote cross-platform consistency and easier maintenance.*
2. *Provide a single API capable of supporting multiple look-and-feels so that developers and end-users would not be locked into a single look-and-feel.*
3. *Enable the power of model-driven programming without requiring it in the highest-level API.*
4. *Adhere to JavaBeans™ design principles to ensure that components behave well in IDEs and builder tools.*
5. *Provide compatibility with AWT APIs where there is overlapping, to leverage the AWT knowledge base and ease porting.*

The most important features that Swing provided were:

- Highly configurable lightweight components.
- Pluggable LookAndFeels.
- A variety of new components such as tables, trees, sliders, progress bars, tooltips.
- Support for JavaBeans.
- Support for Drag and Drop.
- Advanced text handling.
- Support for accessibility through the JFC Accessibility package.

Sun recommends the use of Swing components for desktop and web applications but still will support the peer AWT components, much because Java is not only intended for use on desktop systems. In devices as for example telephones using Java technology, which might not have a big screen to show graphical components on, AWT's peer methodology can be used to map an AWT button to a telephone button.

Model-View-Controller architecture
Swing is loosely modeled after an architectural model known as Model-View-Controller (MVC). The MVC model was invented at Xerox PARC in the 1970s and divides each component into three elements: the model, the view and the controller. Each of the elements has its own specific task to in the component's functionality: (also showed in Figure 2.1)

Model – the model holds and controls the state data of the component and manages transformations on that state. A button model for example holds information about whether the button is enabled, pressed, icons, text on the button etc. The model differs for different components and is always independent of how the component is visualized.

View – the view is the element responsible for showing the component on the screen or other devices, e.g. audio output, Braille display. The view queries the model about which state it is in and then draws (or acts) it according to it.

Controller – the controller handles how the user's input should be handled. Events as mouse clicks, keyboard input and focus gained/lost are examples on inputs that the controller decides what actions to take upon. The controller decides how each component will react to the event.

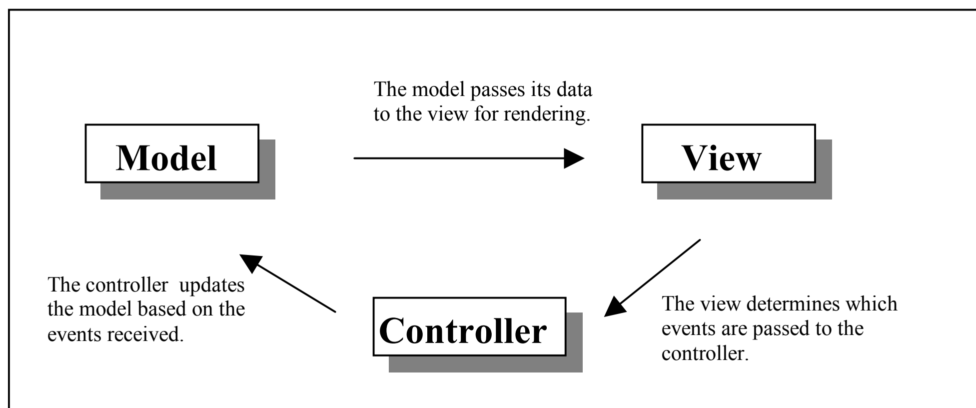


Figure 2-1. The communication between the elements in the MVC architecture.

The MVC architecture has several advantages, the most important ones being:

- Multiple views for one model – a single model can have any number of views connected to it, e.g. a table and a chart or both audio and screen output. If an update is made to the model notifies all its views and lets them update themselves.
- Easy change of the component's appearance – by simply changing the view connected with a certain model the component can change its whole appearance at runtime without affecting the underlying model.

Swing's architectural model

The Swing Team started out using the MVC model, but soon discovered that this model wasn't the best solution. The split between the view and the controller didn't work well because they needed a tight coupling. [Ref 1] "for example, it was very difficult to write a generic controller that didn't know specifics about the view". The Swing Team solution was to collapse the view and the controller into one UI object, known as the UI delegate. This model is referred to as *separable model architecture*, see Figure 2-2.

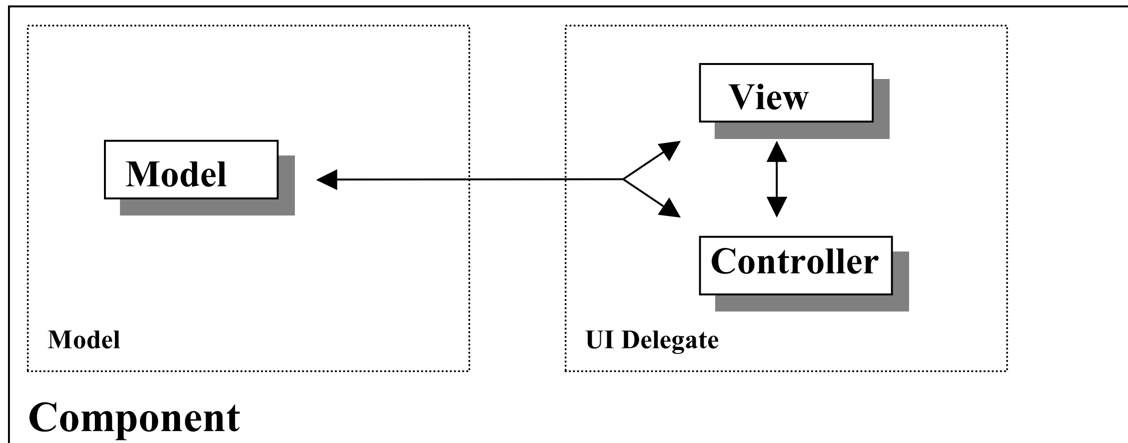


Figure 2-2. Swing's separable model architecture.

The *model* and the *UI delegate* main these tasks in the *separable model* are these:

Model
- Query internal state
- Manipulate internal state
- Add and remove event listeners
- Fire events

UI delegate
- Paint
- Return geometric information
- Handle AWT events, e.g. forward button clicks to the model

For a Swing component the model can be changed, and you can create your own one, but there is always a default model provided with each component, e.g. JButton has the model DefaultButtonModel by default. Each model must implement a for the component specific interface, i.e. "promise" that it will have and support certain methods. To set a new model for the component one simply uses the component's `setModel()` method to start using it. Table 2-1 below shows some examples of Swing components and their models. Note that different components can share the same model interface.

Table 2-1. Some Swing components and their corresponding models.

Component	Model Interface
`Jbutton`	`ButtonModel`
`JtoggleButton`	`ButtonModel`
`Jmenu`	`ButtonModel`
`JcomboBox`	`ComboBoxModel`
`JprogressBar`	`BoundedRangeModel`
`JScrollBar`	`BoundedRangeModel`
`Jtable`	`TableModel`
`Jtree`	`TreeModel`
`JtextArea`	`Document`

In the same way a new UI delegate can be created and used. This ability is what enables Swing to have its Pluggable LookAndFeel architecture, which will be described in detail in the next section. Every Swing component has a corresponding UI delegate which must implement a component specific interface. The name of the corresponding interface is got if the 'J' in the start of the component is removed and 'UI' is added at the back. Thus the UI delegate for JButton is ButtonUI and for JScrollBar it is ScrollBarUI. All the different UI delegates for the components extend ComponentUI, the super class of UI delegates. It defines the basic (view) methods for rendering the component (`paint(), update()`) and defining its geometrical size (`getPreferredSize(), getMinimumSize()`). The controller methods are determined by specific subinterfaces, e.g. `ComboPopup`. What these methods actually are supposed to do is described more in-depth in the next section about the PLAF architecture.

Model – View interaction
When the model wants to notify its view(s) that its data or values have changed it does this through events. Swing models use the *JavaBeans Event Model* to do this. Two different ways to do this is used in Swing:

- Lightweight notification – the model sends out an event (ChangeEvent) simply saying that its state has changed to all the interested parties. It is then their responsibility to do a callback and find out *what* has changed. The primary advantage with this technique is that a single event instance can be used for all the notifications, which is very desirable when the changes occur often, e.g. when a scrollbar is dragged.
- Stateful notifications – the model sends out a new instance of the event describing exactly *how* it has changed to each interested party. More information about the change can the stored in the event, this is often desirable when a changed has occurred in a more complex component where it can be hard for the receiving part distinguish exactly what has changed. For example when a column of cells in a table change value.

The model has no knowledge about which view(s) that is displaying its data, following the MVC architecture. The model only knows which listeners that is interested in knowing about its state changes, these can be UI delegates or an application, and simply notifies these when something has changed. It is the Swing component, i.e. the UI delegate, which is responsible for hooking up the appropriate listeners with the model so that it will repaint itself whenever its state changes.

2.2.3 Swing's Pluggable Look And Feel (PLAF) architecture

Swing's separable model architecture provides the ability to change the look and feel of an application at runtime, and to create your own one. This is referred to as the Pluggable LookAndFeel (PLAF) architecture. The developers designed Swing so that if you don't want to use or create your own LookAndFeel (LAF) it is more or less from hidden to you. On the other hand if you *do* want to create a new LAF or modify it for a component or even a whole new LAF, they have built the PLAF architecture so that you do that without too much trouble. Just modifying or creating a LAF for a component is quite straightforward. Creating a whole new LAF for the whole component set takes more in-depth knowledge and not least a considerable amount of time. The hooks are provided and there are LAF super classes, the Basic LAF, from which you can inherit the basic functions, but it requires a sound understanding of the whole architecture to get it right. The documentation of how to create your own LAF is maybe one of

the biggest shortcomings of Swing today. But that is partly why this project was initiated, to provide the users and developers with a simpler way to change the appearance of their applications.

Creating a new LookAndFeel

What you basically do when you create a new LAF is to create new UI delegates for the components and replace the default ones with these. It is not necessary to replace all the UI delegates of the default LAF. Many methods and behaviors are alike for different LAFs, that is why Swing has a package of abstract super classes for LAFs, the `BasicLookAndFeel`, where these are collected. It is this one you extend to create your own LAF (it is not absolutely necessary to do it but it helps a lot).

Key classes in the PLAF architecture

`LookAndFeel` – this is the base class of a LAF. It provides the information on what UI delegates to use for the components, what colors, fonts etc to use and also a name and identifier of the LAF. Custom LAF extends the abstract `BasicLookAndFeel` class to replace the default properties and define new ones. The properties are stored in a hashtable, the UIDefaults that is described below. The `LookAndFeel` class also provides static convenience methods for simplifying common tasks as installing new borders, colors etc:

```
installBorder( )
installColors( )
```

These methods will be described more in the section about installing and uninstalling UI delegates.

`UIDefaults` – this class consists of a hashtable that contains all the above-mentioned properties, the UI delegate table and some helper methods to access and replace these. Since every entry in a `java.util.Hashtable` is a `java.lang.Object,` this is also what you get when do a `get(String key)` on the hashtable. What these *helper methods* do is that they cast the object from the hashtable to another class, the class that is expected. For example: `public Color getColor("Button.focusColor")` tries to cast the returned object from the hashtable into a `java.awt.Color` object and returns it.

The *UI delegate* table consists of entries like these:

```
"ButtonUI",    "javax.swing.plaf.basic.BasicButtonUI",
"ScrollBarUI, "javax.swing.plaf.basic.BasicScrollBarUI"
```

Using methods like,

```
public Class getUIClass(String uiClassId)
```

(Where uiClassId might be “ButtonUI” for example)

the correct UI delegate class can be retrieved for each component. It is these entries you replace to force the component to use your own custom UI delegates. E.g.:

```
"ButtonUI",    "com.myCompany.plaf.MyOwnButtonUI",
"ScrollBarUI, "com.myCompany.plaf.MyOwnScrollBarUI"
```

The properties for the LAF's colors, fonts, borders, icons etc are stored in a similar fashion:

```
"Button.foreground", "new Color(Color.red)",
"Button.background,  "new Color(Color.blue)"
"Button.font",       "new Font("Times", Font.PLAIN, 12)",
"Button.border",     "new MyButtonBorder()"
```

A developer of a new LAF can replace existing entries or create own unique ones that can be then be retrieved from the UI delegates' code.

Since the UIDefaults has an ordinary hashtable the standard commands for retrieving and inserting entries can be used:

```
public Object put(Object key, Object value)
public Object get(Object key)
```

The information in the UIDefaults hashtable can be accessed straight from the `UIDefaults` class, but the proper way to access it is through the UIManager class.

UIManager – this class provides a simple interface to a variety of information about the current LAF and for installing new ones. It is an all-static class, so all its methods are static and you never have to instantiate it. Perhaps the its most important method is the one used for setting a new LAF:

```
public void setLookAndFeel(LookAndFeel newLaf)
```

This sets the new LAF as the current one. It does not automatically tell all the components to update themselves to use it, but this can easily be done with another of the UIManager's helper methods:

```
public static void updateComponentTree()
```

The UIManager not only handles the current LAF, but also keeps track of a few other ones as well:

Current LAF – the currently installed and used LAF.

Cross-Platform LAF – a LAF that is not modeled after an existing native platform. By default, this is Swing's own Metal LAF.

System LAF – this is the LAF that emulates the current platform. On Windows it is the Windows LAF, on Unix/Linux it is the Motif/CDE LAF.

Installed LAFs – a set of all currently installed LAFs available to an application. By default, the Metal, Windows and Motif LAFs.

Auxiliary LAF – a set of LAFs that provide accessible support for an application, e.g. an audio LAF. By default this set is empty.

All these LAFs and set of LAFs can be retrieved and set by methods that the UIManager provides.

As mentioned before it is through the UIManager the UIDefaults properties are accessed, and therefore provides the same helper methods for retrieving objects, e.g.:
`public static Color getColor(Object key)`

What these methods do is to simply obtain the current UIDefaults and invoke the same method on it.

Another one of the UIManager tasks is to keep track of which properties in the UIDefaults table are actually set the user, the current LAF or if they are system defaults. This is important because if an user sets a specific property, e.g. the font in JTextFields, using one LAF, he also expect that setting to remain even if he changes to a new LAF. This is done my storing all user-set properties in a special UIDefaults table called the *User Defaults* table. This is always checked first, then the current LAF's table is checked and finally the *System Defaults* table. There is another implication to this, when a LAF changes how do the new UI delegate know that a property, say a border for a JButton, it is not actually a user-set border and it can not install its new border instead? This is solved by tagging all the LAF property objects with the tag *UIResource*, which is just an empty interface. So a Color object instead is stored as a ColorUIResource object, where the ColorUIResource is simply a class defined like this:

```
public class ColorUIResource extends java.awt.Color
                             implements UIResource { }
```

These ready-tagged UIResource classes exist for the most common property objects like ColorUIResource, FontUIResource, BorderUIResource, InsetsUIResource and DimensionUIResource.

By tagging the properties like this it is easy to check before setting a new property if the current one is user-set or set by the current LAF. The check is to simply see if the property is a UIResource object by using the `instanceof` check:

```
if( button.getBorder == null ||
    button.getBorder() instanceof UIResource  ) {

   button.setBorder( newBorder );
}
```

Installation of a UI delegate
It is important to understand how the architecture manages the installation of new UI delegates, after that it is easier to understand the different objects' and classes' roles and responsibilities. Below is a flow chart, Figure 2-2, of a JButton being installed with the Metal LAF (not every method call is showed).

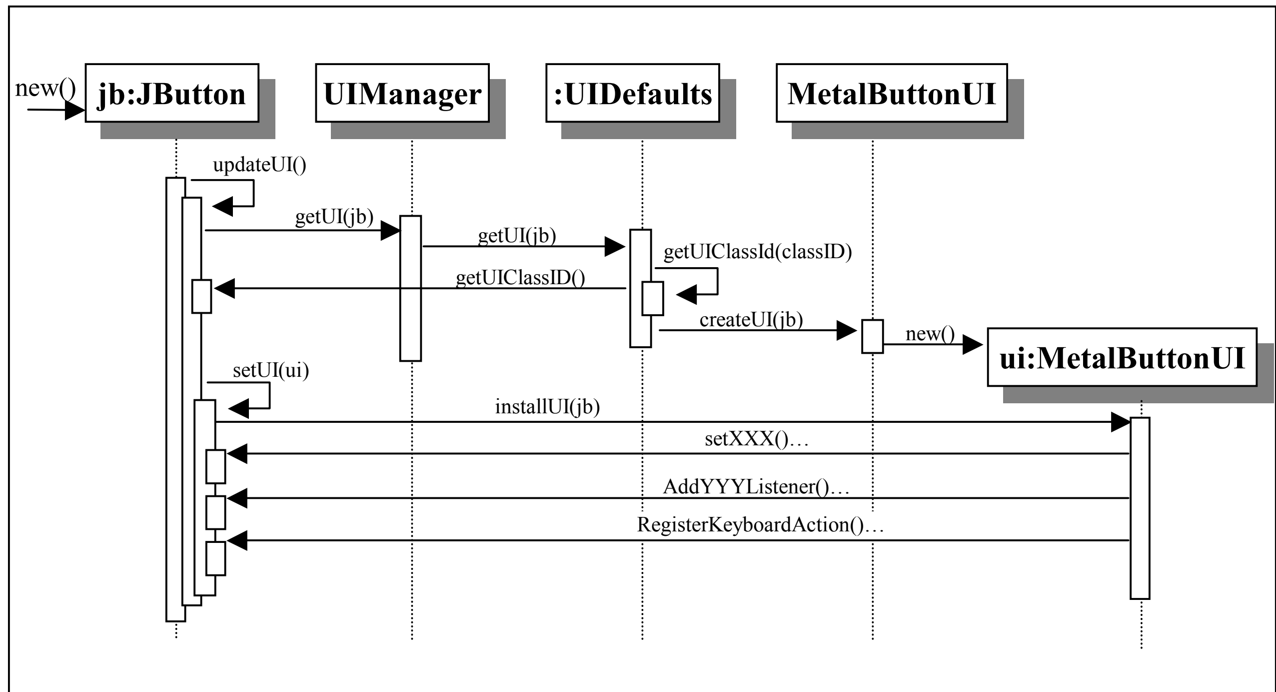


Figure 2-2. The installation of a UI delegate.

- The component's constructor method calls the `updateUI()`, which is a method that every component has. It basically sets a new UI delegate like this:

```
public void updateUI() {
    setUI( (ButtonUI) UIManager.getUI(this));
}
```

- The `getUI()` method of the `UIManager` in turn queries the current `UIDefaults` for the appropriate UI delegate.
- The `UIDefaults` first look up the UI Class ID for the component, e.g. "ButtonUI" for a `JButton`. Then it retrieves the correct UI delegate class for that UI class ID, in our case "MetalButtonUI".
- It uses the UI delegate's method `createUI()` for provide an instance of the class for the component. (This can be a new, unique instance or an instance that is being reused)
- This instance is returned to the component and the `updateUI()` method calls `setUI()`. The `setUI()` method asks the UI delegate to install him by invoking its `installUI()` method.
- The UI delegate in its `installUI()` method calls all the necessary installation methods, e.g.:

```
installDefaults()
installListeners()
installKeyboardActions()
```

After the installation is through the component can now start using the UI delegate for its painting, size geometry etc.

Customization of a UI delegate
When a new UI delegate for a component is to be created, the easiest way is as mentioned above to extend the `BasicLookAndFeel`. It provides the basic functionality of the for all the components' UI delegates. Then one simply overrides the methods that need to be changed with your own code. For example the `paint()` method, which handles the rendering of the component:

```
public void paint(Jcomponent c, Graphics g) {

   //Own code for the rendering goes here.
}
```

Quite often it is not desirable to completely override a super class's method; just some extra functionality is wanted. This occurs often in the UI delegates' installation and uninstallation methods `installUI(), uninstallUI(),installDefaults(), uninstallDefaults(), installListeners(), uninstallListeners()` etc. This is achieved by first calling the method's super method and after that adding your own code:

```
public void installDefaults(Jcomponent c) {
   super.installDefaults(c);

   //Own code goes here.
}
```

Stateful or stateless UI delegate?
One important aspect to consider is if the UI delegate should create a *new stateful instance* for each component or provide a *single static stateless instance* that all components of the same class share. The Swing Team discovered during the development of Swing that much performance and memory can be gained by letting the components from the same class share a UI delegate instance. This is not true for all components though; some more complex ones like JTree and JTable do not gain from sharing UI delegates.

If a static stateless shared instance is provided for a class it has to each time it is to be repainted query the model for all information it needs, it can't store any information locally. One might think that it is needed anyway due to the MVC architecture, where the view should query the model before rendering, but sometimes it is not needed. For example, an image has been scaled or manipulated in a time-consuming operation to fit the component's size and the component rarely changes its size, caching can improve the performance significantly. Or if a listener has been added to the model, the model updates the UI delegate automatically through an event. If some caching is still needed in a static stateless UI delegate, there is a way to do that too. All JComponents provide the methods:

```
public void putClientProperty(Object key, Object value)
public Object getClientProperty(Object key)
```

These methods provide access to an internal hashtable of the component. By using these methods one can cache information that is needed repeatedly. It is important though to remember to nullify the entries when uninstalling the UI delegate.

For the stateful unique UI delegates this is not a problem. Since there is an instance of the UI delegate for each instance of the component, one can cache as much as needed within the UI delegate object.

2.3 Linux and the GTK themes

One can hardly have missed the ongoing hype about Linux. The whole computer industry is trying to be in some way or other involved in Linux (or at least look like they are). During the last two years Linux has come up as the prime contestant to threaten Microsoft's monopoly on the client side, and as a cheap but very versatile platform for server side systems.

2.3.1 Linux

Linux is a free UNIX clone that was initially created by Linus Torvalds, a student at the University of Helsinki, Finland. He began working in 1991 on his own version of UNIX for the Intel x86 platform and released version 1.0 of the kernel (the most important core of the operating system) in 1994. Since then the kernel, and other parts, of the Linux operating system has been in continuos development by Linus Torvalds and large number of independent developers all over the world. Linux is protected under the Gnu Public License (GPL), which means its source code is freely available to everyone. Companies can still charge money for their distributions, a special version of the operating system developed by the company, as long as the source code remains available. Today some of the most popular distributions are Red Hat, Debian and Corel. These distributions have largely helped to ease the use of Linux. Linux, just like UNIX, have always been considered very powerful and highly configurable but also very hard to learn. Everything has been more or less configured by editing various text files or entering command-line commands. With these new easier-to-use distributions a lot of work have been put in to simplify the installation, configuration and running of the system. By providing graphical user interfaces (GUIs) instead of the command-line interface, it is today a much more user-friendly system. A very important part of the GUI is the Window Manager.

Window Managers

A Window Manager is responsible for handling the visual interface of the system. It displays more or less everything you see, the desktop, the windows that applications run in and all the control windows. It visually controls the top-level windows, not the content of them as the GTK package do. (Described below.) Several different Window Managers are available today, the most common and used ones being Enlightenment, KDE, AfterStep and fvwm.

2.3.2 The GTK package and its themes

The Gimp Toolkit (GTK) is a set of GUI components for the Linux platform. It originates from the GNU Image Manipulation Project (GIMP), which was an effort from the Linux scene to create a powerful image manipulation program, a PhotoShop clone. [Ref 3, 4] For the development of the GIMP a new set of object-oriented and robust of GUI components were needed, and the GTK package was created. It has since also been used been used in several other large Linux software projects [Ref 5], e.g. the GNU Network Object Model Environment (GNOME), and a large number of applications based on it exists today (see http://www.gtk.org).

The GTK is built on top of the GNU Drawing Kit (GDK), which is basically a wrapper around the low-level functions for accessing the underlying windowing functions of the X windows system.
The GTK provides the ability to provide your own rendering engine for the components, also known as a theme. With a custom rendering engine one can make the components look just like you want them to. To create a rendering engine one needs to implement the code for the drawing of every component. Several different ones have been developed; many emulate other existing platforms such as Windows, Mac, BeOS or NextStep. It is a complex and time-consuming task to create your own engine, so a Linux developer called Rasterman developed an engine that simplified the construction of new themes. [Ref 6] His engine, called the *pixmap engine*, uses images for the rendering of the components. A text file, the *gtkrc-file*, specifies in a lot of statements what images to use for each component. The images that are provided with the theme are in the Portable Network Graphic (PNG) format, a non-royalty format developed as a substitute for the GIF image standard. The pixmap engine grew very popular because people could now quite easy create their own themes and exchange them with each other. The main place for exchanging these themes on the Internet is http://gtk.themes.org. There is also a third kind of themes, *plain themes*, which only modify the colors of the components. They also have a *gtkrc-file*, but in it there is only color assignments to different components.

The structure of the gtkrc files

The excerpt below in Figure 2-3 is from a gtkrc file and shows have a menubar should be visualized.

```
style "menubar"
{
  font = "-*-verdana-medium-r-normal-*-11-*-*-*-p-*-iso8859-1"
  fg[NORMAL]       = "#00000f"
  fg[PRELIGHT]     = "#000000"
  fg[ACTIVE]       = "#000000"
  fg[SELECTED]     = "#000000"
  fg[INSENSITIVE]  = "#a8a8a8"
  bg[NORMAL]       = "#d8d8d8"
  bg[PRELIGHT]     = "#d8d8d8"

  engine "pixmap" {
    image
      {
          function          = BOX
          recolorable       = TRUE
          file              = "menubar.png"
          border            = { 2, 2, 2, 2 }
          stretch           = TRUE
      }
  }
}

class "GtkMenuBar" style "menubar"
```

Figure 2-3. Excerpt from gtkrc file showing the menubar specification.

The gtkrc file consists of style definitions and class-to-style mappings. Every style definition consists of one or more image definitions (only one in the example above). After a style has been defined it is mapped onto the component(s) that should use it. The *style hierarchy* is built so that

when a component needs to be rendered it checks its style first for the property/state it is looking for. If not found, it checks the *default style* which always exists.

Styles

A style dictates what images and colors to use for a component. What images to use are specified in the image definitions, described more below. A style has an image definition for each state that it wants the component to visually differ for. For example, a style for a button usually has image definitions for the states: normal, pressed, rollover and disabled.
Colors and fonts can be defined in for each style. If not present in a particular style, again the color or the font in the default style is used. The font is used for all text in the component. The defined colors are used for various things as text colors and backgrounds. For a complete listing of what all the color represents see Appendix IV.
A style can also inherit another style's definition and then add its own ones. This is done by adding an equal sign '=' after the style's name and then the name of the super style. (Compare with Figure 2-4)

```
style togglebutton = style button
{
    :  Definition of the style
}
```

Figure 2-4. The style togglebutton inherites the style button.

Image definitions

As stated above each image definition represents a state that a component can be in. It describes what image to use, if they can be stretched, how the border is defined, if it also has an overlay image etc. The different tags within the image definition are described in table 2-2 below.

Table 2-2. The image definition tags and their meanings.

Tag	Example	Meaning
function	= FLAT_BOX	State identifier
recolorable	= TRUE	Recoloring allowed?
state	= INSENSITIVE	State identifier
detail	= "entry_bg"	Component identifier
file	= "entry2.png"	Image file to use
stretch	= TRUE	Stretching of image allowed?
border	= { 3, 3, 3, 3 }	Size of the image's border
overlay_file	= "entry_overlay.png"	Overlay image file to use
overlay_stretch	= TRUE	Stretching of overlay image allowed?
overlay_border	= { 2, 2, 2, 2 }	Size of the overlay image's border
orientation	= HORIZONTAL	Orientation identifier

To identify what image that should be used at a certain state, an image matching method is used (described more in detail in section 4.2.3).

State transition
How does a component know which image to use, and when? A simple example will show how a button will change its image during a state transition.

1. A user clicks the mouse on top of the button, which is in a normal state. The windowing system reports this to the application, which forwards it to the button.
2. The button's controller translates the click into a button-specific action, in this case a press on the button, and reports it to the model.
3. The model changes its state to 'pressed', and forces its view to update itself.
4. The view's update method checks the model's state and size, requests the correct image for them and renders it.

2.4 The combination of PLAF and GTK Themes

Because the Linux GTK pixmap themes only consist of a simple text file and a number of images, they can be read and used on other platforms as well. So by creating a new Java Swing LAF that can read and interpret the themes, they become available to every platform for which there exists a Java Virtual Machine. How that LAF should be design and implemented will now discussed in the next section.

3. Design of the package's architecture

3.1 Initial and basic design goals

The project's initial definition was:
"To create a Swing LookAndFeel that lets the users incorporate and use themes from the existing GTK theme standard in their Swing applications."

The first issue was to decide which of the three different themes (plain, pixmap, and engine) were to be supported. Quite soon, it became clear that the engine themes, because they are entirely written in native C-code, could not easily be incorporated into the Swing architecture. So the first design goal set was to support both the plain and the pixmap themes.

Since the essential part of a pixmap theme is the gtkrc text file, which holds most of the information, it needs to be parsed and its information stored properly for easy and fast access. This information is then accessed by a set of custom UI delegates, a new LAF, which knows how to utilize it properly. Therefore, the main parts of the implementation would be to create the parser and the UI delegates for the components.
The large number of images in the themes, usually around 60-70 in a theme, of which most needed to be stretched to fit each component's size, meant some sort of image caching was probably needed for performance. So the second design goal set was to add image-caching functionality to the LAF.
Robustness and performance are of course important design goals. To take advantage of the big performance improvement, especially in Swing, of version 1.3 of Java, the package was to be designed to compatible with that version.
Another issue was to try to layer the design, where the different layers would be responsible for different tasks and provide services. Each layer uses the underlying layer's services and provide services to the one above. A layered design has several advantages, one of them is that a layer can be re-implemented, to improve the performance for example, and replaced without having to

change the other layers. It also makes the implementation easier, by being able to focus on one layer at the time, in the lines along the divide-and-conquer technique.

3.2 Building a prototype

It was decided with the advisor at Sun, Georges Saab, that a prototype version of the system should be created and then evaluated. The prototype would include a parser and a few UI delegates for a component, along with the architecture for delegating the information from the parser to the UI delegate. The first component that would be included in the prototype was decided to be JScrollBar. JScrollBar was chosen because it was neither too simple nor too complex. Other components were JButton and JTextField. The prototype also included a general study of how the GTK components behaved on Linux when it comes to things like focus, rollover effects etc. Figure 3-1 shows the basic functionality of the prototype system in a sequence diagram.

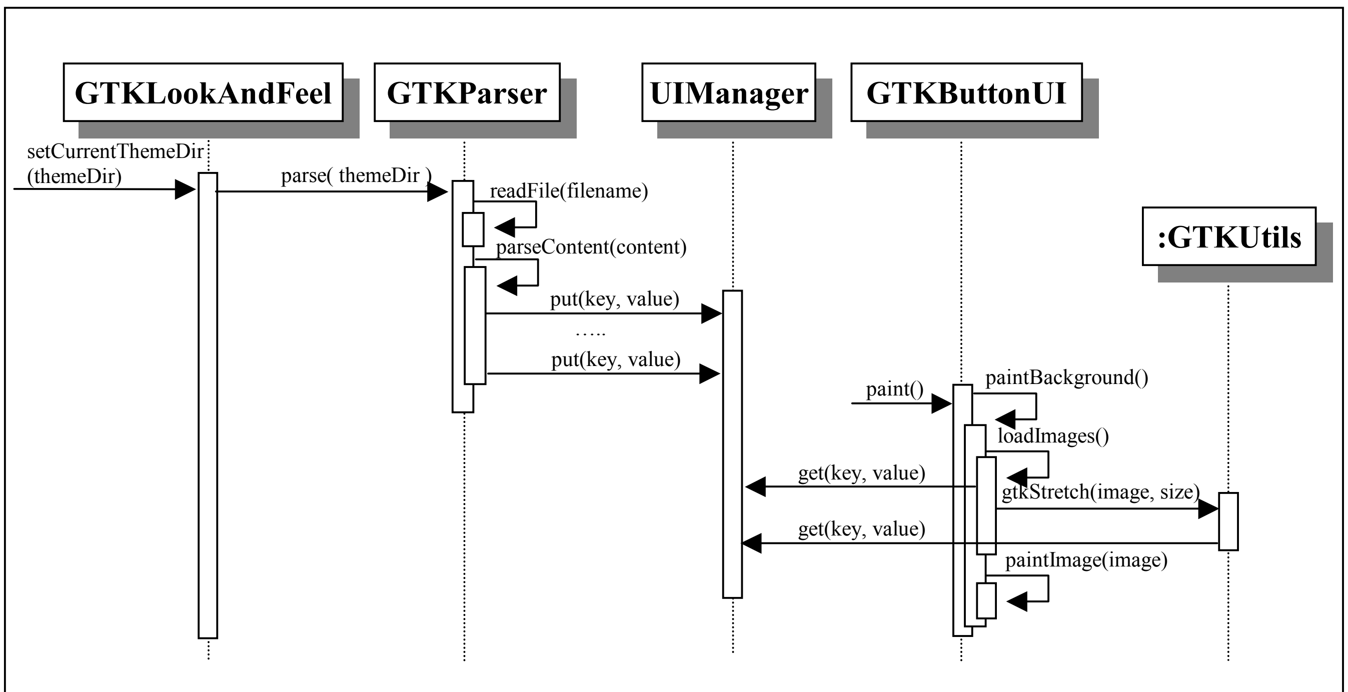


Figure 3-1. Basic functionality of the prototype.

The parser

At this stage in the project, I had not yet found the source code for the Linux version of the parser, to see how it was designed. Despite the fact that Linux is famous for its free and available source code, it was surprisingly hard to find. Not until after the prototype was finished the source code was found. It led to that I had to try to create my own system of identifying the different images corresponding to the different components. This system soon became very complex and irregular since the gtkrc files are not logically structured. It soon became a big problem as all the different image definitions had to be uniquely identified. It was good enough for the prototype, but it would have to be redesigned later.

As the gtkrc files maps GTK styles onto GTK components, not Swing components, some mapping from the GTK and the Swing components is needed. A mapping table was used for this, an example of the table can be seen in Table 4.1.
Since the GTK and the Swing component sets do not exactly match, not all the components can be supported. A new custom Swing component would not be supported either, if not constructed by a combination of existing components.

Image scaling
When looking into how the images needed to be scaled, the initial impression was that it was a quite time-consuming operation. This because of the way that they are supposed to stretched while still keeping their border intact as explained in Figure 3-2. The reason for the detailed description on how the images are scaled is to show how complex it is and that it became a very

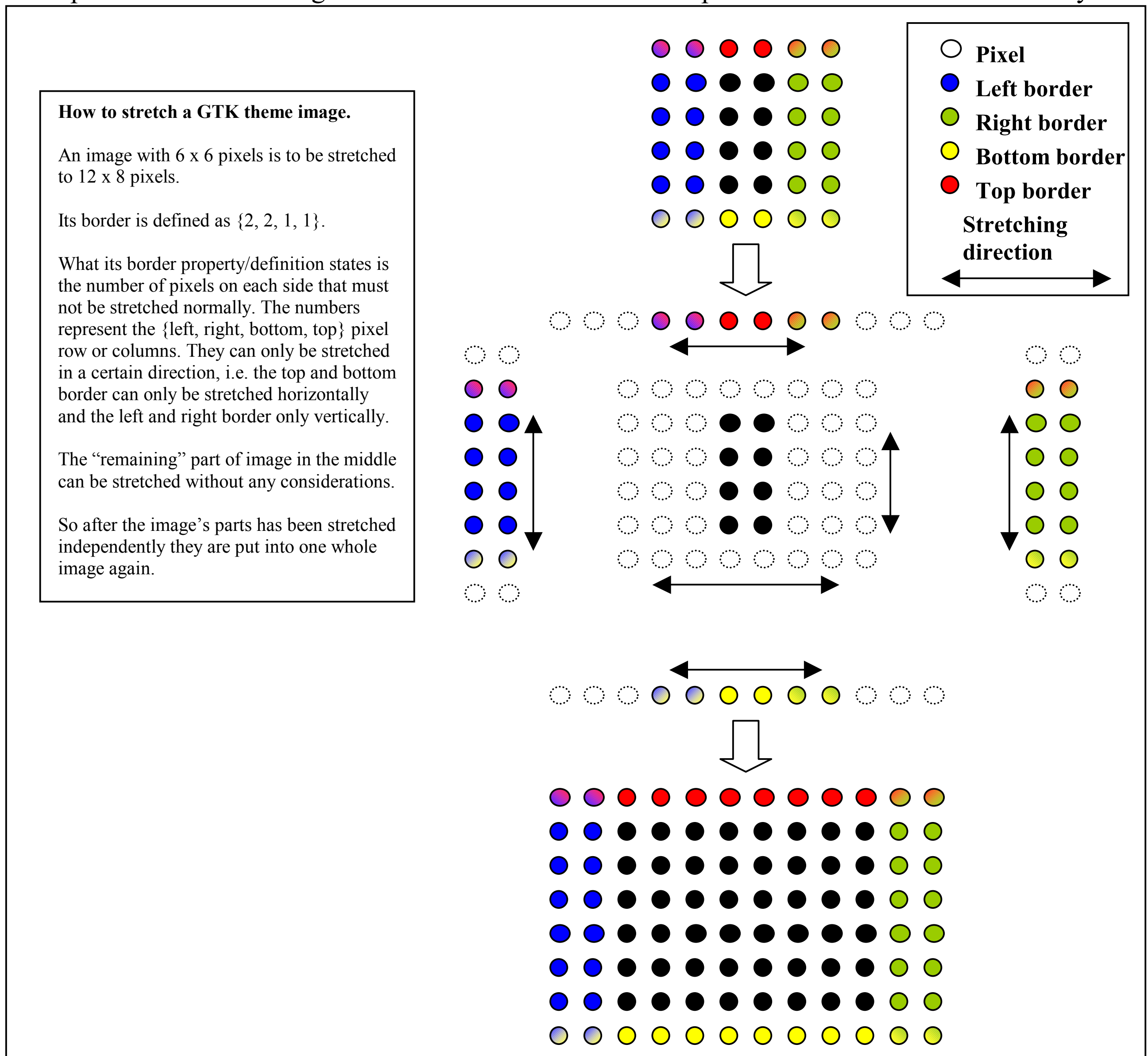


Figure 3-2. How a GTK theme image is scaled.

central part of the project, influencing key decisions.

So caching the images for each instance of the component seemed efficient. It would consume more processor-time at the component's instantiation time, but improve the runtime

performance. When each component, or rather its UI delegate, was created, all the images for its different states was stretched to the component's current size and cached locally in the UI delegate object. Only when the component was resized an update of its cached images was necessary.

Rendering when the component has focus
[**Definition of focus:** A component has focus if it was the last component used. The focus can also the transferred around the component set by using the TAB key. The component in focus can also usually be activated by pressing the ENTER key, e.g. a button would be pressed]
By studying the original GTK components I noticed that they handle focus by painting a special focus image around the component in focus and also decreased its size by 2 pixels on each side. The focus image was usually shared by all components and situated in the default style. If no focus image existed in a theme, no image was drawn and the only visual change that occurred was the decrease of the component's size. These slightly smaller images of the component would also have to be cached.

Not supporting plain themes
To provide the functionality for using plain themes, i.e. themes with only color changes, code for rendering the components without images had to be added. This could be done by taking existing code from another LAF, e.g. the Windows LAF. An initial test whether the current theme was a plain theme or not would decide whether to use this code or to use the pixmap theme rendering code. It soon became clear that this approach was neither efficient nor desirable. The UI delegates' code would become too big and complex. There was also another way which plain themes could be supported by Swing; the Metal LAF provides an option of creating *Metal themes* that lets the user controls simple properties as colors and fonts. So a Metal theme that parses the plain theme's color assignments and sets those as defaults could quite easily be implemented instead.

3.3 Rethinking the design after code and prototype review

After the prototype had been finished I had a meeting with my Sun advisor and some other people from the Swing Team to discuss and evaluate it. The key points and decisions were:

- The parser needed to be redesigned (as already planned). By this time I had found and studied the source code for the original Linux GTK parser and now understood the mechanism for the parsing and the identifying of the images.

- The caching of all the images for each component was not desirable, as the number of images that had to be held in memory would be too great. It might be feasible for an application with a few components, but for large applications it would leave a too great memory footprint. For some components, e.g. a JButton would for each instance of the component have to cache seven (7) images [normal state image, rollover state image, pressed state image, focus image, normal state with focus image, rollover state with focus image, disabled state image]. A more scale-on-fly-painting technique would have to be used. This would eliminate the need for any caching, but also required a fast algorithm for the painting.

3.4 Actual design used

The parser's design
Since the themes' gtkrc files are consist of image definitions which some make up a style, and styles then make a theme, is was natural to create an object design hierarchy as showed in Figure 3-3. An ImageData object represents every image definition. A number of ImageData objects plus a style's color assignments is held in a StyleData object. Finally a ThemeData object holds a

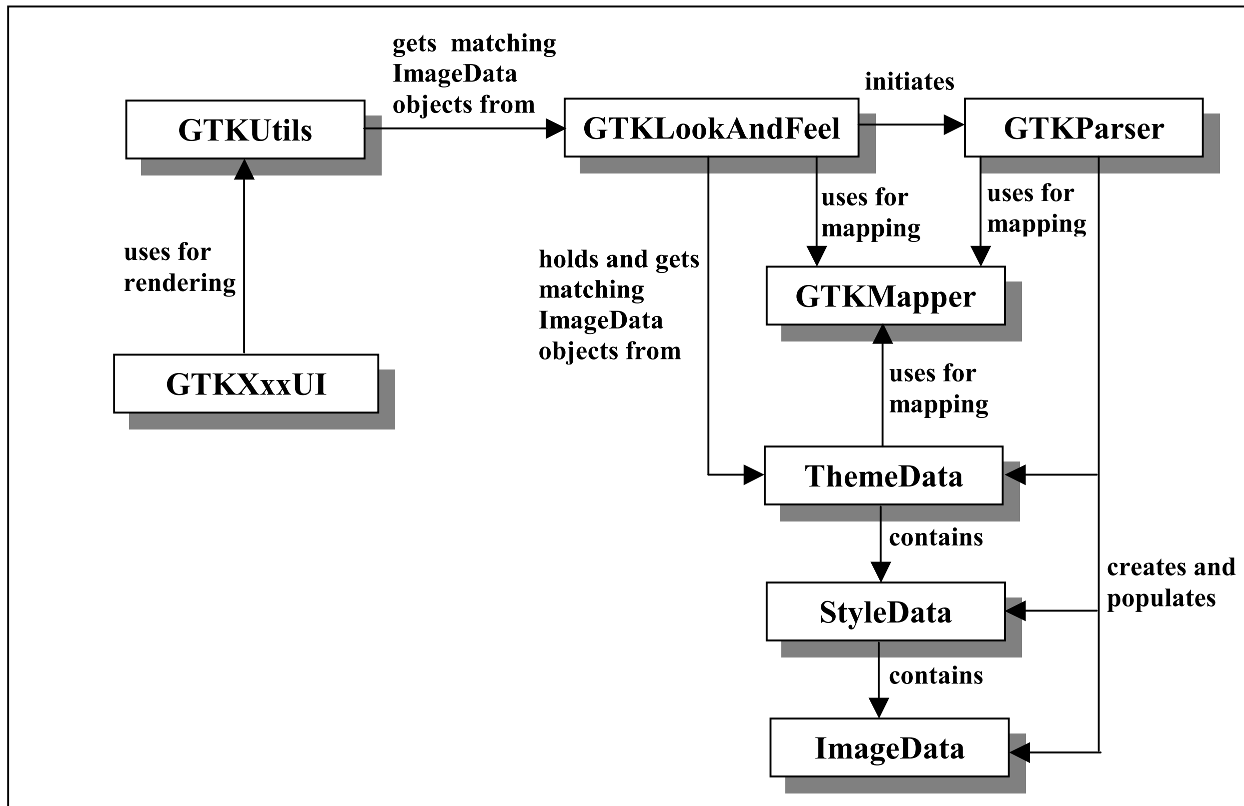


Figure 3-3. The GTK package's key classes and their dependencies.

number of StyleData objects, where one of them is the default style. The ThemeData object will after the parser has finished be stored in GTKLookAndFeel class, for easy access.

The layered on-the-fly-scaling-and-painting design
Since the UI delegates could not store all the images needed for its rendering the original images will instead be stored in the ThemeData structure, which is accessed through the GTKLookAndFeel class. From there all the interested parts can access the images, or rather the ImageObjects, when they need them for the rendering. The only information the needed to fetch an image is an identifier, i.e. a string, of which state the image is supposed to represent.
All an UI delegate has to do when it wants to be rendered with an image is to call a helper method placed centrally in the GTKUtils class. The arguments to the rendering helper method is what size it should be drawn, the identifier of what ImageObject to use and a reference to the UI delegate's drawing area. The helper method will then take care of the rest; fetching ImageObject, check its painting properties (stretching allowed? has overlay images? etc) and then finally paint the image. Thus the "image painting logic" will be held in the GTKUtils class, hidden from the UI delegates.

The “image matching logic”, which will be discussed more in the next section, will be held in the ThemeData object, the GTKLookAndFeel class and a special mapping class called GTKMapper. Some performance enhancing “reference caching” will be also in a helper method of the GTKLookAndFeel class.

The layers in the design begin to seem clear, as shown in Figure 3-4: a top layer with the UI delegates, below a layer in the GTKUtils that requests ImageObjects and renders the components, below that the GTKLookAndFeel class which stores the theme data and obtain the correct ImageData object and at the very bottom the parser which initially “translates” the gtkrc file into a ThemeData object and stores it in the GTKLookAndFeel class.

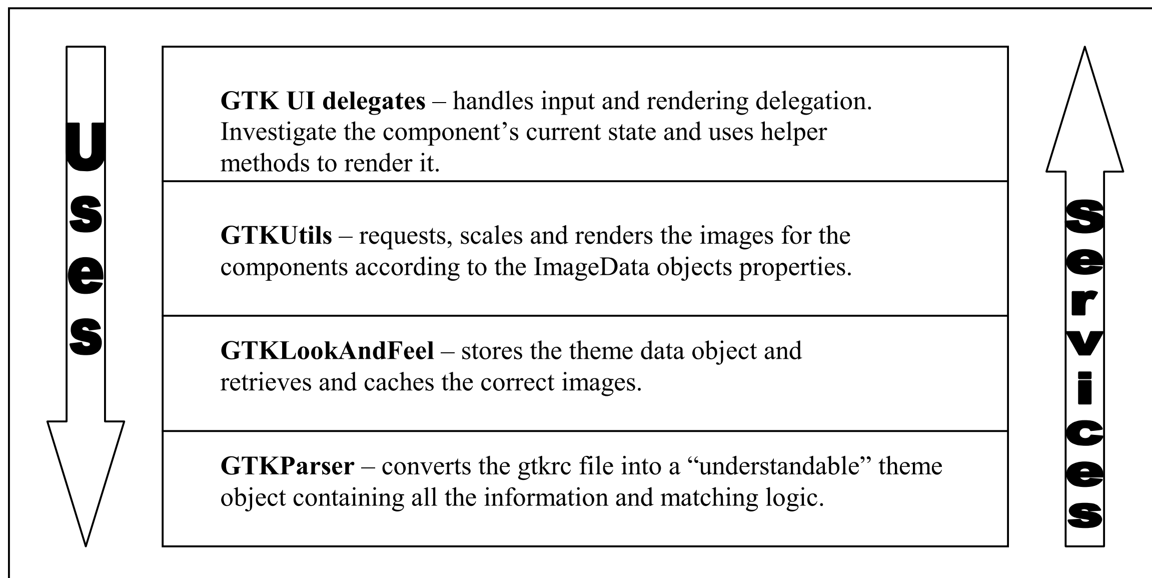


Figure 3-4 The layers in the design and their responsibilities.

4. Implementation

4.1 System setup and tools used

The systems and tools used during the project included:

Java version 1.3

As stated before the version of Java used was 1.3, also known as project Kestrel within Sun. This version was chosen to take advantage of the great performance improvement from 1.2 to 1.3. When I started the project version 1.3 was still in beta, but the code was more or less frozen. The final release was made in March. This meant that the package could not be tested on a Linux-running computer during the project, but since Java is write-once-run-anywhere it did not matter. The developed package is can be used on other platforms as well. The work on a version 1.3 of Java for Linux was well underway and would suitably be released roughly by the time the project finished.

Sun UltraSPARC with Solaris 7 operating system
On this workstation most of the development took place. It was a natural decision since it is the main development platform at Sun Microsystems. Different versions of Java, editing tools and an architecture building/compiling the project were available on the internal network. The source code editing was mainly done in Emacs and for the compiling the Swing Team's existing customized make files were used.

Gateway 400 MHz PC with Windows NT and Red Hat 6.0 Linux operating systems
This workstation was used to test the implementations under a high-end Windows Java Virtual Machine (JVM) and to use the Linux system as a reference.

Dell 133 MHz MMX laptop with Windows 95 and Red Hat 6.0 Linux operating systems
This laptop was used to test the implementations on a low-end computer using the Windows JVM and also as a Linux reference when not in the office. Some days I worked at Sun's drop-in office in downtown San Francisco.
OptimizeIT
A performance test tool used for Java applications. The application shows statistics and information about an application's time-consuming, memory usage, number classes loaded etc. It was used to test and optimize the package.

4.2 Implementation of the design goals

When implementing, the obvious place to start was in the lowest layer and build up from there. In the next sections the most important classes in the package and their functionality will be described, also starting from the lowest layer and move upwards towards the higher-level classes. Interesting and important parts of the source code will be discussed and explained. If a closer look into the code is needed the complete source code, or JavaDoc, is available in Appendix I. Note that all the components' UI delegates have not yet been implemented, a few still remain to be done.

4.2.1 The ImageData, StyleData and ThemeData classes

As showed in Figure 3-3 the gtkrc files' information will be stored in ImageData, StyleData and ThemeData objects. These classes are a logical representation of the information in their counterparts in the files and methods to access them. Below follows a detailed description of these classes: (The complete source code can be found in Appendix 3)

The ImageData class
An ImageData object has uninitialized data members for all the properties that can exist in an image definition in a gtkrc file. An explanation of its data members is shown in Figure 4-1.

```
class ImageData {
    public final static int LEFT = 0;
    public final static int RIGHT = 1;              Static definitions of a border sides.
    public final static int TOP = 2;
    public final static int BOTTOM = 3;

    public StyleData styleData;                     Reference to which style it belongs to.

    public String function;
    public boolean recolorable;
    public String detail;
    public String file;
    public int[] border = {0, 0, 0, 0};
    public boolean stretch;
    public String overlay_file;
    public int[] overlay_border = {0, 0, 0, 0};    Properties which have counterparts in the
    public boolean overlay_stretch;                 gtkrc files. The are set to the same value
    public String gap_file;                         as in the file, if it exists.
    public int[] gap_border = {0, 0, 0, 0};

    public String gap_start_file;
    public int[] gap_start_border = {0, 0, 0, 0};
    public String gap_end_file;
    public int[] gap_end_border = {0, 0, 0, 0};

    public BufferedImage image = null;              The actual images to be used. They are not
    public BufferedImage overlay_image = null;      loaded before actually used for rendering.

    //The haveFoo  attributes indicates whether the Foo-attribut have been
    //set by default or from an ImageData. False = default;
    public String gap_side;
    public boolean haveGap_side;
    public String orientation;
    public boolean haveOrientation;                 These properties also have counterparts in
    public String state;                            the files like ones above, but also each
    public boolean haveState;                       have an boolean indicator showing if they
    public String shadow;                           have been set for the particular object.
    public boolean haveShadow;                      These indicators are used in the matching
    public String arrow_direction;                  method of the ThemeData class.
    public boolean haveArrow_direction;
}
```

Figure 4-1. The ImageData class.

The StyleData class

This class has two main parts, the list of ImageData objects that belongs to its style and the style's properties, i.e. name, font and colors.
The ImageData list holds any number of ImageData objects that are defined within the style in the gtkrc file. ImageData objects can be added to the list through the `addImageData()` method and retrieved through the method `getImageData()`. The whole list can also be retrieved trough the `getImageDataList()` method.
A style's own properties, except from its name, are only set if they are defined within its style. Only the default style, which is recognized by its name 'default', have its color values set by default. This is done because the default style is the "end station" for searches in styles and therefore has the default GTK settings. In the Linux GTK package these colors are built-in into the pixmap theme engine. The class also has a data member that tracks what focus image the style uses.

The ThemeData class
The ThemeData class represents the top instance in the hierarchy and is essentially what differentiates one GTK theme from another. It holds all StyleData and ImageData objects that the theme consists of. It also manages the searches, or matching, of images, keeps track of the default style and provides a helper method for inheriting styles.

It holds all the styles in a list of StyleData objects, which can be accessed with these methods:

```
public void addStyleData(StyleData style)
public  StyleData getStyleData(int indexInList)
public  StyleData getStyleData(String styleName)
public  StyleData[] getStyleDataList()
public  StyleData getDefaultStyle()
```

The class's very important image matching method, `matchThemeImage()`, will be described in the image matching algorithm section 4.2.3 below.

4.2.2 The parser

The GTKParser class contains the methods that read, parse and store the information from the gtkrc files. By calling the method `parseThemeFile(String themeDir)` with the directory of theme files as the argument, it reads the whole file into a String object. This string is in turn passed to the method `parseContent(String content)` where the actual parsing begins.
Through the use of Java's own StringTokenizer class the whole content string is tokenized (divided) into separate strings. The tokenization separates all parts of the initial string that is separated by whitespace or other optional characters. After that you extract the tokens one by one. In this case the StringTokenizer is set to divide upon the characters '=', '\', '"' and ','.
The parsing starts by identifying which one of the top-level tags come first, either a *style* or a *class* statement. Based upon what tag it identifies it calls the next method, either `parseStyle()` or `parseClass()`. These methods in turn tries to identify the next tag, e.g. an image definition or color assignment, and call the appropriate method for handling that information. There exist special parsing methods for all the different tags in the file and which knows how to handle/decode them.The StringTokenizer object is sent along the whole time so that each method can extract the next token if they need to. It continues like that and creates the necessary objects for holding the information on the way until the whole content string has ended, or if an error has occurred, it stops. Figure 4-2 below shows a sequence diagram on how the parsing works.

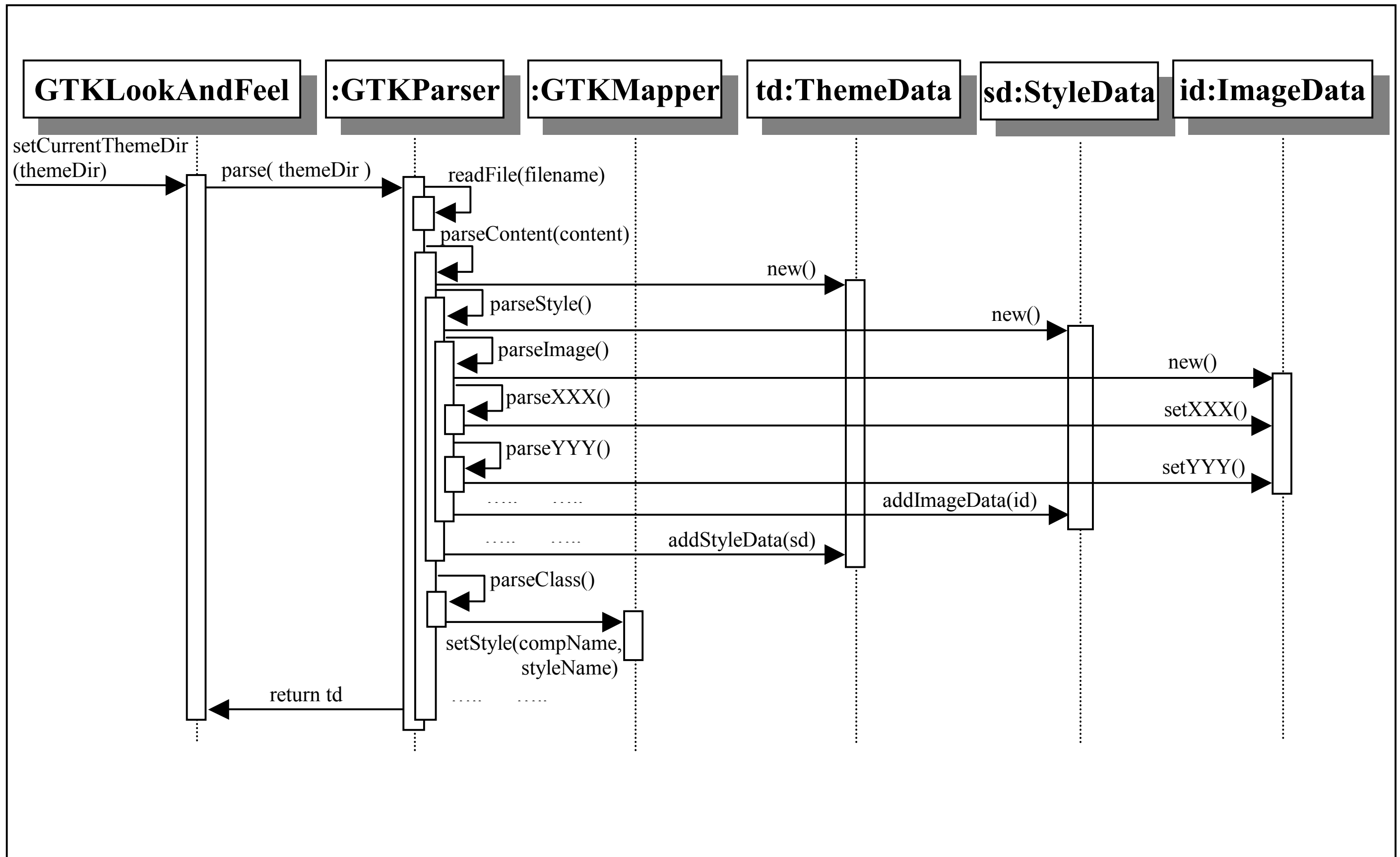


Figure 4-2. Description of how the parser works.

Since the GTK components that are listed in the gtkrc files do not have the same name as their Swing counterparts; a mapping between them is needed to assign a style to them. This mapping is done through the class GTKMapper, a class that holds all the mapping logic for this GTK LAF package. As will be described in Section 4.2.3 it manages the mapping trough the use of a table for each Swing component. So when the method `parseClass()` is called it in turn calls the method `setStyle(String gtkCompName, String styleName)` of the GTKMapper class to assign the style to the GTK component and automatically to its corresponding Swing component.

4.2.3 The image matching algorithm

A very central part of the whole package is how to identify what image to use where. It is sometimes obvious what the image definitions in the gtkrc files are meant to be, but sometimes it

is very hard to understand which image to use. As mentioned above, when implementing the prototype the original GTK algorithm used for matching the images had not been found. This soon led to great difficulties in trying to unique identifying all the image definitions. When the source code for the Linux GTK package was found and studied the technique used became clear. A method called matchThemeImage handles the matching. Its input is an ImageDataDesc describing the image it requests and the name of the component. The output is a matched BufferedImage. The ImageDataDesc object is special holder object, that contains all the properties needed to describe an image.
First of all the component's assigned StyleData is retrieved or the default style if no style had been assigned to it. The properties in the argument are the same that exists in the ImageData objects. They are compared with all the style's ImageData objects' properties in a large **if** – statement. Below the essential parts of the method are shown.

First of all, get the assigned style for the component through GTKMapper's getStyle() method. If no style has been assigned the default style is used.

```
[…]

//Get what style to use
  StyleData style = null;

  //Use the default style or get one by mapping on the component key.
  if( compKey.equals("default") ) {
    style = getDefaultStyle();
  }
  else {
    String compName = compKey.substring(0, compKey.indexOf(".") );
    String styleName = GTKMapper.getStyle(compName);

    Style = getStyleData( styleName );

    if( style == null ) {
      style = getDefaultStyle();
    }
  }
```

By now a style has been retrieved and a list of all its ImageData objects is fetched. This list will be iterated through to try to find a match in the if statement.

```
//Get the ImageData list from the style.
ImageData[] imageList = style.getImageDataList();
ImageData currentData;
int listSize = imageList.length;

int i = 0;

while( i < listSize ) {

    currentData = imageList[i];

    if( ( currentData != null ) &&

              ( function.equals( currentData.function ) ) &&

              ((( currentData.haveState ) &&
                ( state.equals( currentData.state ))) ||
               ( !currentData.haveState )) &&

              ((( currentData.haveShadow ) &&
                ( shadowType.equals( currentData.shadow ))) ||
               ( !currentData.haveShadow )) &&

              ((( currentData.haveArrow_direction ) &&
                ( arrowType.equals( currentData.arrow_direction ))) ||
               ( !currentData.haveArrow_direction )) &&
```

```
            ((( currentData.haveOrientation ) &&
              ( orientation.equals( currentData.orientation ))) ||
             ( !currentData.haveOrientation )) &&

            ((( currentData.haveGap_side ) &&
              ( gapSide.equals( currentData.gap_side ))) ||
             ( !currentData.haveGap_side )) &&

            ((( currentData.detail != null ) &&
              ( detail.equals( currentData.detail ))) ||
             ( currentData.detail == null ))) {

            return currentData;
    }
    i++;
}
```

If no matching ImageData object has been found in this style, so try the default style. If the current style already is the default style then return null, an indicator of that there was not any matching image in this theme.

```
 //If an image wasn't found, try the default style.
 if( style.name.equals("default") ) {
    return null;
 }
 else {
      return matchThemeImage(state, shadowType, detail,
                             arrowType, orientation, gapSide,
                             function, "default" );
}
```

The if statement is essentially the whole matching algorithm. It provides a mechanism for requesting a very special image and if does not exist still getting another image, the next best matching image.

To know what set of properties that should be used to get a certain image one have to look into the source code of the Linux GTK components. For example in the file gtkbutton.c one can see in its painting function that it calls a painting function called paint_box() with some of the properties depending on which state it is in. The method paint_box() is special method for painting flat rectangles, e.g. button. It adds some extra properties before calling the GTK method match_theme_image() with all the properties and then gets the correct image object. So by looking at source code of the components one can see which properties each component and its states are identified by.

4.2.3 The GTKMapper class

GTKMapper is a class where all the *mapping logic* has been placed. What I mean with *mapping logic* is all predefined data that connects the Swing components with the gtk components and their images. All matching and mapping cannot be done programmatically; sometimes it is necessary to by hand define certain relationships. In the GTKMapper class there is two important mapping functions, one for connecting Swing components with GTK components and styles, and one for mapping the Swing components different states with an appropriate ImageDataDesc.

Component mapping

This mapping maps between initially Swing components and GTK components, and eventually between Swing components and their styles. It uses a table with a row for each component and three columns as shown in Table 4-1.

Table 4-1. The table used for component mapping.

Swing component	GTK component	Style (Initially empty)
JButton	Gtkbutton	button
JRadioButton	Gtkradiobutton	checkradiobutton
JCheckBox	Gtkcheckbutton	checkradiobutton
JTextField	Gtkentry	entry
JPanel	Gtkwindow	<empty>
JComponent	Gtkwidget	default

The two first columns are defined "by hand" when the table is initiated, but the third column for the styles is filled as the parser discovers and registers new style assignments in the gtkrc file. The parser calls the method

```
public static void setStyle(String gtkCompName,String styleName)
```

which searches through the table after the GTK component name. When the correct name is found the style name is inserted in the row's third column. All the components will not have a style assigned to them and their third column entry will remain empty, a sign to use the default style instead. 'JComponent' and 'gtkwidget' are the "components" which the default style is mapped onto. When so a Swing component's UI needs to find what style it should use it calls the method:

```
public static String getStyle(String swingCompName)
```

The method searches through the table after the Swing component, when it is located it retrieves its third column entry and returns it. If the entry was empty the style 'default' is returned instead.

ImageDataDesc mapping

This mapping translates a certain Swing component's state into a set of GTK image definition properties, stored in an ImageDataDesc object, which represents that state. Table 4-2 shows a part of the table that stores the mappings. A unique string representing the component and its state is stored in the first column and the corresponding ImageDataDesc object in the second column. This table is defined and constructed when the theme is created and never changed after that. As mentioned before, the values for the ImageDataDesc is obtained by studying the source code of the corresponding GTK component.

Table 4-2. An excerpt from the ImageDataDesc mapping table.

Swing component state	**Corresponding ImageDataDesc object**
`"Button.normal"`	`new ImageDataDesc("normal", "out", "button", "up", "horizontal" , "bottom", "box" ),`
`"Button.rollover"`	`new ImageDataDesc("prelight", "out", "button", "up", "horizontal" , "bottom", "box" )`
`"Button.pressed"`	`new ImageDataDesc("active", "in", "button", "up", "horizontal" , "bottom", "box" )`
`"Button.disabled"`	`new ImageDataDesc("insensitive", "in", "button", "up", "horizontal" , "bottom", "box" )`

The table is accessed through the method,
`public static ImageDataDesc getImageDataDesc(String key)`

where the argument is the component state string that is requested.

Resetting the table entries
When a new Swing GTK theme is to be installed it is important to clear the old theme's style mappings table first. In the GTKLookAndFeel class there is a method called `uninitialize()`, which is responsible for resetting various data in a theme before it is replaced. It takes use of GTKMapper's method `clearAllTables()`, which clears all tables. ((The `uninitialize()` method also uses the GTKMapper method `getMapKeys()`, which returns all the component-state keys from the ImageDataDesc table , to clear the UIDefaults table from the entries in it.))

4.2.4 The GTKLookAndFeel convenience methods

The objects and methodsresponsible for the component rendering, mainly in GTKUtils, do not access the tables and data in the GTKMapper and ThemeData classes directly. There is a layer on top those made up of methods in the GTKLookAndFeel class. They provide an easy way to access the theme's data and also enhance the performance by caching some results.
The resulting ThemeData object from the parsing is stored in the GTKLookAndFeel class, but it is only accessed through convenience methods in the GTKLookAndFeel class.

The getImageData(String compKey) method
When the rendering method, usually a helper paint method in GTKUtils, needs to get the appropriate ImageData object to use for the rendering it calls this method. The argument to the method is a string that identifies the component and its current state, e.g. 'Button.normalImage'. The component's UI delegate has passed this key to the painting method from its own painting method. The getImageData method is responsible for obtaining the right ImageData object based on the key, load an unscaled version of its image and store in the object, and cache the object for future requests.
To get the correct ImageData object the corresponding ImageDataDesc is obtained from the GTKMapper class using the key. With the ImageDataDesc the current ThemeData object's `matchThemeImage()` method is called and the ImageData object is obtained. See the code for this below.

```
ImageData imageData = null;
[…]
ImageDataDesc desc = GTKMapper.getImageDataDesc(key);
[…]
imageData = themeData.matchThemeImage(desc, key);
```

Every ImageData object contains the filename and a java.awt.image.BufferedImage both for its "real" image and for its overlay image. The filename is always existent since the parser stores it there, but the BufferedImage object is not instantiated automatically. (The overlay image filename exists only if it existed in the image definition) It is the `getImageData()` method's responsibility to create a BufferedImage based on the filename and store it in the object. To do this it uses a static helper method in GTKUtils, `loadImage(String filename)`, as shown in the code excerpt below.

```
if( imageData != null) {

  if( imageData.file != null && imageData.image == null ) {
    imageData.image = GTKUtils.loadImage(imageData.file);
  }

  //If an overlay image exists, load it too.
  if( imageData.overlay_file != null && imageData.overlay_image == null ) {
    imageData.overlay_image = GTKUtils.loadImage(imageData.overlay_file);
  }
}
```

To improve the performance of the matching, all the ImageData objects, or rather the references to them, are cached once they have been used. The caching works by storing the already used ImageData objects in a java.util.ArrayList, the imageDataList, and save their list index (position) in the UIDefaults table. The index number is stored with the component state's key that is used for the identification, e.g. 'Button.normalImage'. So the first thing that is checked every time `getImageData()` is called is whether the argument key has an entry in the UIDefaults table. If not, do the whole matching and cache it. If it has an entry, use the entry as an index to the ImageDataList and retrieve the ImageData object directly. The code for it is shown below:

```
ImageData imageData = null;
Integer index = (Integer) UIManager.get(key);

//Test if already have been tried, and did not exist.
if( index != null && index.intValue() == -1 ) {
   return null;
}

//Not cached? Then get the ImageData and cache it.
if( index == null ) {
[… fetching the correct ImageData object …]
if( imageData != null) {  //Add the data to the cache
   //Does it already exist in the cache??
   if( imageDataList.contains(imageData) ) {
      index = new Integer(imageDataList.indexOf(imageData) );
   }
   else {  //It is not in the cache, so load the images and cache it.

      […load the images…]

      //Add it to the cache
      imageDataList.add(imageData);
      //Get the index of the latest addition.
      index = new Integer(imageDataList.size() - 1);
   }
   //Save the index for fast future access.
   UIManager.put( key, index );
      return imageData;
   }
   else {
      //Store an indicator of that there is no image for the key.
      UIManager.put(key, new Integer(-1) );
      return null;
   }
}
```

```
else { //Retrieve and return the cached ImageData object.
   return (ImageData) imageDataList.get(index.intValue());
}
```

When uninstalling a theme it is important to empty the cache and to remove all the UIDefaults's cache index entries. This is handled by the before mentioned GTKLookAndFeel class' method `uninitialize()`, which is described in Section 4.2.7.

Font and color convenience methods
The GTKLookAndFeel class also provides convenience methods for accessing the colors and fonts for the components. These methods are though not directly accessed from the rendering methods, but from within the UIDefaults. A part of the architecture that is inherited from the BasicLookAndFeel is the automatic installation of colors and fonts. These installation methods take their values from the UIDefaults table, for example the foreground color of a button is stored in the table as 'Button.foreground' and the its font as 'Button.font'. This is very convenient, as all you have to do is to replace the entries in the table with your own color definitions, and they are automatically installed. This is where the GTKLookAndFeel's color and font methods come in use, in the initial definition of the UIDefaults. For example the entry for the foreground color of a button becomes:

```
"Button.foreground", getNormalForegroundColor("Button"),
```

There is a method like `getNormalForegroundColor(String compName)` for each color that can be defined in the themes; see Appendix IV. What the methods do is to simply find the right StyleData object for the component and then query it for the color. The font method works in a similar way.

4.2.5 The painting methods in the GTKUtils

The GTKUtils class contains static rendering helper methods that make up painting layer, plus some additional helper methods. The UI delegates use the rendering methods when they need to be rendered. The methods hide the details of getting, stretching and painting the images from the UI delegates.
The first important method is the one that loads the images from the file system:

```
public static BufferedImage loadImage(String filename)
```

It takes the filename and adds it to the current theme directory, using the static method `GTKParser.getThemeDirectory()`, to get the full path to the image file. The image is loaded by using the method `Toolkit.getDefaultToolkit().getImage()` and the loading is supervised by a MediaTracker. MediaTracker is a utility class used to track the status of media objects. Since the Toolkit's image loading method returns a java.awt.Image object and we need a java.awt.image.BufferedImage, for the scaling etc, we create a new BufferedImage with the same dimensions and draw the image upon it.
There is also a slightly different version of the loading method called `getScaledImage()`. That method takes the component key, width and height as arguments, and returns the image in those dimensions if the ImageData's stretch property allows it.
The most important method in the class is the one that scales and paints the gtk images:

```
public static void gtkPaint(JComponent c, Graphics g, String key,
                            boolean paintShadow,
                            int offX, int offY,
                            int w, int h)
```

It obtains the correct ImageData object based on the key, and based on the ImageData object's properties stretches, adds overlay images and paints the image. To comply with the gtk scaling, as explained in Figure 3-2, the image is divided up in to several parts, which are all scaled and painted individually. If the argument paintShadow equals true, a shadow image will be retrieved and painted around the real image. The offX and offY arguments set the offset from which the image will be painted on the graphics context g, using the width w and height h.

Another painting method is available as well, `onlyBorderPaint()`, which only paints the border of an image. It is used by some border classes for their rendering.

Some components, e.g. JPanel, do not use stretching for their images, instead they use a tile of the image. Tiling means that the original image is painted repeatedly over the area, both vertically and horizontally, until the whole area is covered. To provide support for this a helper method exists that creates a tiled image from the based on the image in the argument:

```
public static Image CreateTiledImage(JComponent c, Image oldImage, int newWidth, int newHeight)
```

4.2.6 Painting methods in the UI delegates

It is in the UI delegates that the rendering of the components is initiated. It is here that the knowledge about which state the component is in exists and is used to dictate the rendering. The UI delegate renders itself when its `paint(Graphics g, JComponent c)` method is called. In the paint method the usual behavior is to query the component model to find out the current state and then call the rendering method in GTKUtils with the arguments based on the state. Below is an excerpt from GTKButtonUI's paint method.

```
public void paint(Graphics g, JComponent c) {
   AbstractButton b = (AbstractButton) c;

   if( b.getModel().isEnabled() ) {
      if( b.getModel().isRollover() && !b.getModel().isPressed() ) {

         if( b.hasFocus() ){ //If in focus, decrease painting size.
            GTKUtils.gtkPaint(b, g,
                              "Button.rollover", false,
                              1, 1,
                             b.getWidth() - 2, b.getHeight() - 2);
         }
         else {
            GTKUtils.gtkPaint(b, g,
                              "Button.rollover",  false,
                              0, 0,
                             b.getWidth(), b.getHeight() );

         }
      }
      else { //Normal state
         if( b.hasFocus() ){ //If in focus, decrease painting size.
```

```
            GTKUtils.gtkPaint(b, g,
                              "Button.normal", false,
                              1, 1,
                              b.getWidth() - 2, b.getHeight() - 2);
         }
         else {
            GTKUtils.gtkPaint(b, g,
                              "Button.normal",  false,
                              0, 0,
                              b.getWidth(), b.getHeight() );
         }
      }
      else { //Button is disabled
            GTKUtils.gtkPaint(b, g,
                              "Button.disabled",  false,
                              0, 0,
                              b.getWidth(), b.getHeight() );
      }
}
```

Most of the other component UI delegates operate in a similar fashion although some slight differences exist. Some UI delegates also have to at creation-time create some resources needed in the future. For example the GTKRadioButtonUI have to create and install the icons representing its unselected and selected state. Below are parts of its installation code shown (the installDefaults and loadRadioIcons methods).

```
protected void installDefaults(AbstractButton b) {
   super.installDefaults(b);

   loadRadioIcons(b);

   […]
}

protected void loadRadioIcons(AbstractButton b) {
   unselectedIcon = new ImageIcon(
           GTKUtils.getScaledImage(getPropertyPrefix()+ "unselectedImage",
                                 10, 10 );

   selectedIcon = new ImageIcon(
           GTKUtils.getScaledImage(getPropertyPrefix()+ ”selectedImage",
                                  10, 10 );


   b.setIcon( unselectedIcon );
   b.setSelectedIcon( selectedIcon );
   b.setRolloverSelectedIcon( selectedIcon );
}
```

Some UI delegates, like GTKPanelUI, sometimes uses a tiled image instead of a stretched one. The tiling operation can be quite time-consuming if the original image is too small, therefore the ready tiled image is cached. Since the GTKPanelUI is a shared static instance the image is cached in the component’s client properties (described in section 2.2.3).

4.2.7 Other important classes and methods

It not only from the UI delegates that painting is performed. In the classes GTKBorders and GTKIconFactory classes and methods for painting borders and icons are placed.

Borders

Normally in Swing you install the border for every component, and it then paints itself. Since the GTK images already have their borders defined within the images, there is no need for ordinary borders. Instead the Swing borders are used for focus and shadow painting. As mentioned before, when a GTK component has focus a special focus border image is painted around the component. Shadow images are used by some components, e.g. textfields, to give a 3D feeling. The shadow images, which can be "in" or "out", are painted around the component like a border. The FocusBorder is like an ordinary Swing border, except that it checks whether the component has focus each time it is asked to render. If the component has focus, it fetches the component's focus image and paints it. If not, it does not render at all.
The ShadowBorder simply gets the component's shadow image and renders it every time. Because most of the components usually have the same focus image, specified in the default style, this default ImageData object is cached for faster access.

Icons

Some icons in Swing are images that are loaded when they are needed, but some icons are drawn programmatically. It means that each time that the icon needs to be rendered its `paintIcon()` method is called and instead of using an image it draws all the lines, dots and graphic it needs. Many of these self-rendering are centrally placed in a class called GTKIconFactory (MetalIconFactory for the Metal LAF). In the GTK PLAF most of the images are inherited from BasicLookAndFeel, why there is not that many inner icon classes in the GTKIconFactory. The ones that exist are radio and checkbox icons for menuitems.

Theme-uninstalling methods

When a user of the GTK package switches GTK theme, i.e. wants to start using a new set of GTK images, certain measures have to be taken to ensure that the previous theme is fully uninstalled. These steps are collected in the `uninitialize()` method of the GTKLookAndFeel class. This method basically does four things:

1. Calls GTKLookAndFeel.resetImageDataList() which clears the GTKLookAndFeels's image caching list.
2. Calls GTKBorders.resetBorders() which resets all the static ImageData objects used for caching in the GTKBorders class.
3. Calls GTKMapper.clearAllTables() which clears all the mapping tables as described above in section 4.2.3.
4. Removes all the caching index entries for the components in the UIDefaults table by setting all the entries got from the method GTKMapper.getMapKeys() to null.
5. Nullifies various static references in the package to ensure that they are garbage collected properly.

4.3 Optimizations and improvements

To fulfill the requirements stated in section 1.1.1, to improve the performance and try to minimize the memory footprint, certain improvements had to be made. These optimizations and improvements and their impact will be described in this section and then evaluated in section 5 on tests and evaluation. Some of the optimizations were made after studying the package in work in the testing and probing program, OptimizeIT. When studying what and how many classes that are loaded by the package, some hints were given where bottlenecks could be found.

The first optimization step taken was when deciding on what design to use. By abandoning the initial idea of caching each component's different images, the memory footprint became significantly smaller. The speed performance may have deteriorated, but hopefully not too noticeable.
Another improvement already mentioned was the ImageData caching in the GTKLookAndFeel class (see section 4.2.4). This caching speeds up the ImageData retrieving by not having to go through the whole list of StyleData and ImageData objects, and try to find a match.
Some components, e.g. GTKPanelUI, use a tiled image. This tiling operation can be time-consuming if the original image is too small. Therefore two actions are taken to improve this. First the tiled image is cached within the component, and is only re-tiled if the component changes its size. Second, if the original image is too small, less than 10 by 10 pixels, a new tiled image first, 5 times bigger, is stored instead in the ImageData object. This improves the performance of the tiling operation significantly.
As mentioned before most of the components use the same focus image, and therefore its ImageData object is cached. This speeds up the focus painting, which is a quite frequent-occurring operation.
When storing objects in the UIDefaults Swing gives the option of storing objects as a ProxyLazyValue as well as storing them just as they are. This object is a sort of wrapper around the original object. The use of these objects improves the time for loading a LookAndFeel, at the cost of a slight performance reduction the first time they are used. They do this by not instantiating the object or loading the class until it is actually needed. This effect will not be tested since its gain is already proven [Ref 7].

Optimizations made after studying the package with OptimizeIT
Several performance issues were discovered when closely studying the package's behavior with the OptimizeIT. OptimizeIT lets you study things as: total memory used, different methods time-consuming, number of loaded classes and the use of temporary objects. The main discoveries were:

- When switching from a GTK theme to another, or a Swing theme like Metal, some of the objects created by the GTK theme were not garbage collected. This led to that the memory used by the application kept on increasing. The problem originated from that an object can not be garbage collected if there is still a reference to it. This is a common problem when using static references, because they can keep a reference alive even though the class is not in use any more. So by explicitly nullifying all static references when uninitializing a theme this problem is overcome.

- The parser created many temporary objects when parsing the gtkrc files. In almost every parsing method new temporary objects were created to save temporary results. By instead

using static objects that could be reused the number of created temporary objects decreased.

- The GTKParser.readFile() method consumed a lot of time. When studying it closer it could be seen that it was the String operations that was the problem. A String object in Java is an immutable object, i.e. it can not be changed. Instead it creates a new String object that incorporates any change to it. The class StringBuffer on the other hand uses the same object the whole time, which improves the performance but does not provide the same altering operations as a String object do. By replacing some of the Strings used in the method with StringBuffers the performance was improved significantly. (See section 5.2)

4.4 Problems encountered

During the implementation some problems were encountered, some solvable and some not.
One of the problems that could not be solved is due to a bug in the BasicComboBoxUI class. The bug is that when uninstalling the UI delegate, or rather the BasicComboBoxUI that is inherited by the GTKComboBoxUI class, a focus listener is not unregistered when the JComboBox is in the editable state. This leads to that it fires events that have no listeners, causing a NullPointerException to be thrown. I have reported the bug to the responsible person in the Swing Team and for the next release 1.4, which includes a redesign of the JComboBox, it will be fixed.
When loading the images from the file system and then trying to get their width through the getWidth() method they sometimes returned –1. This was due to that they had not yet been loaded completely. The problem was solved by using a MediaTracker to supervise the loading of all images, it keeps track of all the images' loading and does not "release" them until they are fully loaded.
Some of the GTK components turned out not to have any corresponding Swing component, e.g. ruler. These were not implemented to keep Swing's component-set intact (and there was not time for it either). A GTK component as option-menu did have an exact match, but could be considered to be a non-editable JComboBox, and was therefore mapped upon it.
Another problem encountered was that not all components supported rollover sensitivity, i.e. you can tot tell if the mouse pointer is within the component, and more important within certain parts of a component. For example the thumb of a GTK scrollbar changes its overlay image when it is in a rollover state. This can be solved by registering mouse listeners and continuously tracking the mouse movements and compare them to the thumb's current bounds. This has though not been implemented yet, because there were higher priorities, but will probably be added in a future version.
When parsing the font as stated in the gtkrc file, the java.awt.Font.decode() method is used. This method takes a name, style and size as input argument and tries to match with an existing system font. When doing this it has to load all the current system fonts, which turned out to be quite time-consuming the first time it is done. This problem can not be overcome, as it is the only way to decode the font. The exact times consumed by the font decoding operation can be seen in the testing section 5.2. The font decoder also consumes extra memory to hold the system fonts in, which it keeps during the rest of the Java session, adding extra to the memory footprint.

5. Testing and evaluation

5.1 What tests and evaluation used and why

The tests used to evaluate the package are mainly focused on comparing the package "against itself". What that means is that because there is no real similar product to compare it with, the different improvements made to the package is compared to its "unimproved state". So the tests described below tries to evaluate the gains of the optimizations discussed in section 4.
To test the package a simple GUI application with a set of common Swing components was used. The code for the test application, GTKTest.java, can be seen in the source code appendix 2.

5.2 Performance tests and results

5.2.1 Caching all images versus scale-and-paint-on-the-fly

This change of design that was adopted early meant that instead of caching a component's state images, the scaling and painting should be on-the-fly. It is easy to see that it would result in faster initialization and a smaller memory footprint, but probably a more time-consuming painting. Another advantage with the new design is that if a component is resized it makes no difference to the time used for rendering, but for using the old design all the component's images would have to be reloaded.

Memory footprint comparison

During this test sequence the test application was started, with the default Swing Metal theme. Then a GTK LAF theme, using the Ganymede theme (showed on the cover), was initiated and set as current LAF and the application memory (blue line in figure, or the lightest one if printed using grayscale) and heap size (red line, or the darkest using grayscale) was studied. (Heap size is the total memory allocated to the Java VM, while the memory size is the actual size used by the application) After the theme has been fully initialized and is up and running, the memory usage levels. Some of the memory now is held in temporary objects so to see actual memory usage of the theme the garbage collector is forced to run. That is the last dip in the chart. So the last level after the chart represents the applications "resting-level" with the current theme.
Figure 5-1 shows the memory usage for the "new" version, using cached images for all the components, while Figure 5-2 shows the "new" scale and paint on-the-fly version. A significant difference in the between the two versions' memory usage can be seen. The old version peaks at roughly 7400 kilobytes (kb) and then stabilizes at 6500 k, while the new one peaks and stabilizes at 4100 kb. After the garbage collector is run the old version drops to 4000 kb and the new version to 2400 kb. Note also the initial Metal LAF's memory usage, roughly 1500 kb, but keep in mind that it does not have to keep any images in memory.

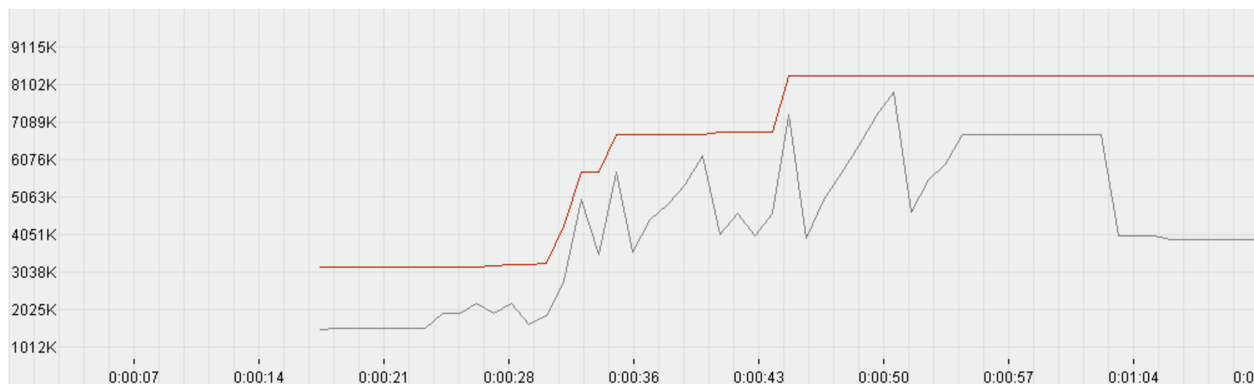


Figure 5-1. Memory usage when installing a theme with the caching design.

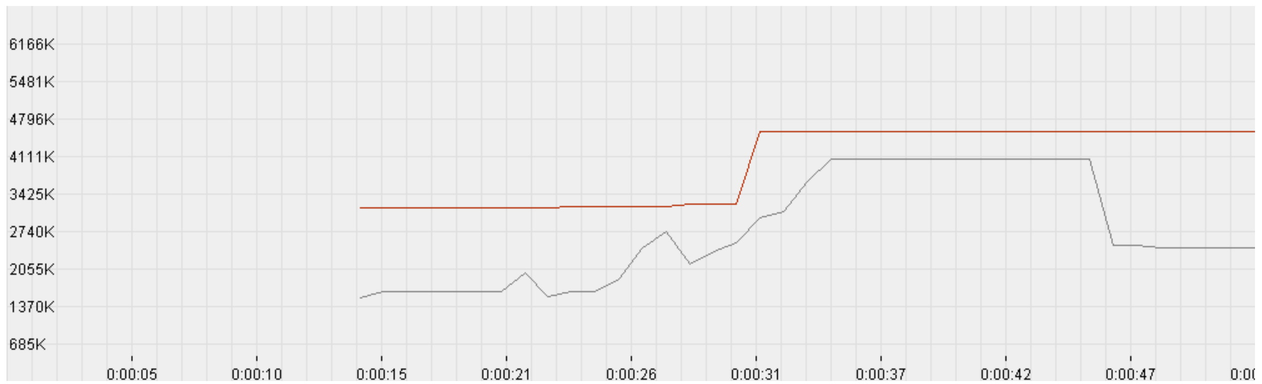


Figure 5-2. Memory usage when installing a theme with the non-caching design.

A closer look at the memory consumption is provided in Table 5-1. It shows the values for three different themes using the different versions and also what classes make up for the biggest memory increase. As can be seen the new version uses considerably less memory, from 25 % using the pixmap theme to 70 % for the BeOS theme. The last column shows the classes responsible for the largest increases. The `int[]` classes that makes up for the biggest increase is due to the image storing and handling, which uses the int arrays to store all the pixels and other info for the images in. The `char[]` and `String` classes mainly originates from the parser, the ThemeData class's content and the UIDefaults, which all contains many strings.
How much memory that is gained by using the new version depends somewhat on the theme used. In the case with the BeOS theme, it uses a few images to visualize many of the components, they share them. Less images has to be loaded and held in memory, and therefore the big difference versus the old version which still have to load all the images although they are the same.

Table 5-1. Memory footprint and the most increasing classes for the caching and non-caching versions.

		Caching version			**Non-caching version**		
Theme	**Initial size (k) (Metal)**	**Top level (k)**	**Level after gc (k)**	**Top 3 increasing classes (k)**	**Top level (k)**	**Level after gc (k)**	**Top 3 increasing classes (k)**
Pixmap	1485	6135	4339	int[] +2559 char[] +132 String +99	4581	3264	Int[] +1623 Char[] +76 String +36
BeOS	1485	5509	3873	int[] +2079 char[] +137 String +107	3129	1677	Char[] +80 String +38 Int[] +28
Ganymede	1485	7200	4032	int[] +2116 char[] +150 String +113	3112	2435	Int[] +708 Char[] +95 String +52

Although not tested here, the new version has another advantage in that the more components an application has, the more memory is saved, relatively. Since they all share the same ImageData objects it does not matter how many that share them, which is not the case with the old version.

Speed performance
It is obvious that the old version using the cached images will be faster than the new one. But how much faster and is it visually noticeable? To measure this the painting method of ButtonUI was timed using Java's built-in time method, `System.currentTimeMillis()`, which return the current time in milliseconds. So by noting the time right before and right after the paint method, its execution time can be got. When just measuring the time for one painting, with the possible getting and rendering of the image, the milliseconds measuring was too coarse. The first time for both the versions was roughly 50 – 60 ms, but after that 0 (zero) ms was reported for each painting. So too get a better picture of the times involved the paint method was forced to paint 1000 times, and the time for that was measured. (Using the pixmap theme) The times for the operations can be seen in Table 5-2. The multiple painting operation was repeated three times to also compare the results after initial getting and stretching had already been done. This time is roughly the 3rd time, after that the values leveled off.

Table 5-2. Time consumed to paint by the different versions.

Theme	**Caching version** – time for painting button 100 times (ms)	**Non-caching version** – time for painting button 100 times (ms)
Pixmap	1st time 220 2nd time 110 3rd time 100	1st time 1150 2nd time 880 3rd time 600

As expected the old version would be faster, due to its caching. It is about 6 six faster than the current used version. This may seem much, but as a user it is very hard to notice any difference, because these are already very short time periods, down in milliseconds.

5.2.2 ImageData caching in the GTKLookAndFeel class

Since all the objects that want to access the ImageData objects have to go through the GTKLookAndFeel class to get the matching ImageData objects from the stored ThemeData, it seemed reasonable to provide some sort of caching there. To skip the ImageData matching mechanism when it already once has been matched for that particular component key could save time. So as described in section 4.3 references to all the processed ImageData objects are stored in an array and their index is stored under the component key in the UIDefaults. So for example the image matching for a 'Button.normalImage' will only have to be done once, which would save considerable time. The test was conducted using the same setup as the test above, by getting the time before and after the `GTKLookAndFeel.getImageData()` method and comparing them. Table 5-3 shows the four first (until it settled) time the method is called and the time to execute them 50000 times. The high number of repetitions was necessary to get a comparable value. The first value is always higher because not only has it not been cached yet (if using new version) but also because the images have to be loaded from the file system the first time it is used.

Table 5-3. Difference in speed with and without ImageData caching.

Theme	**Without cache** – Time for 50 000 getImageData() calls (ms)	**With cache** – time for 50 000 getImageData() calls (ms)
Pixmap	1st time 600 2nd time 390 3rd time 380 4th time 380	1st time 1860 2nd time 1380 3rd time 1430 4th time 1380

The caching speeds up the image matching and retrieving method `getImageData` 3-4 times, which is desirable since the method is called frequently.

5.2.3 Caching and pre-upsizing of tiled images

Because tiling an image can be very time-consuming, for example using a 4*4 pixels big image to fill a JPanel's background a size of 600*400 pixels. To make these tiling operations more efficient for components that often tend to be tiled and large as JPanel and JPopupMenu, caching and pre-upsizing has been used. The pre-upsizing means that if the image is too small, e.g. 4*4 pixels, it is replaced by a larger tiled image created from the original image. The caching simply stores the current tiled background image and uses that instead of re-tiling each time. In Table 5-3 below the different times for painting a JPanel's background using the pixmap and Aqua themes are displayed. The last column represents the version used in the package.

Table 5-3. The performance gains of using pre-upsizing and caching for a JPanel's tiled image.

Theme	Without caching Without pre-upsizing (ms)	Without caching With pre-upsizing (ms)	With caching Without pre-upsizing (ms)	With caching With pre-upsizing (ms)
Pixmap	1st 660 2nd 550 3rd 550	1st 660 2nd 550 3rd 550	1st 720 2nd < 1 3rd < 1	1st 660 2nd < 1 3rd < 1
Aqua	1st 8460 2nd 7520 3rd 6810	1st 990 2nd 830 3rd 770	1st 8846 2nd < 1 3rd < 1	1st 990 2nd < 1 3rd < 1
DigiBlue	1st 710 2nd 600 3rd 600	1st 610 2nd 550 3rd 540	1st 770 2nd < 1 3rd < 1	1st 700 2nd < 1 3rd < 1

By looking at the table above it is easy to spot that the Aqua theme uses a very small original tiling image, only 4*4 pixels, why the pre-upsizing speed the tiling up 9 times. The pixmap theme has 256*256 pixels tiling image and the DigiBlue 128*128 pixels. These are too big to be affected by the pre-upsizing, but as all the themes gain enormously the second it is painted. So although it was decided before not to use image caching in this new design, a few exceptions have to be made to fix bottlenecks, which this was.

5.2.4 Optimization of the GTKParser.readFile() method

Before I started using OptimizeIT to evaluate and test the package I had been using many String objects in the `GTKParser.readFile()` method. The Strings are easy to manipulate and convenient to use, so they were the natural first choice, but OptimizeIT showed that the `readFile()` methods created many temporary String objects and was very time-consuming. After studying the class closer I realized that it was all the Strings and their initialization that was so costly. As mentioned before, a String object can not change size or content, instead a new String object has to be created. This proved to be very undesirable because of all the operations on the strings in the method. By replacing some of the String object with StringBuffers objects, which can the changed, a great deal of performance was gained. Table 5-4 shows the time for the method to run with only String objects and with some StringBuffer objects.

Table 5-4. Testing the use of StringBuffer vs. Strings in the GTKParser.readFile() method.

Theme	Only String objects – time for readFile() method (ms)	Using StringBuffer objects – time for readFile() method (ms)
Pixmap	1200	60
Aqua	770	50
DigiBlue	820	50

The time-consumed by the readFile method decreased dramatically, up towards 20 times! This improvement has another side to it, it lowers the number of Strings used and therefore the memory usage. OptimizeIT's loaded classes chart showed that size of Strings created decreased from **+119 k** to **+90 k**.

5.2.5 Delays for decoding the fonts in the parser

The decoding of what font to use through Font.decode() is a very time-consuming task, because all the system fonts have to be loaded. This delays the initialization of the GTK LAF, but only the first time a GTK theme is loaded. After the first time though, the systems fonts remain in memory leading to that the following decodings are neglectable. If the application is closed and then restarted later, sometimes the new Java VM still keeps some tracking on the previously loaded system fonts, resulting in a faster initialization. The speeds of the `parseFont()` method, with the different scenarios are as follows:

First time Java VM and the method is run	**3570 ms**
First time method is run in a session	**1540 ms**
Second time method is run during the same session	**< 1 ms**

So the first time an application using the GTK package is started there will be a noticeable delay due to the font decoding. Fortunately once the system fonts has been loaded the font decoding is very fast, not even noticeable.

5.3 Example of usage

One of the explicit requirements on the package was that it should as easy to use as the other existing LAFs. This requirement is fulfilled, as it is very simple to incorporate and use this package into an application. First the package needs to be imported by using Java's import statement:

```
import com.sun.java.swing.plaf.gtk.*;
```

Then set the path to the theme directory, where the GTK theme archive has been extracted. Be sure to catch the IOException that is thrown if the directory does not exist.

```
try {
     GTKLookAndFeel.setCurrentThemeDir(String themeDir);
}
catch {IOException e){
    System.err.println("Error while setting themeDir" + e);
}
```

After that set the GTK LAF to be the current one and it will be created and installed automatically. To update all the components, use the UIManager's convenience method.

```
UIManager.setLookAndFeel("com.sun.java.swing.plaf.gtk.GTKLookAndFeel");
UIManager.updateComponentTree(JComponent topLevelContainer);
```

A full source code example of a simple frame that has a button, a textfield and wants to be able to enter a theme name (full path) and then set it by clicking the button would look like this:

```
import javax.swing.*;
import java.awt.event.*:
import com.sun.java.swing.plaf.gtk.*;
public class MyClass extends JFrame {

   JButton button = new JButton("Try Theme");
   JTextField textField = new JTextField(10);

   public MyClass () {
      //Init the frame
      this.setSize(200, 200);
      this.setLayout(new FlowLayout() );

      //Add actionlistener to the button
      button.addActionListener( new ActionListener() {
         public void actionPerformed(ActionEvent e) {
            setTheme();
         }});

      //Add the components
      this.getContentPane().add(button);
      this.getContentPane().add(textField);
      }

   private void setTheme() {
      String dirPath = textField.getText();
      textField.setText("");

      try {
         GTKLookAndFeel.setCurrentThemeDir(dirPath);
         UIManager.setLookAndFeel("com.sun.java.swing.plaf.GTKLookAndFeel");
         UIManager.updateComponentTree(this);
      }
      catch ( Exception e) {
         System.err.println( "Error while setting theme:"+e);
      }
```

```
    }

    public static void main(String[] args) {
       MyClass frame = new MyClass();
       frame.setVisible(true);
    }
}
```

So the behavior for using the GTK LAF package is very similar to using the existing LAFs.

5.4 Conclusion from the tests

After testing and studying the package with OptimizeIT many important issues unveiled. It gave new insight in how the different parts of the package used memory and processor-time, and the performance could be improved significantly. The tests showed that:

- Not caching all the components' images, but instead fetching and scaling them on-the-fly, reduced the memory usage up to 70 %. The speed was lower, but not noticeable to the user.

- By caching ImageData objects in the fetching phase, the time for getting the correct image was reduced 3-4 times.

- By pre-upsizing and caching certain large images, the time for rendering was in some cases (the Aqua theme) reduced to a fraction.

- Through the usage of StringBuffer objects instead of String objects in the readFile(), the time used to read the file and remove unwanted character and lines was reduced up to 20 times.

- The ease-of-use of the package is satisfying, as it requires as few as two method calls to initiate and use the GTK PLAF package.

- The bottleneck proved as expected to be the scaling and painting. It can though hopefully be sped up with some smarter algorithm and more caches in future versions. Also the computers nowadays get faster all the time and come with more powerful graphics card that will ease the burden for the processor in this kind of rendering.

6. Conclusion and future work

Conclusion

I have created a GTK PLAF package that in an easy way incorporates GTK themes into Swing applications. By improving and optimizing the package its memory footprint and performance was improved significantly, not far from the existing Swing LAFs. The design of the package has four layers, each responsible for certain tasks. From the bottom-up: the parsing layer, the storing and caching layer, the painting layer and the UI delegate layer.
The most important improvements were: to skip the caching for each component and instead use a fetch-scale-and-paint on-the-fly technique, to cache the ImageData objects in the image fetching layer, to cache large background images and to minimize use of costly temporary objects.

These optimizations were tested against the package itself, i.e. with the optimization and without, and drawing conclusions from that. The conclusions were that the improvements were significant.

I have also had the chance to learn Java and Swing at the very place to do it, at Sun Microsystems in Silicon Valley, which was a very inspiring place. Although during the initial stages of the project, it was advancing sometimes slowly as the learning curve of Swing's PLAF architecture is quite steep in the beginning, but after a while as things started to get clearer it sped up. The original time plan had to be modified initially, due to the mentioned steep learning curve, but after that the project ran more smoothly. Some work still remains to be done, a few more components to be added, but with the design of the whole architecture implemented it is easily done.
After the summer's vacation period, the Swing Team and I will decide how this package should be released and when. It will most probably first be released through the Swing Connection website, www.java.sun.com/products/jfc/tsc, later this autumn.

Future work
The use of user customizable applications is definitely a part of the future, and so is Java. So therefore this combination of them a logical step that will probably be appreciated by the user and developer community. I feel that in the future maybe other standards of themes, or skinz, could be added to this package. By replacing the some of the layers in the design, a totally new type of theme could be supported without affecting the UI delegates' top layer.
Another issue is that there are to many different theme standards today, while users want a theme to work on all his programs. A unifying theme/skin standard would be the obvious solution, and there might be one just around the corner, XML User interface Language (XUL, pronounced 'zool'). Recently Mozilla [Ref 8], an open-source project, announced that the next version of Netscape, version 6, will partly be customizable through XUL files. I believe that this standard will be important, and that an incorporation of XUL into this package could be the logical progression. It would make this package one of the frontrunners on the themeing scene.

References

1. **A Swing architecture overview**, Amy Fowler, 1997 [www page]
http://java.sun.com/products/jfc/tsc/articles/architecture/index.html
2. **Fundamentals of Swing: Part I,**
http://developer.java.sun.com/developer/onlineTraining/GUI/Swing1/shortcourse.html
3. **What is Linux.** [www page] http://www.linux.org/info/index.html
4. **What is Linux.** [www page] http://www.li.org/li/whatislinux.shtml
5. **GTK Tutorial.** [www page] http://www.gtk.org/cvs/gnome/ gtk+/ docs/ gtk_tut.sgml
6. **GTK Theme documentation**, [www page] http://gtk.themes.org/php/docs.phtml
7. **Performance improvements in JDK1.3** [www page].
http://java.sun.com/j2se/1.3/docs/guide/swing/PerformanceChanges.html
8. **Netscape plays catch-up with latest browser,** Paul Festa, CNET News.com, August 9, 2000.
[www page] http://news.cnet.com/news/0-1005-200-2481675.html?tag=st.ne.1002.srchres.ni

Other sources used:

Java Swing. R. Eckstein, M.Loy and Dave Wood, 1998.
Java in a Nutshell. David Flanagan, 1997.
The 'LookandFeel' Class Reference. A PLAF Lookup Guide for Swing Programmers.
[www page] http://java.sun.com/products/jfc/tsc/articles/lookandfeel_reference/index.html

Appendixes
I. Code

The code showed here using JavaDoc does not show the whole package. This would take up too much space and it not necessary since many of the UI delegates has a similar structure. Instead the most important architectural classes and examples of UI delegates will be shown.
Further source code can be got by contacting me at weddie@home.se.

---

com.sun.java.swing.plaf.gtk

Class GTKParser

```
java.lang.Object
  |
  +--com.sun.java.swing.plaf.gtk.GTKParser
```

---

public class **GTKParser**

extends java.lang.Object

Field Summary	
private static int[]	**resultBorder**
private static java.lang.String	**themeDir**
(package private) java.lang.String	**token**

Constructor Summary
GTKParser()

Method Summary	
private static java.lang.String[]	**getFontWords**(java.lang.String line) Helper methods that extract the words necessary for the decoding of the font.
protected static java.lang.String	**getThemeDirectory**() Returns the currently used theme directory.
private static boolean	**parseArrow_direction**(java.util.StringTokenizer st, ImageData data) Stores the arrow direction property.
private static boolean	**parseBorder**(java.util.StringTokenizer st, ImageData data) Stores the border property.
private static boolean	**parseClass**(java.util.StringTokenizer st, ThemeData themedata)

	Handles the 'class' tag, which maps a style to a GTK component, and the ultimately a Swing component.
private static boolean	**parseColor**(java.lang.String colorKey, java.util.StringTokenizer st, StyleData styleData)
private static ThemeData	**parseContent**(java.lang.String content) Parses the content passed in as the argument, creates new object all calls initiates the the appropriate methods.
private static boolean	**parseDetail**(java.util.StringTokenizer st, ImageData data) Stores the detail property.
private static boolean	**parseFile**(java.util.StringTokenizer st, ImageData data) Stores the filename.
private static boolean	**parseFont**(java.util.StringTokenizer st, StyleData styleData) Parses the font and stores it in the StyleData object. In turn calls the java.awt.Font.decode() for extracting the correct font.
private static boolean	**parseFunction**(java.util.StringTokenizer st, ImageData data, StyleData styleData) Stores the function.
private static boolean	**parseGap_border**(java.util.StringTokenizer st, ImageData data) Stores the gap border property.
private static boolean	**parseGap_end_border**(java.util.StringTokenizer st, ImageData data)
private static boolean	**parseGap_end_file**(java.util.StringTokenizer st, ImageData data) Stores the gap end file property.
private static boolean	**parseGap_file**(java.util.StringTokenizer st, ImageData data) Stores the gap file property.
private static boolean	**parseGap_side**(java.util.StringTokenizer st, ImageData data) Stores the gap side property.
private static boolean	**parseGap_start_border**(java.util.StringTokenizer st, ImageData data) Stores the gap start border property.
private static boolean	**parseGap_start_file**(java.util.StringTokenizer st, ImageData data) Stores the gap start file property.
private static int[]	**parseGeneralBorder**(java.util.StringTokenizer st) General helper method that parses and stores borders.
private static boolean	**parseImage**(java.util.StringTokenizer st, StyleData styleData) Parses the image statements, in turn calls the appropriate methods to handle the tags within the image statement.
private static boolean	**parseOrientation**(java.util.StringTokenizer st, ImageData data) Stores the orientation property.
private static boolean	**parseOverlay_border**(java.util.StringTokenizer st, ImageData data) Stores the overlay border property.
private static boolean	**parseOverlay_file**(java.util.StringTokenizer st, ImageData data) Stores the overlay file property.
private static	**parseOverlay_stretch**(java.util.StringTokenizer st,

boolean	ImageData data) Stores the overlay stretch property.
private static boolean	**parseRecolorable**(java.util.StringTokenizer st, ImageData data) Stores the recolorable property.
private static boolean	**parseShadow**(java.util.StringTokenizer st, ImageData data) Stores the shadow property.
private static boolean	**parseState**(java.util.StringTokenizer st, ImageData data) Stores the state property.
private static boolean	**parseStretch**(java.util.StringTokenizer st, ImageData data)
private static boolean	**parseStyle**(java.util.StringTokenizer st, ThemeData themeData) Handles the 'style' tag, creating a new style a populating it
static ThemeData	**parseThemeFile**(java.lang.String themeDir) parseThemeFile(String themeDir) Reads and parses the the gtkrc file in the theme's directory, which is the argument. It returns a ThemeData object that contains the information file the file.
private static java.lang.String	**readFile**(java.lang.String file) Reads the file in the argument, processes it to remove tab character, and returns a String with the whole content of the file.

Methods inherited from class java.lang.Object
, clone, equals, finalize, getClass, hashCode, notify, notifyAll, registerNatives, toString, wait, wait, wait

---

com.sun.java.swing.plaf.gtk

Class ImageData

```
java.lang.Object
  |
  +--com.sun.java.swing.plaf.gtk.ImageData
```

---

class **ImageData**

extends java.lang.Object

A class that stores all the properties stated in an image definition. Each property has a counterpart in the gtkrc file.

Field Summary	
java.lang.String	**arrow_direction**
int[]	**border**
static int	**BOTTOM**

java.lang.String	detail
java.lang.String	file
java.lang.String	function
int[]	gap_border
int[]	gap_end_border
java.lang.String	gap_end_file
java.lang.String	gap_file
java.lang.String	gap_side
int[]	gap_start_border
java.lang.String	gap_start_file
boolean	haveArrow_direction
boolean	haveGap_side
boolean	haveOrientation
boolean	haveShadow
boolean	haveState
java.awt.image.BufferedImage	image
static int	LEFT
java.lang.String	orientation
int[]	overlay_border
java.lang.String	overlay_file
java.awt.image.BufferedImage	overlay_image
boolean	overlay_stretch
boolean	recolorable
static int	RIGHT
java.lang.String	shadow
java.lang.String	state
boolean	stretch

StyleData	styleData
static int	TOP

Constructor Summary	
(package private)	ImageData()

Methods inherited from class java.lang.Object
, clone, equals, finalize, getClass, hashCode, notify, notifyAll, registerNatives, toString, wait, wait, wait

com.sun.java.swing.plaf.gtk

Class StyleData

```
java.lang.Object
  |
  +--com.sun.java.swing.plaf.gtk.StyleData
```

class **StyleData**

extends java.lang.Object

StyleData A class that represents a GTK style, hold a number ImageData objects.

Field Summary	
static int	BASE_DISABLED
static int	BASE_NORMAL
static int	BASE_PRESSED
static int	BASE_ROLLOVER
static int	BASE_SELECTED
static int	BG_DISABLED
static int	BG_NORMAL
static int	BG_PRESSED
static int	BG_ROLLOVER
static int	BG_SELECTED
private	colorList

javax.swing.plaf.ColorUIResource[]	
static int	**FG_DISABLED**
static int	**FG_NORMAL**
static int	**FG_PRESSED**
static int	**FG_ROLLOVER**
static int	**FG_SELECTED**
private ImageData	**focusImage**
private javax.swing.plaf.FontUIResource	**font**
java.util.ArrayList	**imageList**
java.lang.String	**name**
static int	**TEXT_DISABLED**
static int	**TEXT_NORMAL**
static int	**TEXT_PRESSED**
static int	**TEXT_ROLLOVER**
static int	**TEXT_SELECTED**
private ThemeData	**themeData**

Constructor Summary

StyleData(ThemeData themeData, java.lang.String name) The arguments are what ThemeData object the style belongs to and what name it should have. If the name is 'default' some default settings are made.

Method Summary

void	**addColor**(int colorIndex, javax.swing.plaf.ColorUIResource color) Adds a color to the style. The argument decides what color is requested. The color code a defined in the StyleData class.
void	**addFont**(javax.swing.plaf.FontUIResource font) Adds a default font to the style
void	**addImageData**(ImageData imageData) Adds an ImageData object to the style
javax.swing.plaf.ColorUIResource	**getColor**(int colorIndex) Returns a specific color from the style. The argument decides what color is requested. The color code a defined in the StyleData class.
ImageData	**getFocusImage**() Returns the style's focus image.

javax.swing.plaf.FontUIResource	getFont() Returns the style's font.
ImageData	getImageData(int index) Returns an ImageData object based on the index in the imageList.
ImageData[]	getImageDataList() Returns the imageList, containing the style's ImageData object.
void	setFocusImage(ImageData imageData) Set what focus image the style uses.

Methods inherited from class java.lang.Object
, clone, equals, finalize, getClass, hashCode, notify, notifyAll, registerNatives, toString, wait, wait, wait

---

com.sun.java.swing.plaf.gtk

Class ThemeData

```
java.lang.Object
  |
  +--com.sun.java.swing.plaf.gtk.ThemeData
```

---

class **ThemeData**

extends java.lang.Object

Represents a GTK Theme. Contains all the StyleData object, and in turn all the ImageData objects belongning to the theme. Also contains helper methods for accessing the contents.

Field Summary	
private StyleData	defaultStyle
java.util.ArrayList	styleList

Constructor Summary	
(package private)	ThemeData()

Method Summary	
void	addStyleData(StyleData styleData) Adds a StyleData object to the theme
void	clearTheme() Clears all the theme's data, i.e. the style list.
StyleData	getDefaultStyle() Returns the default style
StyleData	getStyleData(int index)

	Returns a StyleData object based on its position in the styleList. The argument is the index.
StyleData	**getStyleData**(java.lang.String styleName) Returns the style object, or the deafult one.
StyleData[]	**getStyleDataList**() Returns the style list
static StyleData	**inheretStyle**(StyleData subStyle, StyleData origStyle) Helper method that enables a style to inherit a exsiting style's properties
ImageData	**matchThemeImage**(ImageDataDesc desc, java.lang.String key) Returns the matching ImageData object based on the ImageDataDesc and the component key in the arguments.
ImageData	**matchThemeImage**(java.lang.String state, java.lang.String shadowType, java.lang.String detail, java.lang.String arrowType, java.lang.String orientation, java.lang.String gapSide, java.lang.String function, java.lang.String compKey) Returns the matching ImageData object based on the properties and the component key in the arguments.
void	**setDefaultStyle**(StyleData styleData) Add and sets the default style

Methods inherited from class java.lang.Object
, clone, equals, finalize, getClass, hashCode, notify, notifyAll, registerNatives, toString, wait, wait, wait

---

com.sun.java.swing.plaf.gtk

Class GTKMapper

```
java.lang.Object
  |
  +--com.sun.java.swing.plaf.gtk.GTKMapper
```

---

public class **GTKMapper**

extends java.lang.Object

GTKMapper This class handles the mapping from the gtk components to the Swing components, and also from the Swing components' states to the corresponding ImageDataDesc.

Author:

Fredrik Lagerblad

Field Summary	
private static java.lang.String[]	**compStyleMapTable**
private static java.util.Hashtable	**descMapTable**

private static boolean	**hasInitCompStyleMapTable**
private static boolean	**hasInitDescMapTable**

Constructor Summary
GTKMapper()

Method Summary	
static void	**clearAllTables**() Clears all the tables.
static ImageDataDesc	**getImageDataDesc**(java.lang.String key) Returns the corresponding ImageDataDesc for the argument's component state key.
static java.lang.String[]	**getMapKeys**() Returns an array with all components' states key.
static java.lang.String	**getStyle**(java.lang.String swingCompName) Returns the style that is mapped to the Swing component. If no particular style has been assigned, the default style is returned.
private static void	**initCompStyleMapTable**() Creates and populates the GTK component to Swing component to GTK Style mapping table.
private static void	**initDescMapTable**() Creates and populates the descriptor-component state table
static void	**resetStyleTable**() Resets the component-to-style table.
static void	**setStyle**(java.lang.String gtkCompName, java.lang.String styleName) Sets a style to correspond to a gtk component

Methods inherited from class java.lang.Object
, clone, equals, finalize, getClass, hashCode, notify, notifyAll, registerNatives, toString, wait, wait, wait

---

com.sun.java.swing.plaf.gtk

Class ImageDataDesc

```
java.lang.Object
  |
  +--com.sun.java.swing.plaf.gtk.ImageDataDesc
```

---

class **ImageDataDesc**

extends java.lang.Object

This class is a holder-class for the information used to match and identify a ImageData object.

Field Summary

java.lang.String	**arrow**
java.lang.String	**detail**
java.lang.String	**function**
java.lang.String	**gap**
java.lang.String	**orientation**
java.lang.String	**shadow**
java.lang.String	**state**

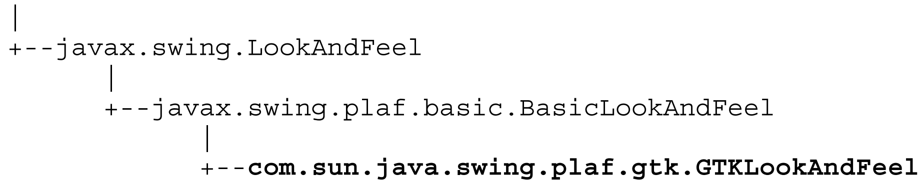
Constructor Summary
ImageDataDesc(java.lang.String state, java.lang.String shadow, java.lang.String detail, java.lang.String arrow, java.lang.String orientation, java.lang.String gap, java.lang.String function)

Methods inherited from class java.lang.Object
, clone, equals, finalize, getClass, hashCode, notify, notifyAll, registerNatives, toString, wait, wait, wait

---

com.sun.java.swing.plaf.gtk

Class GTKLookAndFeel



---

public class **GTKLookAndFeel**

extends javax.swing.plaf.basic.BasicLookAndFeel

The GTK LookAndFeel contains theme specific data and helper methods for accessing the gtk images and data.

Author:

Fredrik Lagerblad

Inner Class Summary	
static class	**GTKLookAndFeel.ProxyLazyValue** This class provides an implementation of LazyValue that can be used to delay loading of the Class for the instance to be created.

Field Summary

private static sun.awt.AppContext	**cachedAppContext**
private static ImageDataDesc	**desc**
private static ImageData	**imageData**
private static java.util.ArrayList	**imageDataList**
private static boolean	**isOnlyOneContext**
private static javax.swing.plaf.FontUIResource	**smallFont**
static ThemeData	**themeData**
protected static java.lang.String	**themeDir**

Fields inherited from class javax.swing.LookAndFeel
modifierKeywords

Constructor Summary

GTKLookAndFeel()

Method Summary

javax.swing.UIDefaults	**getDefaults**()
java.lang.String	**getDescription**()
static javax.swing.plaf.ColorUIResource	**getDisabledBackgroundColor**(java.lang.String key)
static javax.swing.plaf.ColorUIResource	**getDisabledBaseColor**(java.lang.String key)
static javax.swing.plaf.ColorUIResource	**getDisabledForegroundColor**(java.lang.String key)
static javax.swing.plaf.ColorUIResource	**getDisabledTextColor**(java.lang.String key)
static javax.swing.plaf.FontUIResource	**getFont**(java.lang.String key) Helper method to get the font assigned to the component
java.lang.String	**getID**()
static ImageData	**getImageData**(java.lang.String key) Returns an ImageData object based on the argument key. Uses the current theme and the GTKMapper class to retrieve the correct object. Also load its images and cache it.

static ImageData	getImageData2(java.lang.String key) This method below is used as a reference when testing perfromance
java.lang.String	getName()
static javax.swing.plaf.ColorUIResource	getNormalBackgroundColor(java.lang.String key)
static javax.swing.plaf.ColorUIResource	getNormalBaseColor(java.lang.String key)
static javax.swing.plaf.ColorUIResource	getNormalForegroundColor(java.lang.String key) Color helper methods.
static javax.swing.plaf.ColorUIResource	getNormalTextColor(java.lang.String key)
static javax.swing.plaf.ColorUIResource	getPressedBackgroundColor(java.lang.String key)
static javax.swing.plaf.ColorUIResource	getPressedBaseColor(java.lang.String key)
static javax.swing.plaf.ColorUIResource	getPressedForegroundColor(java.lang.String key)
static javax.swing.plaf.ColorUIResource	getPressedTextColor(java.lang.String key)
static javax.swing.plaf.ColorUIResource	getRolloverBackgroundColor(java.lang.String key)
static javax.swing.plaf.ColorUIResource	getRolloverBaseColor(java.lang.String key)
static javax.swing.plaf.ColorUIResource	getRolloverForegroundColor(java.lang.String key)
static javax.swing.plaf.ColorUIResource	getRolloverTextColor(java.lang.String key)
static javax.swing.plaf.ColorUIResource	getSelectedBackgroundColor(java.lang.String key)
static javax.swing.plaf.ColorUIResource	getSelectedBaseColor(java.lang.String key)
static javax.swing.plaf.ColorUIResource	getSelectedForegroundColor(java.lang.String key)
static javax.swing.plaf.ColorUIResource	getSelectedTextColor(java.lang.String key)
static javax.swing.plaf.FontUIResource	getSubTextFont()
protected void	initClassDefaults(javax.swing.UIDefaults table)

	Initialize the uiClassID to BasicComponentUI mapping.
protected void	**initComponentDefaults**(javax.swing.UIDefaults table) Load the SystemColors into the defaults table.
void	**initialize**() Initializes the GTK theme. Creates static resources.
boolean	**isNativeLookAndFeel**()
boolean	**isSupportedLookAndFeel**()
protected static void	**resetImageDataList**() Resets the ImageData list used for caching.
static void	**setCurrentThemeDir**(java.lang.String themeDir) Sets the current theme directory. Checks if it exists and throws and exception if not.
void	**uninitialize**() Unitializes the current theme. Clears and nullifies static variables.

Methods inherited from class javax.swing.plaf.basic.BasicLookAndFeel
initSystemColorDefaults, loadResourceBundle, loadSystemColors

Methods inherited from class javax.swing.LookAndFeel
, installBorder, installColors, installColorsAndFont, makeIcon, makeKeyBindings, parseKeyStroke, toString, uninstallBorder

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, registerNatives, wait, wait, wait

---

com.sun.java.swing.plaf.gtk

Class GTKUtils

```
java.lang.Object
  |
  +--com.sun.java.swing.plaf.gtk.GTKUtils
```

---

class **GTKUtils**

extends java.lang.Object

This class contains helper methods used by many classes. Contains the GTK painting layer, i.e. the methods that know how to fecth, stretch and paint the GTK Images.

Author:

Fredrik Lagerblad

Field Summary	
(package private) static int	**bottom**

protected static javax.swing.JLabel	**comp**
(package private) static int	**holdA**
(package private) static int	**holdB**
private static ImageData	**imageData**
(package private) static int	**imageHeight**
(package private) static int	**imageWidth**
(package private) static int	**left**
(package private) static boolean	**onHold**
(package private) static int	**right**
protected static java.awt.Toolkit	**toolkit**
(package private) static int	**top**
(package private) static java.awt.MediaTracker	**tracker**

Constructor Summary	
(package private)	**GTKUtils**()

Method Summary	
private static void	**_gtkPaint**(javax.swing.JComponent c, java.awt.Graphics g, java.awt.Image image, int[] border, int offX, int offY, int w, int h, boolean paintMiddle) Scales on the fly and paints the images. It uses the border values to paint them correctly.
static java.awt.Image	**createTiledImage**(javax.swing.JComponent c, java.awt.Image oldImage, int newWidth, int newHeight) Stretches, using tiling, the image oldImage to completely cover new size of newWidth and newHeight Mostly used by GTKPanelUI for backgrounds. . !NOTE!When the original image is small performance is bad.
static java.awt.image.BufferedImage	**getScaledImage**(java.lang.String key, int w, int h) Load and stretched the image to the argument dimensions if allowed by the ImageData's stretch property.
static void	**gtkPaint**(javax.swing.JComponent c, java.awt.Graphics g, ImageData id, java.lang.String key, boolean paintShadow, int offX, int offY, int w, int h) Helper method that paints the component using the GTK Images. Check what needs to be drawn, shadow, overlay etc and calls _gtkPaint.
static void	**gtkPaint**(javax.swing.JComponent c, java.awt.Graphics g, java.lang.String key, boolean paintShadow, int offX, int offY, int w, int h) Helper method that paints the component using the GTK Images. Gets the correct

	ImageData object and calls gtkPaint().
(package private) static boolean	**isLeftToRight**(java.awt.Component c)
static java.awt.image.BufferedImage	**loadImage**(java.lang.String filename) LoadImage(Jcomponent, filename) Loads an image from the filename under control of a MediaTracker
(package private) static void	**onlyBorderPaint**(javax.swing.JComponent c, java.awt.Graphics g, java.awt.Image image, int[] border, int offX, int offY, int w, int h) Scales on the fly and paints the images. This method only paints the border, not the middle. Used by borders etc. Takes an image as argument as supposed to the next method.
(package private) static void	**onlyBorderPaint**(javax.swing.JComponent c, java.awt.Graphics g, java.lang.String key, int offX, int offY, int w, int h) Scales on the fly and paints the images. It uses the border values to paint them correctly. . It uses the border values to paint them correctly. This method only paints the border, not the middle. Used by borders etc. Takes a component key string as argument, and loads an image from it.

Methods inherited from class java.lang.Object
, clone, equals, finalize, getClass, hashCode, notify, notifyAll, registerNatives, toString, wait, wait, wait

com.sun.java.swing.plaf.gtk

Class GTKIconFactory.CheckBoxMenuItemIcon

```
java.lang.Object
  |
  +--javax.swing.ImageIcon
        |
        +--com.sun.java.swing.plaf.gtk.GTKIconFactory.CheckBoxMenuItemIcon
```

private static class **GTKIconFactory.CheckBoxMenuItemIcon**

extends javax.swing.ImageIcon

implements javax.swing.plaf.UIResource

An Icon that sense the components state, selected or not, and paints itself according to that.

Field Summary	
private boolean	**isSelected**
private java.awt.image.BufferedImage	**selCheckImage**
private int	**selectedHeight**

private int	selectedWidth
private java.awt.image.BufferedImage	unselCheckImage
private int	unselectedHeight
private int	unselectedWidth

Fields inherited from class javax.swing.ImageIcon
component, description, height, image, imageObserver, loadStatus, tracker, width

Constructor Summary
GTKIconFactory.CheckBoxMenuItemIcon(javax.swing.JMenuItem menuItem)

Method Summary	
int	getIconHeight()
int	getIconWidth()
void	paintIcon(java.awt.Component c, java.awt.Graphics g, int x, int y)

Methods inherited from class javax.swing.ImageIcon
, getDescription, getImage, getImageLoadStatus, getImageObserver, loadImage, readObject, setDescription, setImage, setImageObserver, writeObject

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, registerNatives, toString, wait, wait, wait

---

com.sun.java.swing.plaf.gtk

Class GTKBorders.NewFocusBorder

```
java.lang.Object
  |
  +--javax.swing.border.AbstractBorder
        |
        +--com.sun.java.swing.plaf.gtk.GTKBorders.NewFocusBorder
```

Direct Known Subclasses:

GTKBorders.NewPlainBorder

public static class **GTKBorders.NewFocusBorder**

extends javax.swing.border.AbstractBorder

implements javax.swing.plaf.UIResource

A focus border that only paints itself when the component has focus

Field Summary

private boolean	**alwaysPaint**
private static java.awt.Insets	**borderInsets**
private ImageData	**imageData**
private java.awt.Insets	**insets**

Constructor Summary

GTKBorders.NewFocusBorder(ImageData imageData)

Method Summary

java.awt.Insets	**getBorderInsets**(java.awt.Component c)
void	**paintBorder**(java.awt.Component c, java.awt.Graphics g, int x, int y, int w, int h)
void	**setAlwaysPaint**(boolean alwaysPaint)

Methods inherited from class javax.swing.border.AbstractBorder
getBorderInsets, getInteriorRectangle, getInteriorRectangle, isBorderOpaque

Methods inherited from class java.lang.Object
, clone, equals, finalize, getClass, hashCode, notify, notifyAll, registerNatives, toString, wait, wait, wait

com.sun.java.swing.plaf.gtk

Class GTKButtonUI

```
java.lang.Object
  |
  +--javax.swing.plaf.ComponentUI
        |
        +--javax.swing.plaf.ButtonUI
              |
              +--javax.swing.plaf.basic.BasicButtonUI
                    |
                    +--com.sun.java.swing.plaf.gtk.GTKButtonUI
```

Direct Known Subclasses:

GTKToggleButtonUI

---

public class **GTKButtonUI**

extends javax.swing.plaf.basic.BasicButtonUI

GTKButtonUI implementation

Author:

Fredrik Lagerblad

Field Summary

private boolean	**defaults_initialized**
protected java.awt.Color	**disabledTextColor**
protected boolean	**hasInitToolBarButton**
protected boolean	**installBorder**
protected boolean	**isToolBarButton**
protected java.awt.Color	**pressedTextColor**
protected java.lang.String	**propertyPrefix**
protected java.awt.Color	**rolloverTextColor**
protected java.awt.Color	**selectColor**
protected boolean	**useFocusImages**

Fields inherited from class javax.swing.plaf.basic.BasicButtonUI
buttonUI, defaults_initialized, defaultTextIconGap, defaultTextShiftOffset, iconRect, propertyPrefix, shiftOffset, textRect, viewRect

Constructor Summary

GTKButtonUI()

Method Summary	
protected javax.swing.plaf.basic.BasicButtonListener	**createButtonListener**(javax.swing.AbstractButton b) Creates a FocusListener which forces a repaint.
static javax.swing.plaf.ComponentUI	**createUI**(javax.swing.JComponent c)
protected java.awt.Color	**getDisabledTextColor**()
protected java.awt.Color	**getPressedTextColor**()
protected java.lang.String	**getPropertyPrefix**()
protected java.awt.Color	**getRolloverTextColor**()
protected java.awt.Color	**getSelectColor**()
private void	**initToolBarButton**(javax.swing.AbstractButton b) Initializes certain toolbar-button specific properties, if the button is a ToolBar button.
protected void	**installDefaults**(javax.swing.AbstractButton b) Install defaults as borders, textcolor, insets etc
void	**paint**(java.awt.Graphics g, javax.swing.JComponent c) Paints the button, queries the model for its state and delegates the painting to GTKUtils.gtkPaint().
protected void	**paintButtonPressed**(java.awt.Graphics g, javax.swing.AbstractButton b) Paints the button when its pressed.
protected void	**paintText**(java.awt.Graphics g, javax.swing.JComponent c, java.awt.Rectangle textRect, java.lang.String text) PAints the button's text
protected void	**uninstallDefaults**(javax.swing.AbstractButton b)
private void	**uninstallImages**(javax.swing.AbstractButton b)

Methods inherited from class javax.swing.plaf.basic.BasicButtonUI
, clearTextShiftOffset, getDefaultTextIconGap, getMaximumSize, getMinimumSize, getPreferredSize, getTextShiftOffset, installKeyboardActions, installListeners, installUI, paintFocus, paintIcon, setTextShiftOffset, uninstallKeyboardActions, uninstallListeners, uninstallUI

Methods inherited from class javax.swing.plaf.ComponentUI
contains, getAccessibleChild, getAccessibleChildrenCount, update

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, registerNatives, toString, wait, wait, wait

II. Code for the test program

The tests that were used a standard GUI application with a common set of components. Some things that were tested used the tryMethod – method, where a certain method could be timed. Other finer-grained tests had to be done within the package.

Here is the code for the application used:

```
import java.io.*;
import java.awt.*;
import javax.swing.*;
import com.sun.java.swing.plaf.gtk.*;
import java.awt.event.*;

public class GTKTest extends JFrame {

  //Declare components
    JMenuBar menuBar;
    JMenu menu1;
    JMenu menu2;
    JMenu submenu;
    JMenuItem item1;
    JMenuItem item2;
    JMenuItem item3;
    JMenuItem item4;
    JMenuItem item5;
    JMenuItem item6;

    JRadioButtonMenuItem rbitem1;
    JRadioButtonMenuItem rbitem2;
    ButtonGroup buttonGroup;
    JCheckBoxMenuItem cbitem;

    JButton button1;
    JTextField textField;
    JComboBox comboBox;
    JComboBox systemComboBox;
    JToolBar toolBar;
    JList list;

    JRadioButton rb1;
    JRadioButton rb2;
    ButtonGroup buttonGroup2;
    JCheckBox cb1;
    JProgressBar progressBar;
    JTextArea textArea;

  public GTKTest() {

    this.setTitle("GTK Test");
    this.setSize(600, 400);
    this.getContentPane().setLayout(new FlowLayout() );
    this.addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
    });

    //Create components
    item1 = new JMenuItem("Open...");
    item2 = new JMenuItem("Close");
    item4 = new JMenuItem("Slussen");
    item5 = new JMenuItem("Maria Torget");

    submenu = new JMenu("SubMenu");
    submenu.add(item4);
    submenu.add(item5);

    item3 = new JMenuItem("Exit");

    menu1 = new JMenu("File");
    menu1.add(item1);
```

```
menu1.add(item2);
menu1.add(submenu);
menu1.add(item3);



item6 = new JMenuItem("Debug");

rbitem1 = new JRadioButtonMenuItem("Kiss FM");
rbitem2 = new JRadioButtonMenuItem("KJCM");

buttonGroup = new ButtonGroup();
buttonGroup.add(rbitem1);
buttonGroup.add(rbitem2);

cbitem = new JCheckBoxMenuItem("Check!");
menu2 = new JMenu("Extra");
menu2.add(item6);
menu2.add(rbitem1);
menu2.add(rbitem2);
menu2.add(cbitem);

menuBar = new JMenuBar();
menuBar.add(menu1);
menuBar.add(menu2);
this.setJMenuBar(menuBar);

button1 = new JButton("Try it!");
button1.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    tryMethod();
  }
});

textField = new JTextField(10);
textField.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    JTextField source = (JTextField) e.getSource();
    getAndSetLook( source.getText() );
    source.setText("");

  }
});

//init comboBox
String[] themes = getThemes();
comboBox = new JComboBox(themes);
comboBox.setEditable(false);
            comboBox.addActionListener( new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    getAndSetLook((String) ((JComboBox) e.getSource()).getSelectedItem() );
              }});

//init system comboBox
String[] systemThemes = {"metal", "motif", "windows"};
systemComboBox = new JComboBox(systemThemes);
systemComboBox.setEditable(false);
            systemComboBox.addActionListener( new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    setSystemLook((String) ((JComboBox) e.getSource()).getSelectedItem() );
              }});

toolBar = new JToolBar();
toolBar = new JToolBar();
            toolBar.add(new JButton("left",new ImageIcon("left.gif")));
            toolBar.add(new JToolBar.Separator() );
            toolBar.add(new JButton("middle",new ImageIcon("middle.gif")));
            toolBar.add(new JButton("right",new ImageIcon("right.gif")));

String[] listString = new String[] {"One", "Two", "Three", "Thirtytwo"};
            list = new JList(listString);

rb1 = new JRadioButton("Choose...");
rb2 = new JRadioButton("...one!");
```

```
  buttonGroup2 = new ButtonGroup();
  buttonGroup2.add(rb1);
  buttonGroup2.add(rb2);
  JPanel panel = new JPanel();
  panel.setLayout(new GridLayout(0, 1));
  panel.add(rb1);
  panel.add(rb2);

  cb1 = new JCheckBox("Check Mate!");

  progressBar = new JProgressBar();
  progressBar.setStringPainted(true);
  progressBar.setValue(30);

  textArea = new JTextArea(10,10);

  this.getContentPane().add(button1);
  this.getContentPane().add(textField);
  this.getContentPane().add(comboBox);
  this.getContentPane().add(systemComboBox);
  this.getContentPane().add(toolBar);
  this.getContentPane().add(list);
  this.getContentPane().add(panel);
  this.getContentPane().add(cb1);
  this.getContentPane().add(progressBar);
  this.getContentPane().add(new JScrollPane(textArea));
}

 public void getAndSetLook(String name) {

  String lnfName = "com.sun.java.swing.plaf.gtk.GTKLookAndFeel";
             try{
     com.sun.java.swing.plaf.gtk.GTKLookAndFeel.setCurrentThemeDir("themes/"+name);
                UIManager.setLookAndFeel(lnfName);
                SwingUtilities.updateComponentTreeUI(this);
             }
  catch (Exception e) {
               System.out.println("Error creating theme: "+ e);
             }

 }

 public void setSystemLook(String name) {
  String lnfName = "";
             if( name.equals("metal") )
               lnfName = "javax.swing.plaf.metal.MetalLookAndFeel";
             else if( name.equals("motif"))
               lnfName = "com.sun.java.swing.plaf.motif.MotifLookAndFeel";
             else if( name.equals("windows"))
                lnfName = "com.sun.java.swing.plaf.windows.WindowsLookAndFeel";

             try{
               UIManager.setLookAndFeel(lnfName);
               SwingUtilities.updateComponentTreeUI(this);
             }
             catch (Exception e) {
               e.printStackTrace();
             }
 }

 protected String[] getThemes() {
             String dirName = "themes";
             File dir = new File(dirName);
             String[] all = dir.list();

             int count = 0;
             for(int i= 0; i < all.length; i++) {
               if( (new File(dirName + "/"+all[i])).isDirectory() )
                           count++;
             }
             String[] dirNames = new String[count];

             int j =0;
             for(int i =0; i < all.length; i++) {
               if( (new File(dirName + "/"+ all[i])).isDirectory() ) {
```

```
                              dirNames[j] = all[i];
                              j++;
                 }
               }
               return dirNames;
  }

  public void tryMethod() {
    long time1 = System.currentTimeMillis();
    for(int i=0; i <50000; i++) {
      //Use method needed to be tested stated here.
      GTKLookAndFeel.getImageData("Button.normalImage");
      }
    System.out.println("Time:" + (System.currentTimeMillis() - time1));

  }

  public static void main(String[] args) {
    GTKTest gtkTest = new GTKTest();
    gtkTest.setVisible(true);
    gtkTest.invokedStandalone = true;
  }
  private boolean invokedStandalone = false;
}
```

III. Glossary

LAF	Look And Feel – a front-end user interface for an application that decides the look and the behavior.
PLAF	Pluggable Look And Feel – a LAF that can be “plugged” in and replace another LAF.
GUI	Graphical User Interface – a graphical interface which represents the applications state to the user.
AWT	Abstract Window Toolkit – Java’s older GUI toolkit, using peer-to-peer technology.
JFC	Java Foundation Classes – a set of core Java APIs consisting of AWT, Swing, Accessibility, Java 2D and Drag’n Drop.
MVC	Model-View-Controller – a technique to build GUI components, which divides the components in three parts, model, view and controller.
GDK	GNU Drawing Kit – a Linux low-level screen drawing kit.
GTK	Gimp Toolkit – a set of Linux GUI components.
GIMP	GNU Image Manipulation Program – a powerful image manipulation program for Linux
PNG	Portable Network Graphic – an image format developed as a free alternative to GIF.

IV. GTK Color Representation

The colors are coded by using a prefix and combining them with a state.

The prefixes

fg	Used for the text color.
bg	Used for backgrounds when no image exists.
base	Used for backgrounds of some components with texts, e.g. textfields and lists.
text	Used for the text color of text components, e.g. textfields.

The states

NORMAL	Used when the component is in a normal state.
PRELIGHT	Used when the component is in a rollover state.
ACTIVE	Used when the component is in an active state, e.g. a button is pressed.
SELECTED	Used when the component is in a selected state.
INSENSITIVE	Used when the component is in a disabled state.

Examples

Color code	*Example of usage*
fg[ROLLOVER]	The text color of a button when the mouse is in rollover state.
bg[NORMAL]	Used for the background of a panel, if no image exists for it.
base[SELECTED]	The background color of a textarea when it is selected.
text[DISABLED]	The text color of a textfield when it is disabled.