Charles University in Prague
Faculty of Mathematics and Physics

MASTER'S THESIS

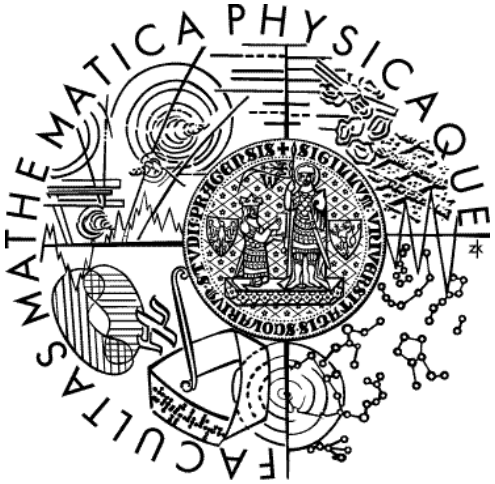
Jan Tauš

Visual editor for creating of Swing components

2007 Jan Tauš

Charles University in Prague
Faculty of Mathematics and Physics

MASTER'S THESIS



Jan Tauš

Visual editor for creating of Swing components

Department of Software Engineering
Supervisor: Dušan Pavlica
Study Program: Computer Science, Software Engineering

I hereby declare that the thesis is a work of my own and that only cited sources have been used. I permit lending of this thesis.

Praha, 27.4. 2007 Jan Tauš

Table of Contents

Název práce: Visual editor for creating of Swing components

Autor: Jan Tauš

Katedra: Katedra softwarového inženýrství

Vedoucí diplomové práce: Dušan Pavlica, Sun Microsystems, Inc.

e-mail vedoucího: dusan.pavlica@sun.com

Abstrakt: Současná implementace knihovny Swing – standardního uživatelského grafického rozhraní Javy – umožňuje jen omezené možnosti modifikace vzhledu ovládacích prvků. V této práci hledám mechanizmus jak rozšířit tyto možnosti s využitím zásuvných témat Look and Feel. Také analyzuji různé programy umožňující kompletně změnit vzhled jednoduchým způsobem zvaným skinování. Snažím se aplikovat vzory nalezené u skinovatelných programů na nově vytvořené Look and Feel téma. Na závěr popisuji nový modul do editor NetBeans, který umožňuje editovat navržené téma.

Klíčová slova: Swing, NetBeans, grafický návrh, skin

Title: Visual editor for creating of Swing components

Author: Jan Tauš

Department:Department of Software Engineering

Supervisor: Dušan Pavlica, Sun Microsystems, Inc.

Supervisor's e-mail address: dusan.pavlica@sun.com

Abstract: Current implementation of Swing library – standard graphical user interface of Java – enables only limited means of modification of visual components. In this paper, I'm looking for a mechanism how to enrich these means using pluggable Look and Feel. In addition I analyze various programs which allow to completely change their look using very comfortable technique called skinning. I try to apply patterns used by skinned programs to a new Look and Feel theme. I am creating framework for editing of this theme. At the end I describe newly created NetBeans Module which enables to edit the new Look and Feel theme.

Keywords: Swing, NetBeans, graphical design, skin

Chapter 1
Introduction

When a programmers create a Java application with graphical user interface, they will probably use Swing graphical components which are a standard part of Java platform. They can write code which construct dialogs and windows themselves, or they can use some GUI designer like the Form designer in NetBeans Java Editor.

Sometime developers need to change appearance of components. They want to make the text bold or different color. Or they want to set the background color of their application to blue. But often the developers want to completely alter appearance of some control components of their application.

For such purposes each swing component provides properties which modify painting of a component. However, there are few problems with these properties:

- Only limited set of properties describing appearance of component are available for each component. You can set the background color of panel, but you cannot set its background image. You can set the button color, but you cannot set color of a pressed button.
- Developers must set these properties for each component separately.
- Sometimes the designer of the user interface is not the same person as the programmer. But there is no standard way how to pack these properties into some template which the programmer can apply as a whole to components.

1.1. Goal

The goal of this thesis is to find a solution for the presented problems. First part will analyze the problem. I will find the way how to change appearance of a component, how to do it simple and how to allow the designer to define such appearance in GUI.

In the next part of the thesis, I implement a library to show that proposed solution can be implemented, and I create a visual editor of the components look in NetBeans Platform.

1.2. Structure of the work

This study is divided in following chapters:

- Chapter 1: Introduction
  This chapter
- Chapter 2: Skins

Here I investigate some programs which allow to define they appearance in an easy way. I concentrate on the concept known as the skin. I list some patterns used in skinned programs and show how these patterns can be used for designing a Java component.

- Chapter 3: Java Look and Feel
  In this chapter, I describe the possibilities how to change appearance of a swing component and I discus their advantages and disadvantages. In addition, I study current state of pluggable L&F interface of Java platform.
- Chapter 4: Skin Theme Implementation
  Findings from previous two chapters result into the creation of a set of classes which can be used to define new look for swing components and then simply apply this look on these components. At the end, I describe a test application which uses these classes to design appearance of some component.
- Chapter 5: ThemeEditor Module
  I bring implemented libraries together with designer into NetBeans platform using NetBeans Module.
- Chapter 6: Conclusion
  Final conclusion.

1.3. Glossary

Thorough whole text I use some specific terms which are defined here:

- Swing
  Standard library for creating GUI in Java. Part of Java platform. Based on older GUI library AWT.
- Swing component, graphical component
  Basic building block of Swing GUI. Class extending JComponent. In this work, I exclusively address altering of appearance of Swing Components.
- JavaBeans, JavaBeans pattern
  Java doesn't support natively object properties as Object Pascal from Borland or C# from Microsoft do. Still properties can be simulated using reflection and special naming convention of object methods. See JavaBeans Specification .
- Current Java, current Java platform
  I mean Java 6.0 newest stable Java platform in time of writing this work

Skins

In introduction, when I wrote about the different role between designer of the application and the programmer, my point was the well known fact that programmers are not always good graphic designers and vice versa. The need of graphic designer for a nice GUI arises from the first part of this statement, and the need of tools which allow to define the look of some program and which are not based on programming arise from second part of this statement.

In Java there exist L&F themes which offer the designer a great opportunity to simple modify their appearance, but often you identify which one is used. I will write about them in next chapter. Now I concentrate on finding a way how to define the appearance of some GUI component with the greatest possible freedom, but still allow defining such appearance in visual editor without the need of writing Java code.

1.4. What is a Skin

Many programs allow to change their look by providing a set of images. Even non-programmers are able to create such images in graphic editors. These sets of images are often called *Skins*.

From Wikipedia: *In computing, skins and themes are custom graphical appearances (GUIs) that can be applied to certain software and websites in order to suit the different tastes of different users. Such software is referred to as being skinnable, and the process of writing or applying such a skin is known as skinning. Applying a skin changes a piece of software's look and feel - some skins merely make the program more aesthetically pleasing, but others can rearrange elements of the interface, potentially making the program easier to use.*

The term Skin is in fact a synonym for a Theme. But often (and the programs I will present are an evidence of this fact) skins are made only from small set of images and a few text files with other resources (font, positions). The main goal is simplicity of their creation and the amount of available skins for such programs is often huge.

1.4.1. WinAmp

This is a popular audio player for Microsoft Windows OS. WinAmp skins are simple to create and they are widely popular. A huge repository of skins is available on WinAmp home page http://www.winamp.com (see Figure 1.1).

Figure 1.1: WinAmp main window with various skin applied

WinAmp is so popular for playing audio on Windows platform that Linux audio player XMMS was created by its example. XMMS support even WinAmp skins.

1.4.2. Metacity Windows manager

Metacity is a window manager for Unix X Windows graphic environment. It was developed and used for GNOME desktop. One of the windows manager's tasks is to paint borders of the windows (see Figure 1.2).

Figure 1.2: Metacity manager with various themes

1.4.3. QuickTime Movie Player Skins

Quick Time Movie player is an Apple application which can play video files and streams. A QuickTime movie can contain a skin – the Information which defines the look of the. Although this skin definition is very simple in comparison with previous two examples, it can incorporate Flash components controls and the Flash technology enables the QuickTime movie player skin to be very powerful and variable.

Figure 1.3: QuickTime Movie Player with applied skin

1.5. Skin Patterns

With the list of example applications I will start describes what have these application themes in common. In my opinion this constitutes the term skin.

1.5.1. Application is Image

Rather then defining buttons and other control components and layout for them, skinned programs just display an image. The whole window of such program is an image and the program just knows where each button lies. When click event is generated, the program decides whether the cursor is over a button and which appropriate action is to be performed.

Some skins consist of multiple images. Some images are application window background and others are control components like buttons. There is another file (usually simple text file or XML file) which defines image and position for each control component.

1.5.2. Each State of Component is Image

Rather then trying to modify the painting of a component in its various states (e. g.: draw disabled button in gray scale, or shift button when it is clicked), the skin defines images for all states of each component. It depends on particular component implementation which states it supports.

Example: Buttons in Figure 1.4 have only two states: normal and pressed.

Figure 1.4: Image of three buttons in normal/pressed state

Example: In the following 2 images, we can see skinned equalizer windows and observe the difference between the scrollbars backgrounds. One changes color according to the position while the other just fills a part of the scrollbar below pointer.

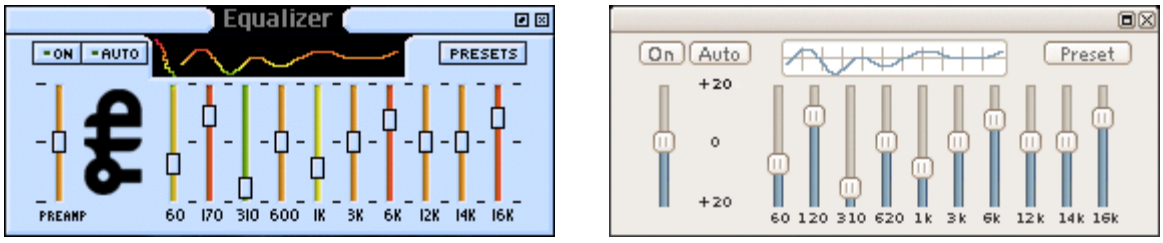


Figure 1.5: Scrollbars in equalizer window of WinAmp

Rather than to define some complex logic in scrollbar painter and then in a much more complicated way configure it, WinAmp simply paints background using set of images defined in them:

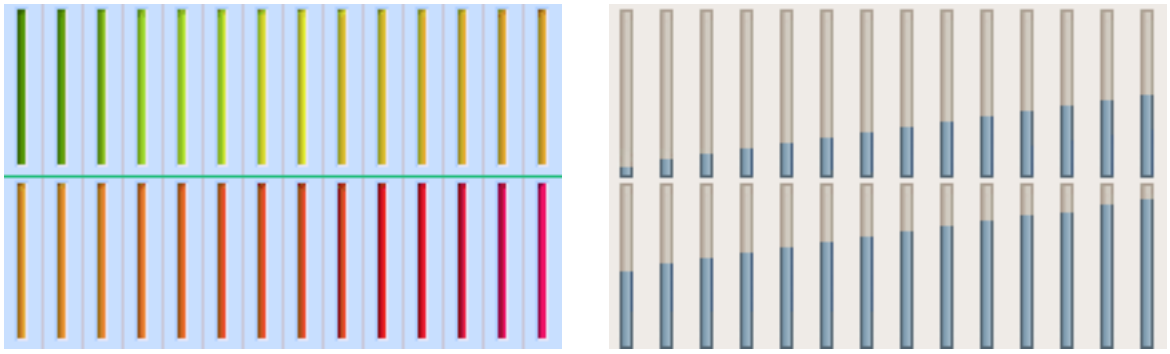


Figure 1.6: Scrollbar background definition in WinAmp skin

It is important to note that this pattern can be used only when component has a limited set of states and its dimension is also limited. Resizable element pattern below can resolve the limit of dimensions, but does not solve the limit of number of states .

1.5.3. Images are Packed in Larger Images

The fact that various states of the components of skinned application have their own image can lead to huge amount of images in one skin. In addition, in many image editors working with many images is not comfortable. Its better to group all images in to few larger files. Images are usually grouped thematically (see Figure 1.7).

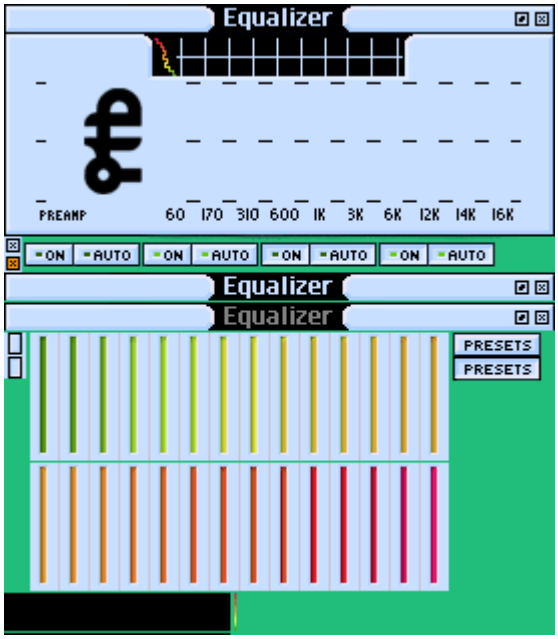


Figure 1.7: Full definition of WinAmp equalizer window in one image

1.5.4. Resizing Elements

Some elements of skinned application may change their size. It is not feasible to prepare an image for each size of each element. We would have to prepare an extremely large amount of images and still there would remain some sizes which our skin would not support.

I will demonstrate how to deal with this limit on a theme used in GNOME Metacity Windows Manager. In theme SphereCrystal, the top border of application window paints this way:

The icon on the left is the application icon. Name of the window is painted using font defined in theme's configuration file and the icons on the right come from the image file(s) which is a part of a theme. We shall focus on the background of the border, which is defined in three separate files:

The side images are tiled according to the width of the application window and the middle image is always painted at the fixed size.

1.5.5. Masks

QuickTime media player skin[QTT] consists of three images. One is the image of the player and the two others are black-white images called masks. The masks have the same size as the player image. The black pixels in the mask define a particular area. One mask called shape mask defines the shape of the window, and the other mask called drag mask defines that part of the window which can be handled with cursor to move the player window .

I will use the drag mask in my skin theme. But I will omit the shape mask. This one bit per pixel information can be stored in the image alpha channel and in the case of swing components we can even use partially translucent components[1].

1.5.6. Fonts

WinAmp skin contains images of individual characters which are used for example for rendering of play time displayed in WinAmp main window.:

Such maps of characters can be used only when we now the whole set of possibly displayed characters (numbers in this case). I will not use this pattern in SkinTheme because Swing components display Strings which usually contain localized text with unpredictable various characters. It is almost impossible for a skin designer to add all characters into the skin, and In addition I believe that it does not add to the functionality. For this purpose, designer would probably rather change the font of the application.

1.5.7. Color Maps

The equalizer graph is a good example of how list of colors can be defined in the image file. Look at the Figure 1.8. You can see thin rainbow next to the big black rectangle at the bottom part. This color map is used for painting equalizer graph in window presented earlier in Figure 1.5.

Figure 1.8: Color set defined in skin

Color palette can be defined in an image. This is used especially at the application skins to reduce the information which the user must write into description files. Because SkinTheme contains graphical configurator, it is better to add definition of the palette inside the configuration and provide an editor for this rather than force the designer to create additional files with some very abstract content.

1 QuickTime shape mask defines shape of the window because alpha channel of image is used for determination where you will see the movie. Also window mask is one bit because support for partially translucent windows is even today not common

1.5.8. Animations

Some component of skinned application can change their appearance with time rather than on user input. Thus they are in fact animated. Animation consists of set of frames. Some image formats like GIF have a direct support of animated images – built of frames. If we want to use some other format, we can simply place animated images into one main image. Additional information regarding animation can be saved in a configuration file.

1.5.9. Scripts

For greater flexibility skins can contain scripts. We can divide these scripts to two categories:

- painting scripts – these scripts are evaluated only when component is painted. They contain a list of painting operations. The logic which defines the state of the component and handles user input is included in the program. Such scripts are used in Gtk 2.0 themes and produce very complex designs.
- interactive scripts – these scripts can react directly based on the user input and then change components state. Painting scripts are their part.

The main advantage of scripts is in in situation when skinned application is distributed as a compiled binary independently on skin. In this common scenario, skin designer can change the look and even behavior of the components without manipulating the source code.

In the implementation part, I will not work with scripts, because one of the intentions of this paper is to provide a way to customize component look inside Java code. When you have access to Java code you can program any logic directly in to the program so that programming this on other place and possibly in other language will often not be of an advantage (see next section).

Note: Java 6.0 Scripting greatly helps in interaction between Java and scripting languages. Thus a theme could actually be developed that would put together skins and scripts without too much additional effort.

1.5.9.1. More Advanced Techniques

In this paragraph, I'd like to mention one other technique that can be used for skinning.

Skin of a Quick Time movie can use objects defined in Flash as a control element. Flash objects contain scripts to define their behavior and there are advanced editors for them. The Movie Player then only needs to provide some control object for Flash scripts. All the component logic is defined outside the player. This way player can be smaller and simpler while leaving the full creative freedom to the developer of the skin.

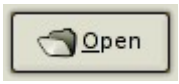
There is another technology witch is intended to partially replace Flash. SVG has public specification and its support rapidly grows. Linux desktop GNOME supports SVG images natively and lots of its themes already use scalable SVG icons. In Java world there are projects bringing SVG to Java like Batik SVG Toolkit[Batik].

Same conclusions as for the scripts apply for using scripted SVG . But the existence of editors for SVG with lots of feature actually justify the existence of parallel code in swing component's Java code and scripted SVG.

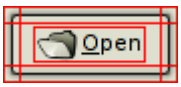
1.6. Object Resizing

In my opinion, what makes images suitable for skinning components even if we don't yet know their position and dimension is Resizing Element pattern(see 1.5.4 above). The main window of WinAmp always has s fixed size because we know all of its content in advance. But when we want to use images for painting swing component without further knowledge of its future content, we must pay special attention to appropriate modification of these images.

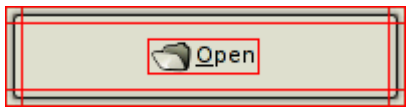
Now I will analyze possibilities of how to reuse and improve the procedure described in Resizing Element patter. I took button as an example component. We can use an image as a background of it. But as I stated above, we do not know the button's text in advance so we must somehow resize its background image on the fly. If we look at the button painted by my favorite Gtk theme Gorilla,



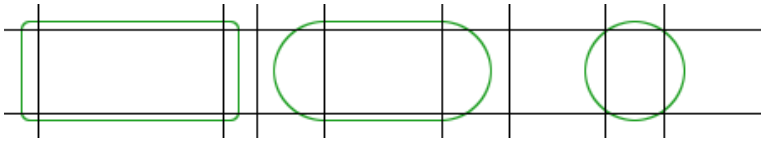
we can see that it is simply divided in border part and center;



and then if we resize it, we can scale one part of it and leave the other parts unmodified.



Of course we cannot use this resizing scheme for any component. For example different shapes of buttons



will be deformed, if resized this way.

We can see that some other division can be useful in such cases. Lots of such divisions can be made, if required.

I will call these divisions canvas and the parts of canvas I will call tiles. Then we may ask following questions:

- How to define the structure of a canvas?
- How to scale the tiles?
- Can a tile be something other than image?

1.6.1. Canvas Definition

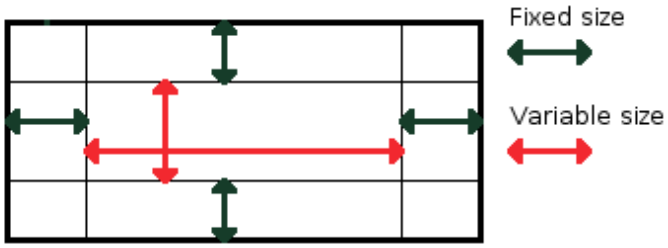
The main responsibility of canvas is to resize and place tiles into proper place according to canvas' own dimension. Many schemes of such placings can be created. I will focus on one such scheme, which can be easily defined and modified.

1.6.1.1. Grid Canvas

Grid canvas works similar to GridBagLayout in Java. It defines columns and rows, and their resizing modes. Canvas looks like a table[2]. The column or row respectively can have one of two modes:

- fixed size
  this columns/rows size doesn't change,
- variable size
  this column/row fills remaining space, there can be additional value to this mode called weight. The remaining space after fixed sizes are subtracted from the size of the whole canvas is the distribute in proportion to this weights.

Tiles are then placed in the cells of this table, and some tiles can span multiple cells, but they must remain rectangle shaped. For example grid canvas for button can be defined this way:



The grid canvas enable us to define two properties which are useful when working with visual component in Java:

- minimum dimension
  Minimum with is a sum of all withs of fixed components and minimum height is a sum of widths of all fixed rows.
- maximum dimension
  Maximum with is a sum of all fixed columns in case there are only fixed columns in canvas otherwise with is undefined or infinite (usually Integer.MAX_VALUE in Java). Maximum height is computed the same way from canvas rows.

1.6.2. Resizing of Images

The problem is following. We defined some image which has fixed size yet we want to fill with it greater/smaller rectangle. When images are used for filling some space usually they are:

- left at the same dimension
  The remaining space is filled with some predefined background color.
- scaled
  Image is stretched across the whole rectangle. This usually degrades the image.
- tiled
  Image is repeatedly painted at original size to fill whole space.

We can apply these modes of filling in both direction independently (see Figure 1.9).

2 Tiles are analogous to components in GridBagLayout. Resizing modes of rows and columns are defined by layout constraints of components which can be ambiguous. I allow to define only resizing mode of rows and columns. Tiles just contains only information about their placement.

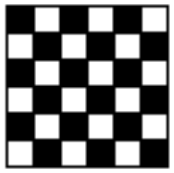
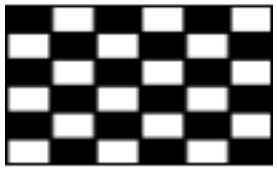
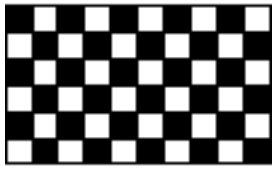
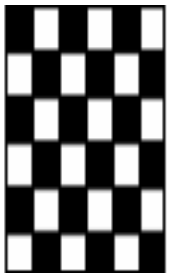
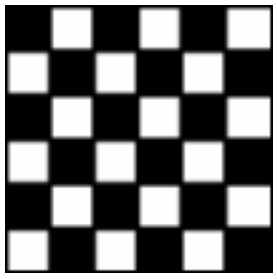
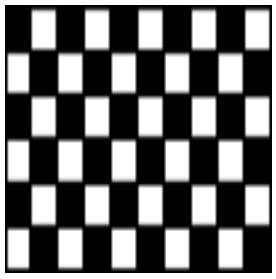
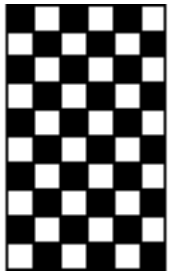
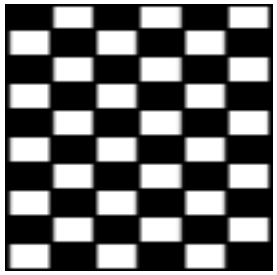
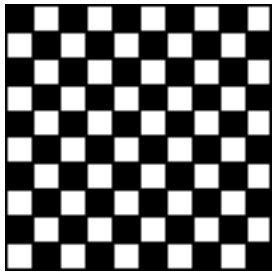
	fixed	scaled	tiled
fixed			
scaled			
tiled			

Figure 1.9: Example how fill mode affect the image

Anchor describing the rectangle side from which to start can be added to these fill modes (see Figure 1.10). Anchor has no meaning in case of a scaled image.

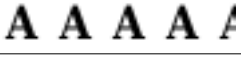
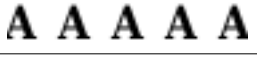
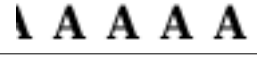
anchor value	0.0	0.5	1.0
fixed			
tiles			

Figure 1.10: Example how anchor affects the image

1.6.3. Tiles

Sometimes one tile can be filled with one color. Then its better to define this tile only with color than creating one-color-one-dot image and resizing it to tiles dimension (as is sometimes seen on web pages). Fill area with color is much more resource save as opposed to a resized image.

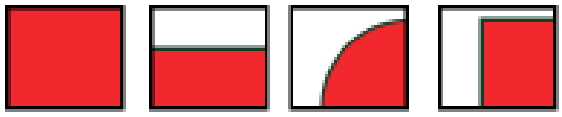
I will now propose examples of three types of tiles additional to image tile described earlier. Naturally, unlimited amount of tile types can be created.

1.6.3.1. Color Tile

As stated above, this tile is defined by only one color and this color always fills the whole destination rectangle.

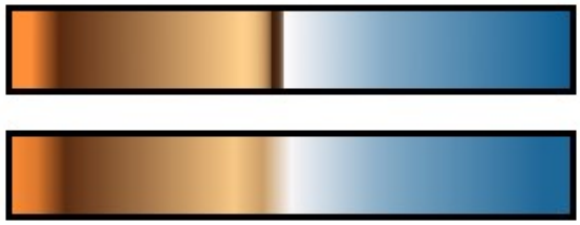
1.6.3.2. Shape Tile

Sometimes a component can consist only of simple colors and patterns and they can be rendered easily by means of Java libraries. Wee can look at them as a vector images. Their best property is that they do not degrade if they are resized. Shape tiles can be used for definition of components borders:

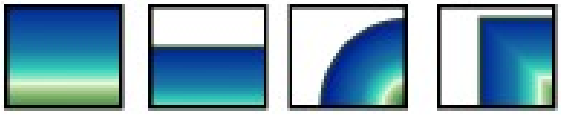


1.6.3.3. Gradient Tile

Gradient tiles are shape tiles but they are filled with a color gradient. There are always better results when you are resizing vector graphics then bitmap ones. Compare rendered gradient on top and same gradient resized from smaller image(bottom).



Gradient tiles can be also shaped in a similar way like shape tiles:

: 

1.6.4. Tiles Defined by Image

Image tiles can be defined in one master image. Then we need to define the position at which the tile image lies. For this, we can use the same canvas which we use for placing the tiles. We can resize canvas to the size of master image and positions, and dimension of tiles defines their sub images in the master image.

This approach is very easy for a designer of the skin. The designer just paints a component and then chooses the proper canvas and adjusts the size .

Chapter 2
Java Look and Feel

The part of Java which handles rendering of the components is called Java Look and Feel. In the following text, I will simply describe the process of rendering visual components and I will demonstrate how this process can be altered. I will also show some problems which arise from design of L&F which one must have in mind when working with it. My analysis is based on various documents about Java Look And Feel, but the main source of information presented below i Java source code of the current Java platform. In this text, I also often mention standard, existing or base Look And Feel. I mean basic L&F and Metal L&F which code I was analyze by all of this terms.

2.1. Existing Look and Feel Themes

2.1.1. Gtk Theme

Gtk theme which mimics Look and Feel of the Gtk+ graphical toolkit. Java applications with this L&F fits to GNOME desktop. In Java 5.0 platform this theme takes configuration from system. Look of the Gtk+ toolkit is extensively configurable. It uses binary plug-ins to draw some components. Such plug-ins are called engines. The Gtk+ theme describes which component is drawn by which engine and provides other resources like icons and colors. These resources are also used by Java Gtk L&F. But there is problem with engines. There are implemented two ways how to solve this problem:

- Java 5.0 platform - only default engine is implemented. Other engines are ignored. Java application still fits to desktop but discrepancies are visible at the first sight.
- Java 6.0 platform - Java components are drawn by Gtk+ library itself on the off screen image. This image is than used to paint swing component on the screen. Components rendered this way are very authentic but it greatly reduce application performance.

2.1.2. Windows XP Theme

Application using Windows XP L&F fits perfectly to the Windows XP desktop. Windows XP user interface didn't change since release in 2001. User interface is not very configurable. With standard tools (Properties of Desktop), users can do only minor changes. They can change color schema and used fonts. For this reasons current Java application with Windows XP L&F is almost undistinguished from the native application.

2.1.3. Metal Theme

This is the default cross-platform theme. If the programmer decides to use this theme in his application. Application will look same everywhere. This theme is very conservative. Java application using Metal L&F looks oldish in comparison with native applications. Metal L&F has on the other side one feature, which enables easy changes of its appearance. Metal L&F can be configured by a metal theme, which is represented by class MetalTheme. Designer can subclass this theme and change fonts of colormap and then register his theme to the Metal L&F.

2.1.4. Basic Look and Feel

This theme can be used as a standalone theme but its main purpose is to be place for common code of other themes contained in Java Platform. It seems that basic theme will be good base for other themes. At the end of this chapter I will show that it is not true and that subclassing basic themes brings many problems to the new theme,

2.1.5. Substance

Substance[Subst] is cross platform L&F which is not standard part of Java platform. It use same pattern as the Metal L&F. SubstanceTheme contains set of colors and fonts used for painting components. But Substance L&F use these colors in more appealing way then the Metal L&F. Substance L&F also enables to set images (called watermarks) as a background of the components and change shape of the button.

Substance L&F is relatively new and supports only Java 5.0. On the other site its in the production quality and it can be easily configured.

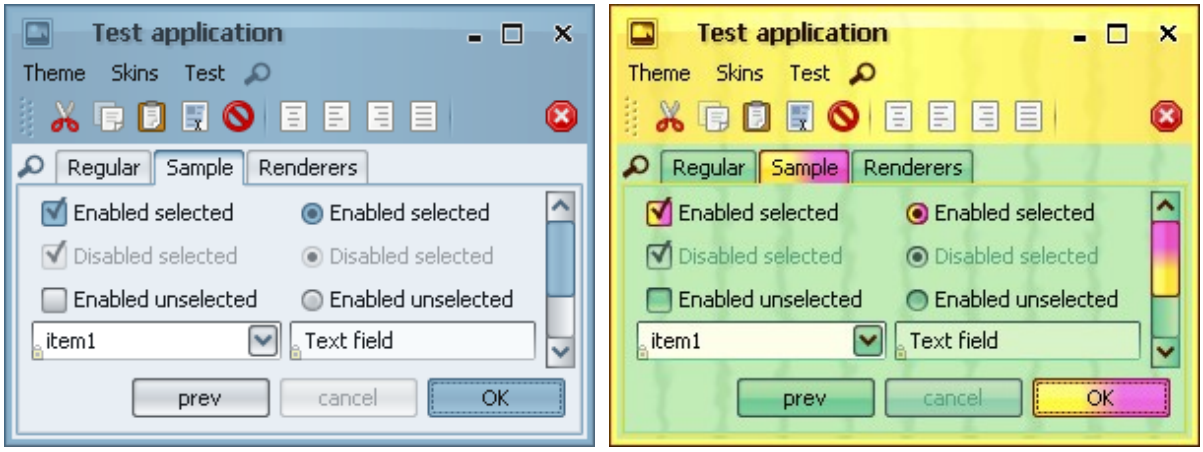


Figure 2.1: Example of Substance L&F with various Themes applied

2.2. How Look and Feel Works

Swing components are painted on screen by component ui delegate. This delegate is assigned to each component according to component's type and current Look and Feel. UI delegate is always a subclass of ComponentUI and can be set to many components by setUI() method.

When the component is rendered, it delegates all the work to its ui delegate.

2.2.1. UI Delegate

ComponentUI class is responsible for painting the component, but this is not delegate's only responsibility. ComponentUI also sets some initial properties of Swing component (e.g. foregroundColor, font or minimumSize).

When ui delegate is set to the component. Method ComponentUI.installUI(JComponent) is invoked. In this method the delegate can set components properties and register event listeners. When ui delegate is removed from the component uninstallUI(JComponent) is invoked.

All ComponentUI methods are called with a component as a parameter. The instance of ui delegate can be thus shared by multiple components. Shared ui delegates can use client properties of JComponent[3] to save any state value.

2.2.2. Property management

There is a problem how to distinguish if some property of a component was set by the user or by previous L&F during installation of a new delegate.

The mechanism using marker interface UIResource solves this problem. There are subclasses of Color, Font, Insets etc. which implements UIResource. These subclasses are named: ColorUIResource, FontUIResource etc. Because they are subclass of the original type they can be used interchangeably as a value of property.

When an ui delegate sets property of a component it uses only the UIResource versions of resources. Later everybody is able to distinguish who have set the property.. Whole situation is more complicated because there are many types of properties with different ways of setting or getting their values:

- **simple property**

  Like margin of JButton. Programmers always set this property with Insets class while the ui delegates use InsetsUIResource. When ui delegate is setting its resource, it gets actual value of the property and if this value is not null and not instance of UIResource, delegates never replace it.

- **inherited property**

  Foreground property of JComponent is inherited from component's container. If the current value of the color is null getForeground() method returns color of component's container.

  The foreground color of whole subtree of components can be changed only by setting foreground color of their common ancestor[4].

  There exists methods like isForegroundSet() which return true if the real value of a component's property is set. UI delegates then replace property value only if the property is not set or if it is an instance of UIResource.

- **delegated properties**

  If the value of components minimumSize property is null then getMinimumSize() method returns value obtained from call to ui delegate. Same applies to the maximumSize property and the preferredSize property. These values are not installed by ui delegate because current mechanism already delegates them to ui delegate..

3 putClientProperty(Object,Object), Object getClientProperty(Object)

4 In reality this mechanism doesn't work because current themes always sets values of basic properties like font or background from global ui defaults map. And these properties are usually present there.

- **simple type property**

  Simple type is not object and we cannot use UIResource “trick”. Such properties has in swing components own boolean flag. This flag is set only if the property was assigned by user using the property setter method. UI delegates can set value of a simple property with package private JComponent.setUIProperty() or public static LookAndFeel.installProperty(). These methods identify properties by their text representation and change them only if boolean flag is off.

  Mechanism of boolean flags do not support all simple type properties[5]. If we want to manage these unsupported properties we must do it on our own. Maybe we can use PropertyChange listener to be informed, when someone have called property setter. Or simply we can ignore user request. UI delegate is installed to the component in the constructor thus prior a programmers call of any property setter. If the L&F is not changed during whole life cycle of a component the effect of ignoring value assigned by the user never comes.

There is only fixed set of delegated and inherited properties. They are part of JComponent class. Subclasses use other two types.

2.2.3. How UI Delegates are Created

When Look and Feel theme is installed it fills global ui defaults table. UI defaults is a map of of configuration objects. Look and Feels (those based on Basic L&F) usually store in ui defaults map resources which are not part of swing component properties.

All L&F register in ui defaults map classnames of their painting delegates. Each Swing component has the unique[6] identifier called uiClassID. Component returns this id by getUIClassID(). During the construction of component method JComponent.installUI() is called. This method gets the value of uiClassID from a component and uses this id as key to global ui defaults map. From the map it gets back classname of the ui delegate factory factory. The factory is instantiate using default constructor and ui delegate is obtained by calling method createUI() .

Shared instances of ui delegates are enabled by the usage of factory instead of direct instantiation of ui delegate.

2.2.4. Setting L&F

Look and Feel is represented by the class LookAndFeel. This class contains some description methods and then the method getDefaults(). getDefaults() is used to populate global ui defaults map when the L&F is installed.

2.3. Problems Modifying Existing L&F Themes

Before we try to modify the painting mechanism of the particular L&F from Java platform, we must fully understand the relation between ui defaults, properties of the component, and ui delegate from the L&F.

5 Exception is thrown when we use setUIProperty with unsupported property.

6 Or a group of components descending from common ancestor like JButton and JCheckBox which are descendants of AbstractButton.

When the ui delegate renders a component, it use resources from the component. Because the definition of Swing component almost never supports all needed resources for painting process, ui delegate must maintain its additional set of resources. In my work I want to modify painting process of the component and so I looked at the Basic L&F and at the Metal L&F implementation to see how they deal with resources.

- There are no publicly defined constants for keys to ui defaults map and each L&F uses another set of the keys. Values of these keys can be found only in source code of the L&F. At least foreground, background and font resource is present in the ui defaults map for Basic L&F and Metal L&F for each component. The key for these resources is composed from component's ui class id and property name (e.g.: "button.foreground").
- The L&F themes don't use all the properties of JComponent. If the L&F is extends Basic L&F it usually respects the values of all standard properties. If components property is assigned by the programmer the painting mechanism of basic theme ancestor is used for rendering the component.

  For example when you set color of Metal L&F button

  to red you will not see anything like

  but

- Some ui delegates have hardwired resources and there is now way to change their values. Or they do not take resources fro ui defaults map but from system theme. Good examples are GtkLookAndFeel[7] and WindowsLookAndFeel.
- When we want to use multiple Look&Feel in one application we can use a small trick: We can change current look and feel during the curse of creation of a new components. But because L&F do not access global ui defaults map only during creation of component but sometime during rendering process, L&F can collide resulting in unpredictable behavior.
- On the other side the existence of resources in ui defaults map enables the creator of some other new component to read them. For example some own GUI components of NetBeans access ui defaults map.

2.4. Problems Implementing Custom UI Delegate

As I wrote in previous section, it is difficult to modify painting process of a current L&F in a way that the modification will influence only selected components and persist across Java platform releases. When developers write their own ui delegate to overcome these problems they face another set of difficulties.

7 Which in newest implementation delegates painting to the Gtk+ library as described in chapter 2.1: Existing Look and Feel Themes

2.4.1. Fixed Hierarchy

UI delegates do not extend ComponentUI class directly. They extend some specialized classes which are themselves extensions of ComponentUI. setUI() method of a particular component needs specific subclass of ComponentUI. The whole hierarchy of ui delegate classes is made of abstract classes even if subclasses of ui delegates brings no other functionality and are in fact empty.

If the developers design their own sets of ui delegates, they must extends each of their class from specific ComponentUI subclass. They cannot use their own hierarchy.

If the hierarchy was defined by interfaces and ComponentUI content was moved to some abstract basic implementation no such problem would occur.

2.4.2. Hidden UIClassID

The constant defining ui class id of each component is not public. You must look in the source code to obtain this id. It seems that the whole pluggable L&F concept was not intended for public use.

2.4.3. Complexity of ui delegates

The fact that swing components are just empty skeletons gives L&F designer great freedom. On the other hand it makes designer's work rather complex.

UI delegate define both aspects of the component: behavior and appearance. Implementation of ui delegate is usually a huge chunks of nontrivial code.

If developers want to redesign some component from scratch they are forced to bother with both aspects. result can be complex (e.g.: JComboBox) and can lead to huge amount of written and managed code. For example basic implementation of ui delegate of JMenuItem has 1000 lines spanning multiple class files.

2.4.4. Basic L&F Theme

To reduce developers work Java contains Basic L&F Theme. This is not a standalone L&F but the common base for other implemented themes in Java platform. Problem of this L&F is that it is designed as a common repository of code for other L&F and if you want to use it for as a base of your L&F with completely different approach to obtaining resources (like my Skin theme described below) you must rewrite great part of it.

Basic theme use global ui defaults map as a source of all resources. Unfortunately resources are stored in ui defaults map under not publicly specified keys.

In my opinion if some ui delegate implementation would be good as a base class for other delegates then it should strictly separate three parts:

- Component state and configuration parameter.
- Components behavior. Modification of components state according to user input.
- Painting of components state.

This is the same as on the public side of the component where usually exists separate data model and data renderer,

Each Swing component can have subcomponents[8]. This property is heavily used in implementation of ui delegates. Delegates freely adds children to the managed component. For example JComboBox is composed from JButton and JTextField or TableCellRenderer(usually JLabel) depending on value of its editable property.

If the component is compound from other components the specification of its state and the separation of the whole painting process is almost impossible.

Despite it mild tendency to do some basic separation can be trace up in Basic L&F delegates. Usually the method installUI() divides its code between other methods, which are protected so the extender can redefine only some of the aspects of the ui delegate creation.

When you look inside BasicTextUI.installUI() method[9] you will found following code:

```
public void installUI(JComponent c) {
    ...
    installDefaults();
    installDefaults2();
    LookAndFeel.installProperty(editor, "opaque", Boolean.TRUE);
    LookAndFeel.installProperty(editor, "autoscrolls",
                                        Boolean.TRUE);
    ...
    installListeners();
    installKeyboardActions();
    updateBackground(editor);
    ...
}
```

All the methods are protected and can be one by one reimplemented except the method called installDefaults2() which is private. Inside it the timer controlling caret blinking is initialized. The interval of blinking is taken from ui defaults map and not from some protected property of ui delegate so if you extending BasicTextUI and you do not want to rely on ui defaults map you must copy all the code from installDefault2() to your class and then you must also copy installUI() to not call original installDefaults2().

So at once your class contains many lines of the code which are closely interconnected with parent class (without properly specified interface). Maybe its better in such situation to copy whole ui delegate and rewrite it to your needs.

Or you can forget about it and before calling super.installDefaults() in your delegate you can fill ui defaults with proper value. Than you are cached to the trap of undefined interfaces and “magic” global variables. You must test your L&F with each new release of Java platform.

By forcing the L&F designers to use such undocumented properties, the developers of Swing are also trapped and they must fear to change any aspect of Basic L&F implementation to not break lots of programs and thus slow down whole process of the public adoption of the new Java releases.

8 There is no distinction between container and leaf component. This is because Swing is back compatible with older AWT and because Java has no multiple inheritance and AWT components are not interface only one AWT class has to be selected as a JComponent parent. Container was selected. This also explains why JComponent.setEnabled(false) do not disable whole subtree but only one component unlike other toolkits like GTK+ or .NET (to great confusion of Swing users). Swing isn't able to distinguish if child components are true child which should be disabled or just child installed by ui delegates where ui delegate is responsible for changes of their state.

9 BasicTextUI is base class for ui delegates of JTextField, JTextArea etc.

Question can be asked why this Basic UI is in public package[10] when it is in fact private implementation of some other L&F themes internals. If all the themes and their common basic implementations were in private namespace, these problems wouldn't exist.

The best solution to this problem would be some basic implementation of look and feel which would solve only behavior of component and which would let the user full freedom in implementing appearance.

10 javax.swing.plaf

Chapter 3
Skin Theme Implementation

First I had to choose a way how to modify appearance of the Swing components. I pointed out few requirements on the solution:

- simple design of components look
- appearance can be changed only for selected components (or component subtrees) in application
- modified components must look same on all platforms using any L&F
- design of a component must be packed in to a single file/archive and this package must be simply portable between projects and users
- integration with IDE (NetBeans platform)

In searching for solution I used findings from previous two chapters:

- Appearance of component can be achieved by modification of its properties. Using properties like setFont() is simple and supported in many IDEs. The problem is that only limited set of resources used during painting process of component is accessible through properties. In addition some L&F ignore values of standard properties.
- I can modify global ui defaults map which is intended and used as a repository of painting resources. For certain L&F this works but the keys to UIDefaults map are nowhere specified and are different for each L&F theme and even for one theme between various Java platform releases. I could do some specialized editor for one theme-platform-release but this is not generic solution I am looking for.
- Create set of ui delegates with fixed set of properties

I choose to modify appearance of component by implementing own ui delegate. This ui delegate will be configurable by designer and used as standalone unit.

The usage of public and documented configuration interface of ui delegate is step beyond uncertainty of ui defaults map of current L&F.

But I do not want to do next L&F with configurable color map and fonts. I want to give the designer much more of freedom. I believe, that skin patterns described earlier provide such freedom and still these patterns can be described by fixed set of properties and edited in some GUI.

3.1. Theme

Basic manipulation unit for working with appearance definition will be package compound of ui delegate and its configuration. UI delegate will be only referenced by its name and shared between packages. Some of the packages will have same ui delegate but they will differ only in configuration.

First I elaborate my requirements from beginning of this chapter to the intended theme object:

- Theme must have fully defined set of properties which affects painting process of component.
- Theme should be represented as a standalone file on disk with simple serialization and deserialization of its data.
- There must be simple way how to install theme for
    - one component
    - all component of one type
    - all components in a component subtree
- Theme can support only limited set of Swing components
- Themes can be combined – more themes can be put together and use as one package.
- All external resources (images etc.) to a theme can be specified relatively so the theme file packed with these resources can be moved to another project or user.

Finally I designed theme consisting of two objects Engine and Configuration. Engine is factory which returns ui delegate for supplied Swing component and Configuration defines resources for the ui delegate. Interfaces of the mentioned classes are:

```
public interface Engine {
   ComponentUI getComponentUI(Configuration config, String uiClassID)
        throws UnsupportedUIClassIDException;
}
public interface Configuration {
   String[] getSupportedUIClassIds();
}
public class Theme {
   public void setConfiguration(Configuration configuration);
   public void setEngineClass(String engineClass);
        public Configuration getConfiguration();
   public String getEngineClass();
   ...
}
```

Theme holds only class name of Engine. Serialization is then simpler because we must not think about serialization of Engines internal state which must not be persistent. All the persistent state is part of the configuration.

I completely throw away resources stored in global ui defaults map:

- I want to use one ui delegate with different configuration concurrently. (e.g.: each button in form has another color).
- UIDefaults map is not typed and do not implement JavaBeans pattern for stored properties.

Configuration returns list of class ids. The list of supported ui class ids can be part of Engine but this is more useful in situations when one configuration class can describe more components but specific instance describes only one of them. For example implemented SkinButtonConfig in ExampleSkinTheme is common for some subclasses of AbstractButton (JButton, JToggleButton, JRadioButton, and JCheckBox) but instance describing JButton cannot be used as a definition of JRadioButton.

When the users will install my themes there won't need to specify ui class id and installation theme to the current L&F will be simpler,

In other case when one configuration is common for all components supported by Engine Configuration simply returns all Engines supported ids.

Engine and Configuration can be used in following scenarios:

- one engine, more components, one configuration

  Engine supports more components and configuration contains common resources. This is same as MetalUI and MetalTheme.

- one engine, more components, multiple configurations

  Engine supports more components, but each of them has its own configuration. Of course such engine can be split into more engines(one for each configuration) but keep all of them in one can reduce complexity when used.

- one engine, one component, one configuration

  Extreme of previous case. Specialized engine for just one component.

- meaningful combinations of previous

3.1.1. Configuration is Java Bean

The Configuration object should be preferably implemented using Java Beans pattern. Java Beans are easily serialized and there exist large framework for introspecting their properties.

3.1.2. Serialization

I decided to use XMLEncoder/XMLDecoder mechanism for writing Java objects into streams. For object which doesn't support JavaBeans pattern fully, programmer can register serialization delegate with static method:

```
public class Theme {
    ...
    public static void addPersistenceDelegate(Class<?> type,
                                              PersistenceDelegate
delegate);
    ...
}
```

The main advantage of XMLEncoder is, that encoded file is in fact program written in XML which is run by XMLDecoder to construct serialized object back. Serialization delegate is needed only during writing process and not during serialization. XMLEncoder also supports JavaBeans pattern and can serialize beans without need of specific delegate.

Theme file with suffix .theme contains one Theme object. Theme class contains support for writing itself to the output stream and static method for reading theme from file.

```
public class Theme {
    ...
    public static Theme read(InputStream is) throws IOException;
    public void write(OutputStream os) throws IOException;
    ...
}
```

3.1.3. Using Theme

I can use ui theme in two ways. I can either create engine and ask it for ui delegate for specific ui-class-id or I can register theme in global ui defaults map.

One theme can be registered for one ui class id or for all ones which Theme's Configuration supports. The mechanism of registering theme must be implemented by means of existing method of registering ui delegates. Because I have more demands on the Theme, the registration process is not straightforward as in existing L&F.

Global ui defaults map contains only class name of ui delegate factory.:

```
ComponentUI getUI(JComponent target);
```

Except component no other data are passed to this method. Data cannot be passed even by using JComponent's client properties, because installation of ui delegate is part of the component's constructor. After all I must somehow pass the Theme which I want to use for particular component to the factory. Existing L&F use one factory per ui delegate. This factory is usually ui delegate itself. UIDefaults object caches the factories so they are not created multiple times.

I don't want create factory for each theme and its configuration. In such case each Theme files would have also one Java file with factory. The theme must be saved in some global static variable which will be accessible during ui delegate instantiation. Such global variable is surprisingly global ui defaults map. I could register there each theme under key which would contain theme engine and ui class id. But I didn't do it and used another global static context called ThemeContext.

3.1.4. ThemeContext

One of the initial demand was to enable grouping of themes and manipulation with them as a unit. In addition these manipulation units should be simply combined and installed to the current look and feel. I implemented this functionality in the class ThemeContext:

```
public class ThemeContext {

    public static ThemeContext getDefaultContext();
    public static void push(ThemeContext context);
    public static ThemeContext pop();

    public ThemeContext();
    public void putAdapter(String uiClassID, ThemeFactory adapter);
}
```

I do not register theme directly to ui defaults by calling putAdapter(). I put there classname of ThemeFactory and I remember the original value from ui defaults. When ThemeFactory.createUI() static method is called it asks ThemeContext for actual factory. If there is no such factory in current context or in any previously pushed i fall back to factory registered by current L&F.

There always exist one default ThemeContext. Additional contexts can be activated with static method ThemeContext.push(). And then removed with static method ThemeContext.pop(). Registration and unregistration of contexts must be mirrored.

For simplicity of usage I provide set of install() methods to Theme class which installs themes to any context for some ui-class-id.

```
public class Theme {
   ...
     public void install();      //default context and all suported
uiClassIDs
     public void install(ThemeContext context); // for all supported
uiClassIDs
     public void install(String uiClassID);     //default context
     public void install(ThemeContext context, String uiClassID);
   ...
}
```

3.1.4.1. Example:

```
Theme redTheme = Theme.read(”red.theme”);
Theme blueTheme= Theme.read(”blue.theme”);
ThemeContext blueCtx = new ThemeContext();
redTheme.install();          //installs red theme to the default context
blueTheme.install(blueCtx); //installs blue theme to the blueCtx context
new JButton();               //button will use redTheme
ThemeContext.push(blueCtx);
new JButton();               //button will use blueTheme
ThemeContext.pop();
new JButton();               //button will again use redTheme
```

3.2. Skin Resources

This section describes basic building blocks of ui delegates which use skin patterns.

3.2.1. Images

Images are basic building blocks of skins. Images can be used through java.awt.Image interface but I will work only with image files so I will use only BufferedImage class.

The most important problem when working with images is how to locate them and how to load them. I use ImageResource class for holding the path to the image. This class also contains boolean flag marking whether the path is absolute path in filesystem or if the path is name of the resource on classpath.

```
public class ImageResource {
    ...
    public boolean isResource();
    public String getPath();
    public URL getURL(ClassLoader loader);
    ...
}
```

The method getURL() is related to second problem. How to load image.

If the image is defined by the filesystem path then I can read it using java.io.File. This is but rare case. Usually the theme will be packed with images in some jar file on the classpath. Image path will be in this case relative to the classpath. The supplied ClassLoader to the method getURL() is then used as image name resolver.

I do not always use default system ClassLoader because when I edit themes in NetBeans Editor (described in chapter 4: ThemeEditor Module) my current system ClassLoader use classpath of running NetBeans platform which don't contain the project I'm working on. NetBeans contains mechanism how to obtain virtual ClassLoader of edited project indeed (see section 4.1.3: ClassPath).

Image loading is delegated to the abstract class ImageSource:

```
public abstract class ImageSource {
     public static void setDefaultImageSource(ImageSource defaultSource);
     public static ImageSource getDefaultImageSource();
     public abstract AnimatedImage getImage(ImageResource resource);
}
```

UI delegate use default image source registered in ImageSource class. Initially this is image source implementation which use system class loader for resolving resource name. However in module for NetBeans platform I alter the default image source with implementation which use virtual ClassLoader of edited project.

ImageSource returns AnimatedImage which is described in next section.

3.2.2. Animated Images

I want to support animations of images. Animated image consist of frames. I did not chosen GIF format for animations from following reasons:

- Its more difficult to read metadata of image in Java then the image itself.
- GIF animations describes only delay between images and whether image shell animate infinitely or stop after one cycle.

My animated images consist only from one image which is regularly split in to the frames. The animation metadata are defined in class AnimationInfo which has following properties:

- **`int count`**

  number of frames of the animations

- **`long interval`**

  how often to change frames of animation in milliseconds

- **`boolean repeating`**

  true if animation should animate infinitely

- **`boolean stateReset`**

  whether change of state of the component (states are described in section 3.2.5 below) resets the animation frame

3.2.3. ImageTile

ImageTile is next abstraction build above the AnimatedImage. This abstraction adds the ability to paint itself. BufferedImage can be already painted using methods from Graphics class. But image can be only scaled or fixed.

ImageTile adds to the image the ability to be painted in modes described in section 1.6: Object Resizing.

3.2.4. Multiple Images

One of the skin pattern (see section 1.5.3: Images are Packed in Larger Images) shows how multiple images are packed in one larger. Class ImagePack represents such larger image. When ImagePack is created, it takes source ImageTile and list of rectangles and cuts the source image to sub images which are identified by ImageId[11]. ImagePack manages these child images and returns them on request. All these sub images share underlaying image data thanks to the use of the BufferedImage.

RegularImagePack is another object which can split larger image to child images. RegularImagePack needs only source image and list of ImageId. The source image is split vertically in to the set of child images with the same dimension. It is similar to the animated image but animated image is split to the frames horizontally so RegularImagePack can co-exist with AnimatedImage. Disadvantage is that each sub image of RegularImagePack has the same count of frames.

Naturally RegularImagePack is implemented on top of ImagePack.

3.2.5. States

Having one image of component for each its state is common skin pattern. I want use this concept and I need to specify the states in one manner for all components. Every state must implemented interface State:

```
public interface State {
    String getName();
}
```

Its only marker interface with ability to get state's name which will be useful in theme editor. Each component has its own set of states (which can have only one item in case of JPanel) represented by StateModel interface.

```
public interface StateModel {
    State[] getStates();
    State getState(JComponent component);
}
```

11 ImageId is only marker interface to keep code clean, I do not like plain Object keys.

Method getState(JComponent) is supposed to inspect component's internal state and returns representation of this state.

3.2.6. Canvas

Canvas is fully described in section 1.6: Object Resizing. In my implementation canvas is represented by interface:

```
public interface Canvas {
    public static class TilePlacement{
        public       TileId    id;
        public final Rectangle position;;
    }

    int getTileCount();
    void updateTiles(TilePlacement[] result, int x, int y, Dimension
size);

    Dimension getMinimumSize();
    Dimension getMaximumSize();
}
```

The function updateTiles() fills prepared array with placement of each tile. Tiles are identified by ImageId.

In addition I created implementation of previously proposed grid canvas. GridCanvas is defined by list of resizing modes of each row and column:

```
public class GridInterval {
   /* creates fixed interval */
   public static GridInterval fixed(int width);
   /* creates resizable interval */
    public static GridInterval resizable(int weight);
   public int getWidth();
   public int getWeight();
   public boolean isFixed();
}
```

And with the list of tiles:

```
public class GridTile{
    public GridTile(TileId id, int x, int y, int width, int height);
    public TileId getId();
    public Rectangle getPosition();
}
```

Finally I also created BorderCanvas which is used as an example division of button at the Skin chapter. This BorderCanvas is very simple because it is configured only by one Insets object.

3.2.7. Skins

Skin is an interface useful for painting the whole component. It can define whole component because it defines various dimension of the component and its active area.

```
public interface Skin {
   boolean isOpaque();
   boolean activeArea(int x, int y, Dimension size);
   void paint(State state, int frame,
              JComponent c, Graphics g,
              int x, int y, Dimension size);

   StateModel getStateModel();
   void setStateModel(StateModel model);

   /* returns null if skin is not animated */
   AnimationInfo getAnimationInfo();

   Dimension getMinimumSize();
   Dimension getPreferredSize();
   Dimension getMaximumSize();
}
```

It has also two not yet mentioned properties:

- opaqueness
  return true if skin covers and paint whole its area. if skin covers whole component ant is opaque we must not draw its background
- activeArea
  Defines which points of skin are active for mouse events.

3.2.7.1. CanvasImageSkin

CanvasImageSkin puts together all previously mentioned concepts and their implementations and is configured by following properties:

1. **`StateModel stateModel`**

   Defines which states this skin can have

- **`ImageResource imageResource`**

  Defines master image to use for painting a skin. This image consist matrix of images of same size. Each column is an animation sequence for one state. The order of states is defined by stateModel.getStates(). Number of frames is part of animation info in imageResouce. Non animated image has only one row.

- **`Canvas canvas`**

  Canvas cuts master image to tiles and defines how they will be placed

- **`Map<TileId, FillInfo> fillInfo;`**

  Defines how each sub image will fill its tile.

- **`boolean activeArea`**

  If true then master image contains one addition column not mentioned in stateModel. This column is the first one and contains mask of the skins active area. Active pixels are only pixels with full value in alpha channel.

  For simplicity only first frame of mask's animated sequence is used for resolving active area.

3.3. Editing Support

Editing support for Theme and its Configuration is important. It should be implemented separately from implementation of Engine.

Engine with Configuration can be packed in “engine” library and editor support in “editor” library.

When the theme is designed in a GUI both libraries are needed. But when an application including themes is deployed only “engine” library is needed.

Because editor of theme should be extensible, editing support of each engine must fit one common framework. I created set of descriptor classes which can be registered to editor program and used for editing of themes.

Engines are described by ThemeDescriptor

```
public interface ThemeDescriptor {
    ConfigurationDescriptor[] getConfigurationDescriptors();
    ConfigurationDescriptor getConfigurationDescriptor(
                                Configuration config);
    String getEngineClass();
    String getName();
    String getDescription();
}
```

One engine can support multiple configurations. The ThemeDescriptor return for each configuration one ConfigurationDescriptor:

```
public interface ConfigurationDescriptor {
    /** return true if this descriptor describes some configuration */
    boolean describes(Configuration conf);
    String getName();
    String getDescription();
    Configuration createConfiguration();
    ThemeEditor createEditor();
    ThemePreview createPreview(ClassLoader loader);
}
```

Configuration descriptor can create new default configuration. For any supplied configuration of the same type it can return the editor and a preview class.

Preview class is an element which displays arranged components with applied theme and which can be easily reconfigured. Preview can be used even when theme is not edited as a preview of theme file in some open or navigation dialogs.

```
public interface ThemePreview {
    JComponent getComponent();
    void update(Configuration config);
}
```

Editor returns component for editing theme and can inform about changes of its state.

```
public interface ThemeEditor {

    void setConfiguration(Configuration config);
    void getConfiguration(Configuration configuration);
    JComponent getComponent();
```

```
    void addChangeListener(ChangeListener listener);
    void removeChangeListener(ChangeListener listener);
}
```

Skin support should be also extensible. I created descriptor for them. This time the interface is simpler because we need no preview:

```
public interface SkinDescriptor {
    String getName();
    String getDescription();
    /** creates new skin with state model and default configuration */
    Skin createSkin(StateModel stateModel);
    /** creates new editor of skin with state model and default
configuration */
    SkinEditor getEditor(Skin skin, StateModel stateModel);
    /** returns true if this descriptor describes specified skin */
    boolean describes(Skin skin);
}
public interface SkinEditor {
    JComponent getComponent();
    /** change state model of edited skin */
    void setStateModel(StateModel model);
    /** returns edited skin */
    Skin getSkin();
    /** registers listeners of changes in edited skin */
    void addChangeListener(ChangeListener listener);
    void removeChangeListener(ChangeListener listener);

}
```

3.4. Implementation Notes

I implemented all previous classes and interfaces in two libraries which can be found on enclosed CD together with their Javadoc.

- SwingSkins.jar
  Contains whole framework for creating engines, configurations and ui delegated implementations using skin pattern.
- SwingSkinEditor.jar
  Contains definition of editor interfaces, together with some helper classes which enable simple creation of theme editors.

3.4.1. Example Theme

On top of the SwingSkins and SwingSkinsEditor libraries I created Example Skin Theme. In this theme swing components all have defined their background by skin. I implemented just ui delegate for JButton as “a proof of concept”. Example theme implementation is distributed in two libraries (available at enclosed CD):

- **ExampleSkinTheme.jar**

  implementation of button's ui delegate and configuration

- **ExampleSkinThemeEditor.jar**

  implementation of ThemeDescriptor for button and button's editor and preview

3.4.2. Example Application

I also created application for editing themes which can be used independently on NetBeans. The application is on the enclosed CD in the directory editor/ and has following limits:

- It supports only Themes and Skins which are explicitly configured in source code. User is unable to add other themes or skins. I didn't want to bother with any configuration when I was writing this example application.
- Path of the images in themes can be only full paths in filesystem. There is no support for relative path of resources.

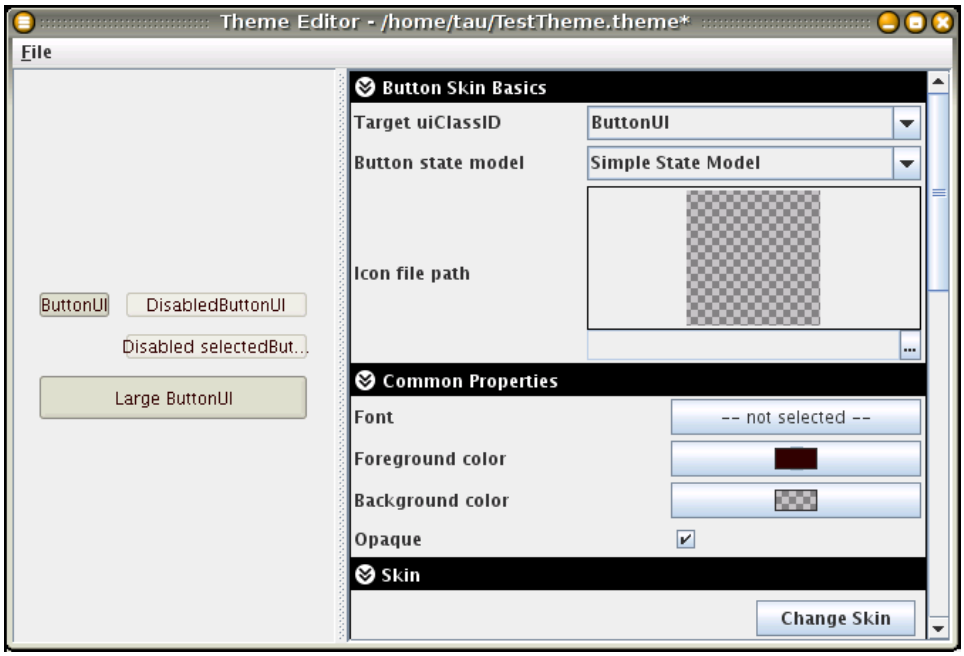


Figure 3.1: Example application main window.

In Figure 3.1 you can see application's main window. On the left side is the preview of edited theme and on the right side is editor itself. Application allows to create theme, save it to disk and then load it and modify.

Chapter 4
ThemeEditor Module

This chapter describes internals of NetBeans module. See Appendixes B and C for instructions how use the module.

NetBeans platform modules are packages of Java classes and other resources which are loaded at the start of NetBeans platform and which extends NetBeans functionality.

ThemeEditor is module which enables to edit theme files in NetBeans.

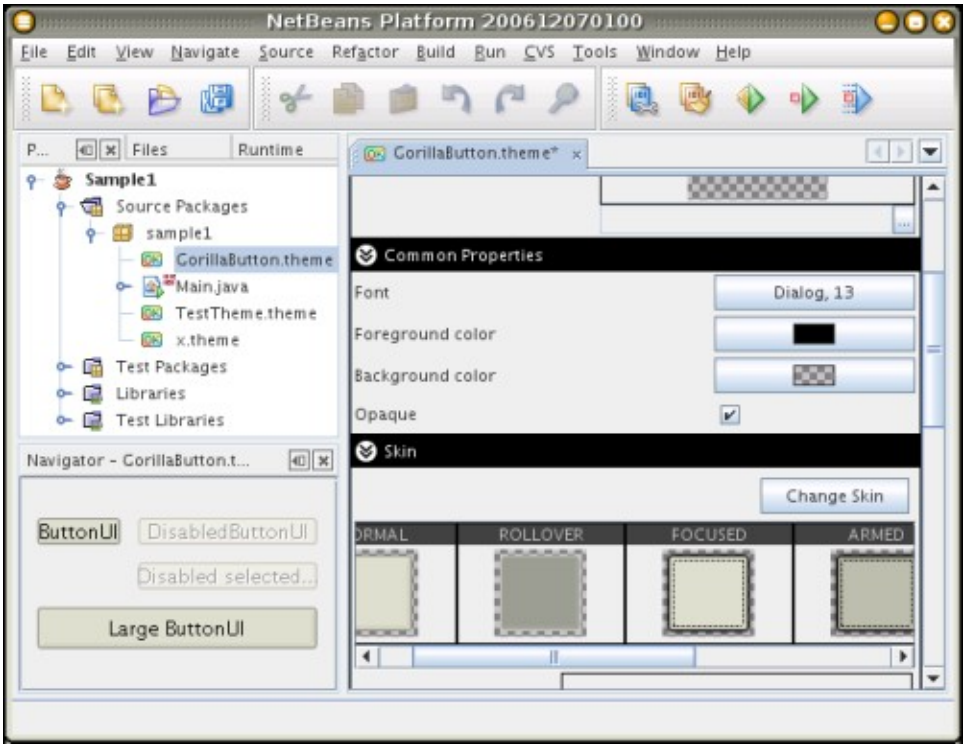


Figure 4.1: NetBeans editing theme file

4.1. NetBeans Concepts

Some NetBeans platform core concepts are worth to mention as I use them for specific purposes in module.

4.1.1. Filesystems and layer.xml

Filesystem interface is an abstraction of the disk filesystem. It holds FileObject objects in tree structure. NetBeans do not use java.io.File objects because Filesystem and FileObject abstractions enable us to have virtual filesystems. For example filesystem representing content of a jar file.

One important implemented Filesystem is System Filesystem because it contains configuration of whole NetBeans platform. This filesystem is pure virtual. Some folder in it contains definition of menus, or objects for resolving file types, some files represents various action etc.

NetBeans can be used as a base for many applications. Bare NetBeans supports nothing. Even they do not support any Java development. Their only functionality is interact with System Filesystem and display menus and windows. All the functionality is added through modules. When some module is loaded it registers its own file objects on various places of System Filesystem tree.

layer.xml is a XML file which defines System Filesystem. Each module contains its own layer.xml. When the module is loaded its layer.xml is merged into the global System Filesystem. During this merge some files and folders are added and others can be hidden (shadowed in NetBeans terminology). In addition there is a mechanism ordering the file object in one folder. Because final order and visibility of file objects depend on the order of merging module's layer.xml files, each modules have list of its dependencies on other modules.

4.1.2. FileObject and DataObject

File Object is abstraction of file in Filesystem. It can be backed by java.io.File or it can be completely virtual. Each file has its set of attributes.

One or more file objects together can represents some data. Data are described by data objects. InstanceDataObject is an example of data object created from file with suffix .instance. This data object can be used as a factory for instantiation objects. Instantiation method and class name are described in file object's attributes.

When obtaining data object from file object NetBeans first resolves mimetype of the file object with mime resolver and then use loader to create data object. There is more data loaders and mime resolvers in NetBeans. Loaders are registered under their mimetype and mime resolvers are in list and are applied all in sequence until some of them is successful in resolving file object's mimetype.

Modules which introduce new file type and operations with it can register their own data loader and mime resolver in to the layer.xml (folder /Loaders/mimetype/ and /Services/MIMEResolver).

4.1.3. ClassPath

Java classpath is a list of directories or jar files in which running Java program searches for class files or other resources. When working with NetBeans there exists many such classpaths. The main one is the classpath of NetBeans itself. Other classpath are those of opened projects. If you specify icon in form designer using icon's resource name, this name is resolved against project's classpath rather then NetBeans classpath. The concept of various classpath is in NetBeans Platform represented by class ClassPath.

4.2. Module

layer.xml (see C.1.3C.1.348layer.xml484848)of the module can be use as a checklist of all important parts of module.

4.2.1. Mimetype

Theme file is a file in which the whole Theme object is serialized using XMLEncoder. This file has the extension .theme. In the module I use the generic mime resolver which resolves files by their extension. I have set it up to resolve files with extension .theme as text/x-java-theme mimetype.

4.2.2. ThemeEditorDataObject

ThemeEditorDataObject is data object representing theme. Data object caches loaded themes so subsequent request for theme must not access disk. Also all modification in editor are made on the copy in cache. This copy is also used by preview Panel and so preview shows actual state of edited theme.

ThemeEditorDataObject use for modification tracking (flag which says if theme definition differs from that on disk) infrastructure provided by DataObject so NetBeans platform is aware of themes modification state.

4.2.3. Preview

For preview window:

is used navigator window. This window tracks selected data objects and then try to load navigator component according to the mimetype of their file object. My implementation of navigator takes theme from current ThemeEditorDataObject and use ThemeDescriptor to obtain ThemePreview component. This component is then returned as navigator panel.

4.2.4. ThemeDescriptor Factory

To get theme descriptor for loaded theme I created class ThemeRegistry. This class keeps map of ThemeDescriptors. This map is initiated from System Filesystem from folder /SwingSkins/Descriptors/. Each file in this folder must be instance file of a class which implements ThemeDescriptor. Others can make their whole new themes with editing support and then create module which will provide this support to NetBeans.

```
<folder name="SwingThemes">
```

```
  <folder name="Descriptors">
    <file name="SkinTheme.instance">
      <attr name="instanceClass"
            stringvalue="com.sun.taus.skintheme.SkinThemeDescriptor"/>
      <attr name="module"
            stringvalue="com.sun.taus.skintheme.SkinThemeDescriptor"/>
   </file>
</folder>
```

4.2.5. SkinDescriptor Factory

ThemeEditor use SkinDescriptor object for editing of skin. Skin descriptors are kept in SkinRegistry class. Registry class is implemented the same way as the ThemeRegistry class (previous section). Skin descriptors are defined in layer.xml in the folder SwingThemes/Skins.

4.2.6. ClassPathImageSource

I created ClassPathImageSource which reads classpath resources using class loader of supplied ClassPath object. This ImageSource is set as a default image source when theme is loaded. Editor and Preview panel can show images defined by relative path to the projects classpath (see 4.1.3: ClassPath).

4.2.7. Deployment

Whole editor consists of few modules:

- **com-sun-taus-swingskins-nb.nbm**

  Wrapper module for SwingSkins.jar library
- **com-sun-taus-swingskins-editor-nb.nbm**

  Wrapper module for SwingSkinsEditor.jar library
- **com-sun-taus-themeeditor.nbm**

  Theme editor module
- **com-sun-taus-skintheme-editor-nb.nbm**

  Wrapper module for SkinThemeEditor.jar library
- **com-sun-taus-skintheme-module-nb.nbm**

  Module with registration of SkinTheme in layer.xml
- **com-sun-taus-skintheme-nb.nbm**

  Wrapper module for SkinTheme.jar library

Only first three modules constitute editor. The last three registers SkinTheme to ThemeEditor.

Chapter 5
Conclusion

After working on this paper and after few years developing GUI in Java and other toolkits (Gtk+, Qt), I am strongly disappointed with Swing. Even if I like Java platform the Swing part is not in my opinion well designed.

The good starting idea of pluggable L&F was not implemented well and today only expert programmers are able modify or create own L&F themes and they need to do lots of coding and reading of source code for that.

Despite it there exist many nice looking themes for Java. What makes my approach different is the amount of freedom I gave to designers.

I think that skins are good solution for describing components appearance. They give more opportunities to alter appearance of component than simple lists of colors while in the same time designer isn't forced to write single line of code.

Next big difference between my Theme object and existing L&F is that my theme object can be set for each component independently. The strict definition of needed resources is a great help for implementing editors.

Bibliography

James Elliott, Marc Loy, David Wood, Brian Cole. Java Swing. Second edition. O'Reilly, 2002. ISBN 978-0596004088

Kenneth F. Krutsch. Professional Java Custom UI Components. Peer Information, 2001. ISBN 978-1861003645

Java SE 6.0 Source Code[online], http://java.sun.com/javase/downloads/, JRL license

Java SE 6.0 Documentation[online], http://java.sun.com/javase/downloads/

Net beans 5.5 Source Code[online], <http://www.netbeans.org/community/sources/>, CDDL license

NetBeans Wiki[online], <http://wiki.netbeans.org/>, information for NetBeans platform developers

NetBeans Modules and Rich-Client Applications Learning Trail, <http://www.netbeans.org/kb/trails/platform.html>, information for NetBeans Modules developers

Gtk Themes[online], <http://developer.gnome.org/arch/gtk/themes.html>, GNOME developers website

[Subst] Substance Java Look & Feel, https://substance.dev.java.net/, Website of the project

[JB101] Sun Microsystems. JavaBeans Specification version 1.01[online], <http://java.sun.com/beans/>

[Batik] Batik SVG Toolkit[online], <http://xmlgraphics.apache.org/batik/>, homepage of the Batik project

[QTT] QuickTime Tutorials[online], <http://www.apple.com/quicktime/tutorials/mediaskins.html>, tutorials covering working with QuickTime technology

[XMLENC] Philip Milne. Using XMLEncoder[online], http://java.sun.com/products/jfc/tsc/articles/persistence4/

Appendix A: Content of enclosed CD

- **`masters-thesis.pdf`**
  text of the thesis
- **`src/`**
  source code
- **`lib/`**
  SwingSkins and SwingSkinsEditor libraries
- **`javadoc /`**
  documentation of SwingSkins libraries
- **`module/`**
  NetBeans ThemeEditor module
- **`editor/`**
  example editor application
- **`tutorial/`**
  data for tutorial from Appendix D

Appendix B: Installation

B.1. Installation of NetBeans Module

Everything needed for developing application with skinned components is contained on enclosed CD.

- Open UpdateCenter Wizard by selecting **Tools→Update Center** in menu
- Select Install manually downloaded modules. Select **next**.
- Select all nmb files in directory netbeans/ on enclosed CD
- Follow the steps of Installation Wizard

Now we have support for editing theme installed in NetBeans. we can now design new Themes but to use them we must also add theme libraries to the project which use the themes. For this we must prepare skin library. Copy two jar files from lib/ directory on CD to your computer on same stable place (jar files will not be in contrast to nbm files copied to NetBeans but they will be used from place selected during their installation).

- Open **Tools→Library Manager**
- Create new library by pressing **New Library** button
- Name the library SwingSkins
- Select **Classpath** tab and add two jar files
- Optionally set javadoc folders in **Javadoc** tab. Again copy javadoc from CD to some stable place on your computer,

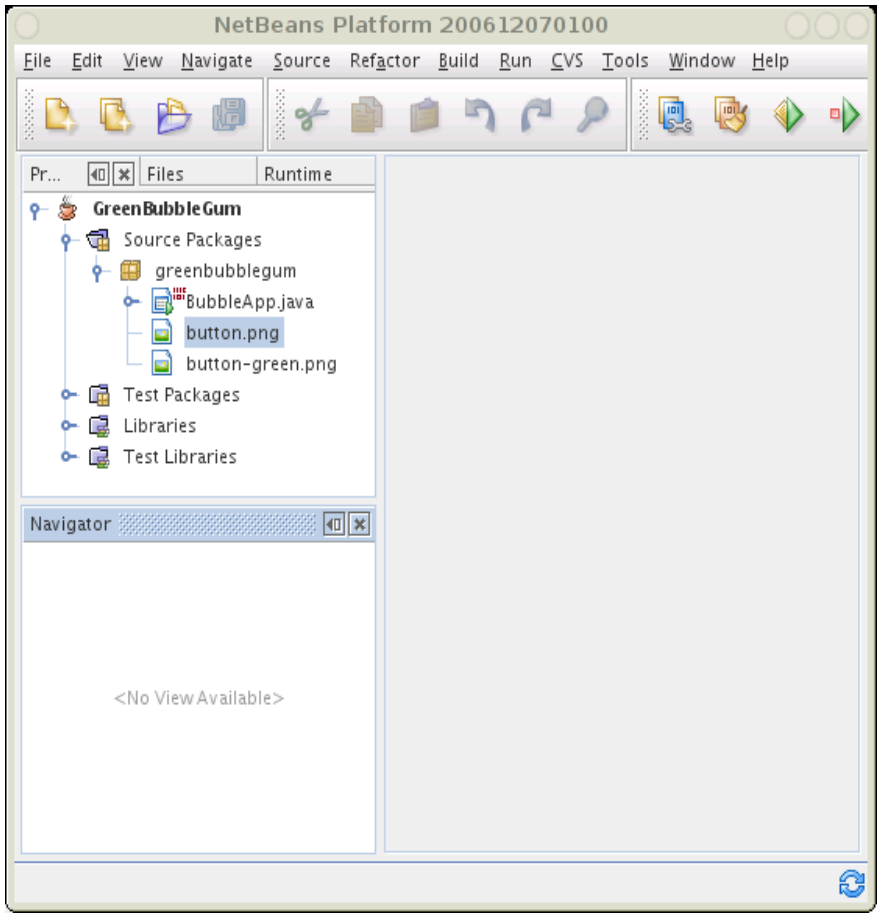
Now you are prepared to proceed with tutorial on next page.

Appendix C: Tutorial

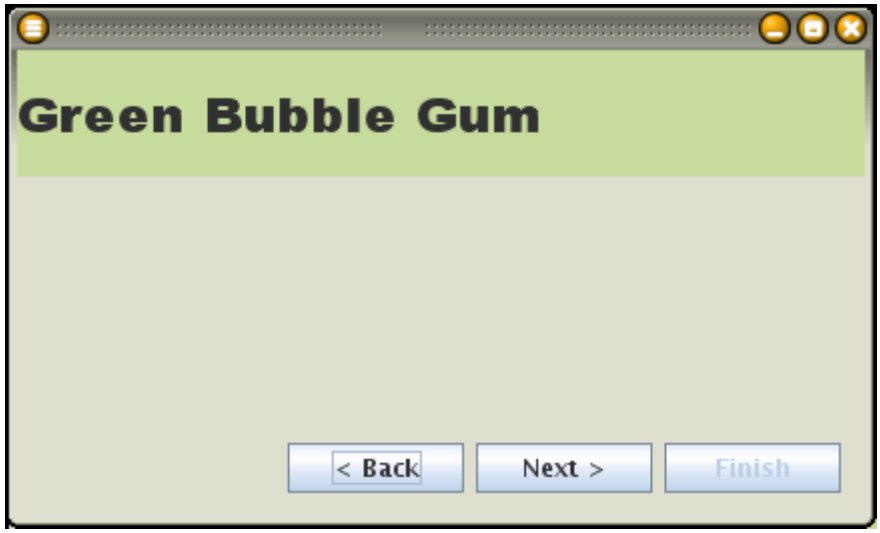
The goal of this tutorial is to demonstrate how new theme can be created and then applied to all components of specific type or only to selected ones.

C.1.1. Preparation

First create new Java project in NetBeans and call it GreenBubbleGum. Copy two images and one Java file from tutorial/ directory on CD. Your should see:



Run your application and see how standard Swing buttons doesn't fit to design of frame.



C.1.2. Theme definition

Now its time to create new theme for buttons.

- Use **New File** select **Other** group and then **Theme file**. Click **Next**.
- On following panel set the name to **button** and put new file to the package **greenbubblegum.** Click next.
- Now select **Skin theme** on the left side. You will use skin theme engine (engine responsible for painting the component). Then on the right side select **Button Skin**. This is one of the configurations which Skin Theme supports.
- Select **Finish**

You've just created first theme file. It have opened in editor. In **Navigator** window bellow **Project View** you see preview of edited theme. Fill following properties in the editor :

Target uiClassID:	ButtonUI
Button State Model:	Rollover StateModel

Continue in Skin section and click **Change Skin**. In the dialog select **Border Skin**. New properties will appear bellow button.

Edit the path of an image (three dot button). Select **Resource** and fill in greenbublegum/button.png. Immediately after acceptation of the dialog there will appear parts of the button background in preview of skin. But it seems that these background images are wrongly aligned. It's because the source image contains also mask of active area which is additional to the button's states.

Check **Active area mask** button bellow and skin preview will be corrected. You've defined skin for a button.

In **Button Properties** section increase left and right margin to 20. Margin describes the space between content of the button (text and icon) and the border. With bigger margin button looks better. You can check **Debug** box to see how the text is positioned in the button.

Save your new theme. In next step we will create additional button theme with different background which will be used for important buttons.

You can repeat steps above to create theme for important button with only one exception. This time the background image will be greenbublegum/button-green.png. Simpler way is to copy the file and change only the image. Save new file as button-green.theme.

C.1.3. Application of Theme

Now we have two theme files and we want to apply them on our buttons. First we must add **SwingSkins** library to the project. Right click on **Libraries** node in P**roject View** and select **Add library** then choose **SwingSkins**.

Open BubbleApp.java and replace with following code content of constructor:

```
Theme themeButton;
        Theme themeButtonGreen;
        try{
             themeButton = Theme.read("greenbubblegum/button.theme");
             themeButtonGreen =
                 Theme.read("greenbubblegum/button-green.theme");
        }catch(IOException e){
```

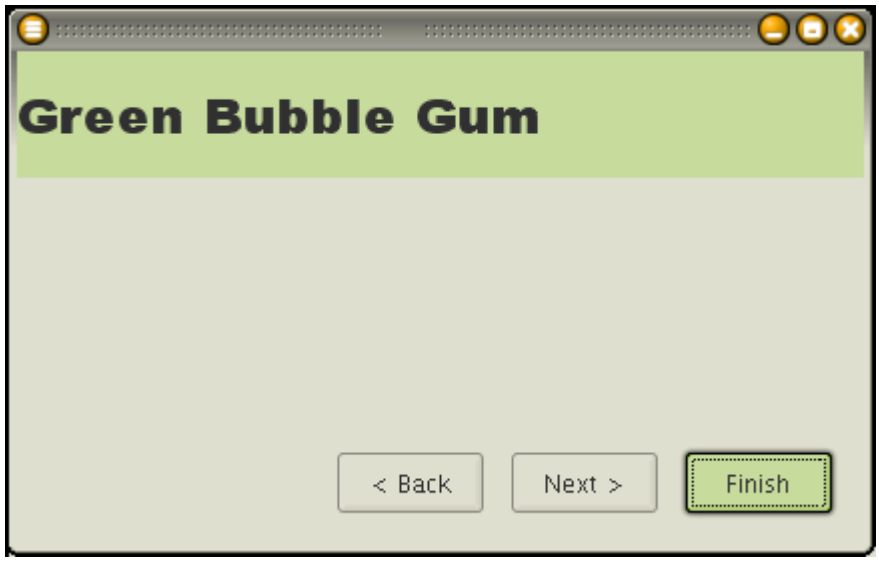
```
            e.printStackTrace();
            return;
        }

        themeButton.install();
        initComponents();
        try {
            buttonOk.setUI((ButtonUI)
                themeButtonGreen.getComponentUI(buttonOk));
        } catch (UnsupportedUIClassIDException ex) {
            ex.printStackTrace();
        } catch (EngineInstantiationException ex) {
            ex.printStackTrace();
        }
}
```

You see that themes are read from theme files. Then themeButton is installed globally and all new created button from this place on will use button theme. Obviously this step must be done before initComponents().

Green theme button is installed only to one component.

Run the application:



Now you can play with various properties of button skin in editor.

Appendix D: layer.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE filesystem PUBLIC "-//NetBeans//DTD Filesystem 1.1//EN"
"http://www.netbeans.org/dtds/filesystem-1_1.dtd">
<filesystem>
    <folder name="Actions">
        <folder name="Edit">
            <file name="com-sun-taus-themeeditor-ActionEdit.instance"/>
        </folder>
    </folder>
    <folder name="Loaders">
        <folder name="text">
            <folder name="x-java-theme">
                <folder name="Actions">
                    <file name="com-sun-taus-themeeditor-
ActionEdit.shadow">
                        <attr name="originalFile"
stringvalue="Actions/Edit/com-sun-taus-themeeditor-
ActionEdit.instance"/>
                    </file>
                </folder>
            </folder>
        </folder>
    </folder>
    <folder name="Services">
        <folder name="MIMEResolver">
            <file name="ThemeEditorResolver.xml"
url="ThemeEditorResolver.xml">
                <attr name="SystemFileSystem.localizingBundle"
stringvalue="com.sun.taus.themeeditor.Bundle"/>
            </file>
        </folder>
    </folder>
    <folder name="Templates">
        <folder name="Other">
            <file name="createTheme">
                <attr name="SystemFileSystem.localizingBundle"
stringvalue="com.sun.taus.themeeditor.Bundle"/>
                <attr name="SystemFileSystem.icon"
urlvalue="nbresloc:/com/sun/taus/themeeditor/theme.gif"/>
                <attr name="instantiatingIterator"
newvalue="com.sun.taus.themeeditor.wizard.CreateThemeWizardIterator"/>
                <attr name="template" boolvalue="true"/>
                <attr name="templateWizardURL"
urlvalue="nbresloc:/com/sun/taus/themeeditor/wizard/createTheme.html"/>
            </file>
        </folder>
    </folder>
    <folder name="Navigator">
        <folder name="Panels">
            <folder name="text">
                <folder name="x-java-theme">
                    <file
name="com.sun.taus.themeeditor.ThemeNavigator.instance"/>
                </folder>
            </folder>
        </folder>
```

```
    </folder>
    <folder name="Windows2">
        <folder name="Components">
            <file name="ThemeEditorTopComponent.settings"
url="ThemeEditorTopComponentSettings.xml"/>
        </folder>
        <folder name="Modes">
            <folder name="editor">
                <file name="ThemeEditorTopComponent.wstcref"
url="ThemeEditorTopComponentWstcref.xml"/>
            </folder>
        </folder>
    </folder>
</filesystem>
```