進歩したSynth

最新のSwingルック・アンド・フィールを使うとカスタムUIが手軽に

Michael Abernethy
Software Engineer
IBM

2005年 2月 01日

Synthのルック・アンド・フィールを詳しく見てみましょう。Java 5.0で登場したSwingに追加されたものとして、Synthは最新のものです。SynthではJavaでのUIプログラミングに「スキン（skin）」という概念を導入しており、これによってアプリケーションに対するカスタム・ルックを迅速に作成、展開できるようになります。この記事では、ソフトウェア技術者であるMichael AbernethyがSynthの概念を順を追って説明しながら、Synthルックを持つアプリケーションをゼロから作り上げます。この記事を読めば、プロの見栄えを持つUIを、短時間で作れるようになるはずです。

Sunは「Javaデスクトップの再導入」を試み続けていますが、Java UI開発者からは相変わらず、完全にカスタムのルック・アンド・フィールを作るのが難しすぎる、という不満が聞かれています。作るのに時間がかかりすぎるだけではなく、SwingのUIコードもそのドキュメントもお粗末に書かれており、計画性なく切り貼りしたように見える部分が多々あります。完全なルック・アンド・フィールを作るためには、Metalルック・アンド・フィールの39クラス、またはBasicルック・アンド・フィールの60クラスをサブクラス化することが必要でした。アプリケーションの見栄えを変えるだけのために、一体誰がパッケージ全体をオーバーライドしたいと思うでしょうか。Swingでカスタムのルック・アンド・フィールを作るのが難しいことを示す、もう一つの例は、多くの開発者がその苦労の成果を無料でオープン・ソースのプロジェクトに寄付している時代にあって、インターネットを探してもSwingでカスタムのルック・アンド・フィールはほとんど無い、という事実です。SourceForge.net（参考文献）にある一握りを含めて、全部で20しかありません。

美しさも一皮むけば

Synthを入力しましょう。Synthはアプリケーションのルック個性化を容易にするものとしてSunが期待しているものです。Synthの目標は単純です。Javaコードを全く書くことなく、新しいルック・アンド・フィールを作れるようにする、ということです。これは良い考えだと思えます。一般的に、プログラマーは最も芸術的なセンスを持っているわけではなく、一方グラフィック・アーティストはJavaコーディングのエキスパートではありません。Synthでは、ルックの記述全体をコードから取り除き、代わりに外部のXMLファイルと画像ファイルに置くように妥協を図って

いるのです。この種のルック・アンド・フィール、つまり完全に外部ファイルによって記述されるものを、スキン（skin）と呼びます。

スキンの概念は別段、Sunが始めたものではありません。例えばWinamp用には何百というスキンが存在しており、Firefox用には数十種類がありますが、これは変更用のXMLファイルだけで簡単に作れるためです。単純にXMLファイルを修正するだけで、素早く容易にJavaアプリケーションのルック・アンド・フィールを作れることを想像してみてください。そしてその結果できる、何百ものユニークなSwingのルック・アンド・フィールも想像してみてください。Java UIの開発者であれば、間違いなくこれを祝福するでしょう。

この記事ではSynthのルック・アンド・フィールの詳細を解説します。完全なルック、つまりスキンを作るために必要なものをすべてお見せしましょう。そしてSynthの重要概念すべてを利用した、スキン化アプリケーションの例についても調べます。その後で、XMLファイルを作りながら、Synthの概念を、順を追って説明して行きます。

また、記事の最後では、Synthのパフォーマンスやバグ、欠陥、Synthがもたらす時間節約などに関して、皆さんが抱く疑問に答えようと思います。この記事を読んだ後、きっと皆さんはルック・アンド・フィールのソリューションとしてSynthを支持し、皆さん自身がSwing用のスキンを瞬時に書けるようになるでしょう。

Synthの基礎

Synthは純真無垢のルックです。完全に空白のキャンバスであり、XMLファイルで何らかのコンポーネントが定義されるまで、全面白のパネルとして現れます。コンポーネントを定義してさえしまえば、アプリケーションに対するSynthルック・アンド・フィールの設定は、これ以上ないほど簡単です。これをリスト1に示します。

リスト1. Synthルック・アンド・フィールを設定する

```
SynthLookAndFeel synth = new SynthLookAndFeel();
synth.load(SynthFrame.class.getResourceAsStream("demo.xml"), SynthFrame.class);
UIManager.setLookAndFeel(synth);
```

ここで、SynthではJavaコードではなくXMLコードなのだ、と理解することが最も重要です。SynthのXMLフォーマットを見ると、最初は尻込みしたくなるかも知れませんが、実は非常に単純なのです。KISSのお念仏（Keep It Simple Stupid: 単純にしておけ、アホ、の意）に従いさえすれば、すぐにXMLファイルが作れるようになり、また新しいルック・アンド・フィールが使えるようになるのです。

KISSのお念仏を頭に入れた上で、Synth XMLファイルのメイン構成ブロック、`<style>`タグを導入するところから始めましょう。`<style>`タグには、コンポーネントのスタイル、つまり色やフォント、画像ファイル、状態、コンポーネント固有のプロパティーなどを記述するための情報がすべて含まれています。一つの`<style>`タグで一つ以上のコンポーネントを記述できるのですが、Synthファイルを作るために一番容易なのは、それぞれのSwingコンポーネントごとにスタイルを作ってしまうことです。

スタイルを作ったら、そのスタイルをコンポーネントにリンクします。`<bind>`タグはSynthエンジンに対して、定義されたスタイルをコンポーネントにリンクするように伝えます（リスト2）。この組み合わせによって、コンポーネントの新しいルックが完全にでき上がることになります。

リスト2. スタイルをコンポーネントにリンクする

```
<style id="textfield">
   // describe colors, fonts, and states
</style>
<bind style="textfield" type="region" key="Textfield"/>
<style id="button">
   // describe colors, fonts, and states
</style>
<bind style="button" type="region" key="Button"/>
```

`<bind>`タグで一つ注意すべきことは、`<bind>`タグの内側にある`key`属性は`javax.swing.plaf.synth.Region`クラスにある定数にマップされる、という点です。Synthエンジンはこうした定数を使って、スタイルを実際のSwingコンポーネントにリンクします。`JButton`や`JTextField`など、単純なコンポーネントは、一つの定数を使います。より複雑なコンポーネント、例えば`JScrollBar`や`JTabbedPane`などは、別々の部分に対して複数の定数を持ちます。

皆さんがSynthフォーマットに慣れ、XMLの中に継承モデルを設定できるようになるまで、私としては「一つのコンポーネントに一つのスタイル」という設定を使うように推奨します。この構造ではXMLの持つ階層機能のすべては利用できませんが、設定やコーディング、デバッグなどのためには最も単純なのです。

もう一つ、Synth XMLファイルを扱う場合に覚えておくべき重要なことは、Synth XMLファイルは何も検証しない、ということです。XMLで綴り間違いをしたり、不正な属性を使ったりしても、そのルック・アンド・フィールがロードされ、ランタイム例外が投げられるまで、その間違いは現れて来ません。どういう意味かというと、「XMLファイルを顧客に出荷する前に、それをテストしておけ」ということです。

デモ用アプリケーション

Synth XMLファイルがどのように動作するかを示すアプリケーションの例として、単純なログイン画面を作る手順を説明しましょう。この画面には充分なだけのコンポーネントがあり、XMLファイルの重要部分と、それらがどのように組み合わされて全体的なルック・アンド・フィールを作るかを見ることができます。

図1と図2を比べてみると分かるように、Oceanルック・アンド・フィールのログイン画面は、ご想像の通り、簡潔で単純、そして、新味のないものです。一方、Synthでのルック・アンド・フィールは、全く異なっています。

図1. Oceanルック・アンド・フィールを持つデモ・アプリケーション

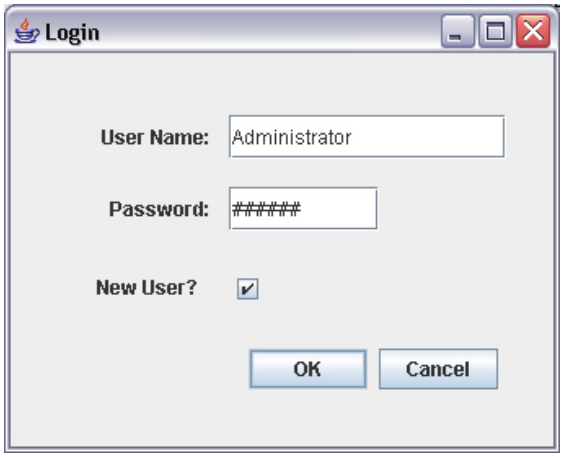

図2. Synthルック・アンド・フィールを持つデモ・アプリケーション

色とフォントを変更する

デモ・アプリケーション用にルック・アンド・フィールを作るための最初のステップは、デフォルトの色とフォントを設定することです。白のAharoniフォントをデフォルトのフォントとすることにします（特にフォントを設定しないコンポーネントのどれに対しても、これがデフォルト・フォントになります）。

フォントを変更するためのXMLは、`<style>`タグ内のどこにでも置くことができます。色は`<state>`タグ内に埋め込みます。`<state>`タグについては、後でまた説明しますが、ここでは、何も属性を持たない単純な`<state></state>`タグがすべての状態を網羅する、ということだけ理解しておけば十分です。

`color`タグ自体には2つの属性が必要です。

- **`value`** は、`java.awt.Color`定数（`RED`、`BLUE`など）の`String`表現、あるいは、前に「`#`」がついた16進表現の色（例えば`#669966`など）。
- **`type`** は、そのファイルがどの領域を設定すべきかを記述します。選択肢としては、`BACKGROUND`、`FOREGROUND`、`TEXT_FOREGROUND`、`TEXT_BACKGROUND`、そして`FOCUS`があります。

fontタグには、必須属性が2つと、オプション属性が1つあります。これらはjava.awt.Fontクラスの3つのパラメーターに直接マップされます。

- **name**: フォントの名前（例えば、VerdanaやArialなど）
- **size**: ピクセルでのサイズ
- **style**: このオプション・タグを指定しないでおくと、標準体のフォントになります。他のオプションとしては、BOLDやITALICがあります。BOLD ITALICのように2つの属性の間に空白を置くことによって、太字と斜字体の両方を指定できます（属性を組み合わせるために空白を使う手法は、Synth XMLファイルのどの属性に関しても有効です）。

最後に、このスタイルをそれぞれのJLabelとそれぞれのJButtonにバインドする代わりに、（.* wildcardを使うことによって）このスタイルをアプリケーションの全コンポーネントにバインドすることもできます。このワイルドカードは、全コンポーネントに対してデフォルトの白いAharoniフォントを与えるように、Synthのルック・アンド・フィールに伝えるのです。リスト3は、コンポーネントのフォントと色を設定するためのXMLコード全体を示しています。

リスト3. 複数コンポーネントのフォントと色を変更する

```
<style id="default">
   <font name="Aharoni" size="14"/>
   <state>
      <color value="#FFFFFF" type="FOREGROUND"/>
   </state>
</style>
<bind style="default" type="region" key=".*"/>
```

画像を使う

図2でのtextfield境界は、ごく一般的な単一ピクセル/角形の境界ではありません。この境界は画像を使って作ります。これは新規な概念ではありません。しばらく前から、画像はボタンやラベルに使われています。しかし皆さんは、少し問題が出てくることが想像できるでしょう。カーソルは、どこに行くべきかをどうやって知るのでしょう？ テキストをペイントするにはどうするのでしょう? 異なるサイズのテキスト・フィールドを作るにはどうするのでしょう？ こうした問題は画像伸張（image stretching）という概念で提起されています。アプリケーション中の、あらゆるサイズのテキスト・フィールドを一つの画像ファイルで記述する必要があるため、画像を適切に伸張する方法や通常textfieldで行われること（carat やテキスト制御）をどう処理するか、XMLファイルに伝えるための方法が必要になります。

幸い、この種の伸張を行うための方法は、スキン化アプリケーションの初期の頃からあるのです。画像は、XMLファイル中の属性によって規定される9つの領域に分割する必要があります。つまり、上部、右上、右、右下、下、左下、左、左上、そして中央です。そうするとレンダラーは、画像が割り当てられた空間に収まるように、ある方法で画像を伸張します。図3はテキスト・フィールドの画像がどのように伸張されるかを示しています。

図3. Synthで画像がどのように伸張されるか

図3で、薄緑色の領域は垂直方向にのみ伸張します。つまり画像よりもテキスト・フィールドの方が背が高い時に縦に伸びます。薄赤色の領域は、画像よりもテキスト・フィールドの方が長いときに、横方向にのみ伸びます。薄黄色の領域が伸びることはありません。テキスト・フィールドがどれほど大きくなっても、この領域は画像ファイルで見えるのと全く同じに見えるように描画されます。この領域は伸張されないため、曲線や特別な色付け、影、その他伸張されると変に見えるもの全ては、この領域に含む必要があります。最後に、中央領域はオプションです。この領域は描画することも、省略することもできます。ここでの例ではテキスト・フィールドの中央は省略されています。そうするとレンダラーはこの領域を使ってテキスト制御や carat を処理します。そして、これで終わりなのです。単一画像を使って、テキスト・フィールドを完全に描画できたのです。

`imagePainter`タグには、ルック・アンド・フィールの中で画像を使うために必要な全情報があります。このタグが必要とする属性は、次の数個にすぎません。

- **`path`**: 使用する画像へのパス
- **`sourceInsets`**: 図3での緑色領域の幅とピンク色領域の高さをピクセル数で表したinsets。上、左、下、右、の順で割り当てます。
- **`method`**: これは恐らく一番混乱しやすい属性と言えるでしょう。これは`javax.swing.plaf.synth.SynthPainter`クラスのファンクションに直接マップします。このクラスには約100のファンクションがあり、そのどれもが`paint`で始まります。それぞれのファンクションがSwingコンポーネントにある、特定なペイント・ジョブにマップします。とにかく正しいものを見つけ、それから`paint`ストリングを取り除き、その後に続く最初の文字を小文字にすることで属性を設定します。例えば、`paintTextFieldBorder`は`textFieldBorder`の属性です。あとはレンダラーが処理してくれます。
- **`paintCenter`**: この属性を使うと、（例えばボタンの中の）画像の中央を保持するか、除去するかを設定できます。ここでの例での`textfield`は、テキストを描画できるように、画像を除去しています。

画像を使って境界をペイントするための最後のステップは、デフォルトの insets を増加して、境界の描画に使う新しい画像を処理することです。insets を変更しないと、何の画像も見えないことになります。insets の中で画像がペイントされるように、`<insets>`タグを追加して insets を増加する必要があります。ほとんどの場合、insets の値は、画像の中で使った insets の値と同じです。

リスト4は、画像をロードするためのXMLコードを示しています。`sourceInsets`を使うことによって、画像の必要部分のみが伸張されることに注意してください。

リスト4. 画像をロードする

```
<style id="textfield">
   <opaque value="true"/>
   <state>
      <font name="Aharoni" size="14"/>
      <color value="#D2DFF2" type="BACKGROUND"/>
      <color value="#000000" type="TEXT_FOREGROUND"/>
   </state>
   <imagePainter method="textFieldBorder" path="images/textfield.png"
      sourceInsets="4 6 4 6" paintCenter="false"/>
   <insets top="4" left="6" bottom="4" right="6"/>
</style>
<bind style="textfield" type="region" key="TextField"/>
```

様々な状態を処理する

ここまでの例で見た通り、コンポーネントを定義する上で、<state>タグが主な焦点となります。リスト3とリスト4では、色とフォントのタグは<state>タグの内側にあります。では、<state>タグが何をするのかを説明しましょう。

テキスト・フィールドや、ラベルの色やフォントを定義するには、<state>タグで何の属性も定義されていないデフォルト状態で充分です（状態が変わるわけではないため）。しかし、例えばボタンなど、状態が変化するコンポーネントでは、状態ごとに、全く異なるルックを定義することができます。それぞれの状態が、独自の色やフォント、そして画像を持つことができるのです。ログイン画面のCancelボタンのデフォルト状態（図4）とmouse-over状態（図5）とを比較してみてください。

図4. デフォルト状態でのCancelボタン

図5. MOUSE_OVER状態でのCancelボタン

<state>タグに必要なのは、実際のコンポーネント状態を定義するvalue属性のみです。リスト3と4のようにvalueを規定しないと、すべての状態にデフォルトが適用されます。valueを規定する場合の選択肢は、ENABLED、MOUSE_OVER、PRESSED、DISABLED、FOCUSED、SELECTED、そしてDEFAULTです。これらの選択肢は、Swingの全コンポーネントがとりうる、すべての状態を網羅しています。また、それぞれの状態の間にandを追加することによって、状態を組み合わせることもできます。例えば、ボタンの上にマウスがあり、そのボタンが押された場合にボタンのフォントを変更したい場合には、状態の値としてMOUSE_OVER and PRESSEDを使います。

リスト5は、デモ・アプリケーションの状態を処理するためのXMLを示します。それぞれの状態が別々の画像やテキストの色を定義している様子に注意してください。

リスト5. 状態を処理する

```
<style id="button">
   <state>
      <imagePainter method="buttonBackground" path="images/button.png"
         sourceInsets="9 10 9 12" paintCenter="true" stretch="true"/>
      <insets top="9" left="10" bottom="9" right="12"/>
      <font name="Aharoni" size="16"/>
      <color type="TEXT_FOREGROUND" value="#FFFFFF"/>
   </state>
   <state value="MOUSE_OVER">
      <imagePainter method="buttonBackground" path="images/button_on.png"
         sourceInsets="9 10 9 12" paintCenter="true" stretch="true"/>
      <insets top="9" left="10" bottom="9" right="12"/>
      <color type="TEXT_FOREGROUND" value="#FFFFFF"/>
   </state>
   <state value="PRESSED">
      <imagePainter method="buttonBackground" path="images/button_press.png"
         sourceInsets="10 12 8 9" paintCenter="true" stretch="true"/>
      <insets top="10" left="12" bottom="8" right="9"/>
      <color type="TEXT_FOREGROUND" value="#FFFFFF"/>
   </state>
   <property key="Button.margin" type="insets" value="0 0 0 0"/>
</style>
<bind style="button" type="region" key="Button"/>
```

<state>タグを扱う場合に重要なことは、どのコンポーネントがどの状態に対応するのかを理解することです。この例では明確に分かる通り、ボタンの持つ状態は、デフォルト、マウス・

オーバー、そしてボタンが押された状態です。この例ではフォーカス（focused）と使用不可（disabled）という状態を定義することもできます。しかし、例えばパネルの場合には、選択（selected）は全く適用できず、マウス・オーバーでパネルの状態を変更しても、単に目障りなだけでしょう。

コンポーネント特有のプロパティーを扱う

どのコンポーネントに対しても汎用性のあるXML属性を定義しても、網羅しきれないコンポーネント特有のプロパティーが残ってしまうものです。そうした例としては、リストの行高さ、ラジオボタンのアイコン、メニューの矢印アイコンなどがあります。コンポーネント特有のプロパティーとして定義できるものには100種類以上ありますが、それぞれに対してXML属性を定義するのは過剰だと言えるでしょう。そこでSynth XMLファイルでは、コンポーネント特有のプロパティーを設定できるようになっているのです。`<property>`タグは`Hashtable`のように動作し、プロパティーを設定するために鍵と値の対を定義します。

ログイン画面の例でのチェックボックスを見ると、コンポーネント特有のプロパティーをどのようにコード化するかが分かります。`imageIcons`を定義することによって、デフォルト状態と選択された状態の両方での`CheckBox.icon`のプロパティーを設定できるのです。それほど簡単なのです・・・いや、100あるプロパティーの中から自分に必要なものを見つけ出すくらいに簡単だ、ということなのですが・・・。

リスト6は、ログイン画面用にコンポーネント特有のプロパティーをコーディングするためのXMLを示しています。最初に`imageIcon`を定義することに注意してください。次に、imageIcon のIDを使って、チェックボックスの各状態に対するアイコンを設定します。

リスト6. コンポーネント特有のプロパティーを定義する

```
<style id="checkbox">
   <imageIcon id="check_off" path="images/checkbox_off.png"/>
   <imageIcon id="check_on" path="images/checkbox_on.png"/>
   <property key="CheckBox.icon" value="check_off"/>
   <state value="SELECTED">
      <property key="CheckBox.icon" value="check_on"/>
   </state>
</style>
<bind style="checkbox" type="region" key="Checkbox"/>
```

カスタム・ペインターに対応する

図2に示すログイン画面を定義するための最後の部分が、曲線を持ち、色が次第に変化する背景を描画することです。これをXMLで表現するのは難しそうです。正直なところ、確かに難しいのです。しかしこれも、SynthでUI設計に使えるのは画像と単純な色のみではないことを示す好例となります。Synthを使えば、なんでも描画できるのです。

Synthでは、`SynthPainter`をサブクラス化し、カスタム・ペイントしたい特定のファンクションのみをオーバーライドすることによって、Synthのペイント・メソッド（`javax.swing.plaf.synth.SynthPainter`にあります）をオーバーライドできるのです。この例では、（このデザインはSynth XMLフォーマットでは記述できないので）`paintPanelBackground`メソッドをカスタム・ペイントする必要があります。

カスタム・ペインターを使うには、あるいは、どんな方法であれXMLの中でクラスを作るには、`<object>`タグを使います。`<object>`タグを使うと、Synthのレンダリングを補うために使用する任意のJavaクラスを作ることができ、また持続させることができます。`<object>`タグは、下記の2つの要素があります。

- **`class`**: 作るべきクラスの完全な名前
- **`id`**: XML文書の中でこのクラス・インスタンスを参照するために使うID名

オブジェクトを使うことによって、背景をペイントするクラスである`BackgroundPainter`クラスのインスタンスを作れるだけではなく、背景色を定義する`ColorUIResource`クラスのインスタンスも作ることができます。ちょっと考えてみてください。背景で使われている色を`BackgroundPainter`クラス内で定義すること自体が、Javaファイルにハード・コーディングせず全て外部XMLファイルで定義する、というSynthの目標と矛盾しているのです。

カスタム・ペインターを使うための最後のステップは、Synthのレンダリング・エンジンに対して、（`SynthPainter`クラスではなく）あなたがファンクションを提供するのだと伝えることです。この例では、あなたが`BackgroundPainter`クラスで`paintPanelBackground`ファンクションを定義し、Synthには`SynthPainter`クラスを使って残りのペイント・ファンクションを定義させるのです。`<painter>`タグを使うと、`SynthPainter`ファンクションをオーバーライドできるようになります。このタグには、下記の2つの要素があります。

- **`method`**: カスタム・ペインターがオーバーライドすべきメソッド。画像を使うで学んだ通り、こうしたファンクションは`javax.swing.plaf.synth.SynthPainter class`にありますが、各ファンクションの先頭にある`paint`ストリングは除く必要があります。（従って、例えば`SynthPainter`での`paintPanelBackground`は、XMLファイルでは`panelBackground`になります。）
- id: メソッドをオーバーライドするクラスへの参照

カスタム・ペインターで色を使うためには、色を`javax.swing.UIDefaults`クラスに保存する必要があります。リスト7とリスト8を見ると分かるように、色を`UIDefaults`に保存するのは単純であり、UIを作ったことのある人には、おなじみのはずです。XMLの中で定義すべきものとして鍵となるのは、`BackgroundPainter`用の実際のJavaコードで色を取得するために使う、`UIManager`での参照でしょう。

リスト7は、このサンプル・アプリケーションでカスタム・ペインターを扱うためのXMLコードを示しています。最初に色を定義する必要があることに注意してください。

リスト7. カスタム・ペインターを扱う

```
<style id="panel">
   <object id="background" class="demo.synth.BackgroundPainter"/>
   <object class="javax.swing.plaf.ColorUIResource" id="startColor">
      <int>30</int>
      <int>123</int>
      <int>235</int>
   </object>
   <defaultsProperty key="Panel.startBackground" type="idref" value="startColor"/>
   <object class="javax.swing.plaf.ColorUIResource" id="endColor">
      <int>1</int>
      <int>20</int>
      <int>80</int>
   </object>
   <defaultsProperty key="Panel.endBackground" type="idref" value="endColor"/>
   <painter method="panelBackground" idref="background"/>
</style>
<bind style="panel" type="region" key="Panel"/>
```

リスト8は、サンプル・アプリケーションでのカスタム・ペイント・クラスのJavaコードを示しています。

リスト8. カスタム・ペイント用のJavaコード

```
public class BackgroundPainter extends SynthPainter
{
   public void paintPanelBackground(SynthContext context,
                                    Graphics g, int x, int y,
                                    int w, int h)
   {
      Color start = UIManager.getColor("Panel.startBackground");
      Color end = UIManager.getColor("Panel.endBackground");
      Graphics2D g2 = (Graphics2D)g;
      GradientPaint grPaint = new GradientPaint(
          (float)x, (float)y, start,
          (float)w, (float)h, end);
      g2.setPaint(grPaint);
      g2.fillRect(x, y, w, h);
      g2.setPaint(null);
      g2.setColor(new Color(255, 255, 255, 120));
      g2.setRenderingHint(
          RenderingHints.KEY_ANTIALIASING, RenderingHints.VALUE_ANTIALIAS_ON);
      CubicCurve2D.Double arc2d = new CubicCurve2D.Double(
          0, h/4, w/3, h/10, 66 * w, 1.5 * h, w, h/8);
      g2.draw(arc2d);
      g2.setRenderingHint(
      RenderingHints.KEY_ANTIALIASING, RenderingHints.VALUE_ANTIALIAS_OFF);
   }
}
```

より高度な設定

このセクションでは、ログイン画面の例で見てきたもの以上の手法を幾つか紹介します。こうした手法は、皆さんが独自のSynthルックを作るときに役立つと思います。

非Swingコンポーネントをペイントする

どのSwingコンポーネントのルックも変更できるのは素晴らしいことですが、Synthは他のコンポーネント、つまりSwingが取り残した隙間を埋めるために開発者が作ったコンポーネントのルッ

クも変更できる必要があります。そうした場合には、Swingコンポーネントがペイントされていないことを反映するために<bind>タグを変更する必要があります。type属性は2つの値を持ちます。つまりSwingコンポーネントにマップされていればregion、非Swingコンポーネントマップされていればnameという2つの値です。従って、<bind>タグを<bind style="mystyle" type="name" key="Custom.*"/>に変更すると、mystyleスタイルを使うためには、クラス名がCustomで始まる全コンポーネント（例えばCustomTextFieldやCustomLabelなど）を変更することになります。

スタイルの階層構造

XMLファイルを作るためのKISSスタイルから決別し、スタイルの階層構造を作ってコンポーネントに適用することもできます。リスト9を見ると、これがよく分かるでしょう。Synthが、最後に定義された属性を使ってコンポーネントをレンダリングしていることに注意してください。

リスト9. 階層構造の例

```
<style id="base">
   <color value="BLACK" type="BACKGROUND"/>
   <state>
      <font size="14"/>
   </state>
</style>
<bind style="base" type="region" key=".*"/>
<style id="sublevel" clone="base">
   <color value="RED" type="BACKGROUND"/>
</style>
<bind style="sublevel" type="region" key="Label"/>
```

リスト9のコードによって、全てのコンポーネントは、赤の背景を持つラベルを除き、フォント・サイズ14、黒の背景となります。sublevelの中でbaseスタイルを複製すると、これが全体のスタイルにコピーされることになります。そのあとで、必要とする特定なプロパティーをオーバーライドします。

Synthのパフォーマンスや信頼性、効率を検証する

Synth用のXMLファイルの作り方や、フォントや色の変更、画像の追加によってカスタム・ルックを作る方法は学んだので、恐らく皆さんはSynthに関して聞きたいことがあると思います。しばらくSwingを扱ったことのある人であれば、まずパフォーマンスの問題を最初に考えるでしょう。私は、SynthによってUIが這うような速さにはならないことを示す、パフォーマンス・テストを作ってみました。また、皆さんが経験しそうな問題領域を調べるために（そして、Synthを扱う中で突き当たった問題を議論するため）、Java Bug Parade（参考文献）でSynthに関して見てみました。そして最後に、皆さんが抱く最も重要な疑問、つまり「Synthで本当に時間が節約できるのか？」という質問に答えようと思います。

多くの画像をロードするとSynthは遅くならないのか？

私はこの質問に答えるために、また他のルック・アンド・フィールと比べてSynthがパフォーマンスの面でいかに優れているかを示すために、2つのテストを作りました。最初のテストでは、この例で挙げたログイン画面のアプリケーションをロードするための時間をテストします。このテストは6つの画像をSynthにロードし、この時間を、開発者が作りそうな平均的画面をロードする時間と比較します。2番目のテストでは、ロード時間に対するストレス・テスト、つまり100以上のコンポーネントを持つフレームのロード時間を測ります。

どちらのテストも、比較のためにOceanとMotifのルック・アンド・フィールについても時間を測定します。公正な評価をするために、両方のテストを3台のマシンで実行しました。Windows XPラップトップ、SuSE Linuxマシン、そしてRed Hat Linuxマシンです。その結果を表1と表2に示します。

表1. ログイン画面の平均ロード時間

マシン設定	Ocean	Motif	Synth
Windows XP - 1.7GHz - 2GB RAM	.32 秒	.29 秒	.57 秒
SuSE Linux 9.0 - 3.3GHz - 2GB RAM	.23 秒	.20 秒	.45 秒
Red Hat Linux 3.0 - 1.4GHz - 512MB RAM	.37 秒	.32 秒	.61 秒

表2. 100のコンポーネントを持つ画面の平均ロード時間

マシン設定	Ocean	Motif	Synth
Windows XP - 1.7GHz - 2GB RAM	.33 秒	.32 秒	.34 秒
SuSE Linux 9.0 - 3.3GHz - 2GB RAM	.23 秒	.23 秒	.30 秒
Red Hat Linux 3.0 - 1.4GHz - 512MB RAM	.40 秒	.40 秒	.43 秒

ご覧の通り、Synthルック・アンド・フィールのロード時間は、OceanやMotifよりも、ごくわずかしか遅くありません。ただし、ログイン画面のロードの方が、ストレス・テストのロードよりも遅いことに注意してください。これがおかしいことはすぐに分かりますが、よく調べてみると、犯人が見つかりました。ストレス・テストではチェックボックスに使っている画像をロードしませんが、ログイン画面ではロードするのです。つまり、Synthルックに使われる画像が追加されるごとにロード時間が増加する、ということです。同じ画像を使う100のコンポーネントの方が、2つの画像を使う2つのコンポーネントよりもロードが早いのです。使う画像の数を減らせば、Synthのロード時間パフォーマンスは改善されることになります。

Synth はSwingの最初のリリースと同じくらいバグだらけですか？

SunのJava開発者向けWebサイトのBug Paradeから判断すると、Synthはバグもなく、きれいな製品のように見えます。しかし、どんなソフトウェアも完全ではありません。ある時点ではSynthには125のバグがあり、そのうち`JTabbedPane`の扱い方に関するものが不釣り合いに多かったのです。ですから、この領域に関して何か問題があったとしても、驚くには当たりません。ただし、Sunを弁護するために言うと、これらの欠陥はすべて「Closed (解決済み) 」の状態です。しかし以前問題があったところでは、恐らく将来も問題が起きるであろう、という話もよくあることです。

Synthのバグ・データベースを見る限りではきれいに見えますが、私はログイン画面をいじっている時に、他の問題を幾つか見つけました。最初、`JPanel`の背景色を変更しようとしたら失敗したのです。私は`JPanel`専用にスタイルを作り、それをすべての`JPanel`にバインドしたのですが、うまく行きません。ところが、代わりに私独自のカスタム・ペインターを使ってみると、うまく動作したのです。

状態が変化する際にウィジェットを再ペイントすると、もっと大きな問題が出てきます。私はボタンやその状態をいじっている際に、ボタン上のテキストをいじっても色がうまく変わらないことに気がつきました。初期化時にデフォルト色の白が現れません。状態変化がトリガーされると現れますが、またデフォルトに戻ってしまうのです。Synthに関するドキュメンテーションをあさってみると、小さなコメントがありました。「状態ごとに別々のフォントを与えることはできますが、一般的に言ってウィジェットは状態変化を再検証しません。従って、それぞれの状態に対して大幅にサイズの異なるフォントを使うと、サイズ調整に問題が起きる可能性があります。」どうやら彼らは、古いレガシーのSwingコードをSynthで扱えるようにしようとして、問題に突き当たったようです。ですから、状態変化時にフォント・サイズを変更する場合には要注意です。

実際のところ、Synthは比較的バグが少ないようです。しかし、動作するはずのコードが動作しない部分で変なことが起きたとしても、不思議ではありません。ただし、回避策を見つけるのは難しくないはずです。私は作業中に生じた問題どれに対しても、回避策を見つけることができました。

Synthで完璧にプロ級のルック・アンド・フィールを作ることはできますか？

答えは「イエス」です。Java 1.4でリリースされたGTK+とWindows XPのルック・アンド・フィールは、Synthのみを使って作られたものです（ 当時は公開のAPIではありませんでした ）。ですから、確実に可能です。

Synthで完全なルックを作ると、Javaコードで作るのと比べ、どのくらい早くできますか？

これは簡単に計算できます。どちらの手法を使ったとしても、2つのステップがあります。

1. ルックを作ること。これは通常グラフィック・アーティストが行います。
2. グラフィカルな作品をコードに変換すること。

JavaでのコーディングでもSynthを使う手法でも、グラフィック・デザインに要する時間は同じです。カスタム・ルックを作ることに関しての私の経験からすると、アプリケーション用に完全なルックを作るために、概算で2人のグラフィック・アーティストで2週間かかります。つまりグラフィックの作業に4人週ということになります。

これも私の経験に基づいて言うと、グラフィックスの下絵をサブクラス化して、完全な、そのまま使えるルック・アンド・フィールにするまでには、Javaのコーディング要員3人で約2ヶ月かかります。つまりJavaコーディングに6人月です。UIサブクラスをオーバーライドしてSwingで完全なカスタム・ルック・アンド・フィールを作るには、グラフィックスの作業に加え、7人月かかります。こうした数字を考えてみれば、インターネットでダウンロードできるものが僅かしかない理由が分かるでしょう。

Synthはグラフィックに関する作業をXMLファイルに変換するため、非常に大きな時間節約となります。Javaコードでルックを作るための6人月が、グラフィックに関する作業をSynth XMLファイルに変換するために1人の開発者が2週間作業するだけでよくなるのです。これによって、Synthで完全なルック・アンド・フィールを作るための合計時間は、6人週だけ、つまり5ヶ月以上もの時

間がSynthを使うことで節約できることになります。2人のグラフィック・デザイナーと2人のプログラマーのチームで、たった3週間で完全なSynthルックを作ることができるのです。

まとめ

SynthはSwingにスキンの概念をもたらしました。Javaコードでカスタムのルック・アンド・フィールを書くという伝統的な方法に比べて、Synthを使う最大の利点は、時間の節約です。Swingのルック全体が1ヶ月以内に作れるということは、Javaでコード化するよりも5倍速いということです。あるいは、もっと野心的な開発者であれば、Javaでコード化した1つのルックの代わりに、5つのSynthルックを作ることもできるのです。

ただし、Synthの世界では全てが完全なわけではありません。Swingのルックをオーバーライドするjavaコードを書くということは、アプリケーションのルックだけではなく、フィールまで完全に変えてしまうことを意味します。Synthで変更できるのはルックだけです。この差は非常に大きなものです。ルックは単に見栄え、つまりアプリケーションで使われる色やフォント、画像といった、見た目だけのものです。一方、フィールは、ユーザーとのやり取りの際の、アプリケーションの振る舞い、つまりここで右マウスクリック、ここでキーを押す、といった動作に対応します。例えば、`JList`の振る舞いを変更し、左マウス・クリックでアイテムを選択し、右マウス・クリックでアイテムを削除したい場合には、Synthではできません。新しいルック・アンド・フィールを作るためにJavaコードを書く必要があります。実はSynthは、ルック・アンド・フィールではなく、新しいSwingルックと言うべきものです。SynthでUIのルックはすぐに変更できますが、フィールは常にデフォルトのSwingのフィールなのです。

とはいえ、アプリケーションの見栄えを一新したいとか、腐ったようなMetalのルックよりもSwingアプリケーションの見栄えを良くしたいと長年思っていた人にとっては、Java 5.0の登場によって、古いルックは歴史の遺物となりました。Synthが素晴らしい選択肢となるのです。Synthはパフォーマンスの問題もなく、比較的バグも少ないようです。そしてSunはGTK+ルックをリリースすることによって、Synthで完全なルックが作れることを既に示しています。

現在のところ、Synthに関するドキュメンテーションやサンプルは驚くほど僅かしかありません。皆さんはこの記事を読んだことによって、Synthがどのように動作するかを理解し、また「一つのコンポーネントに対して一つのスタイル・タグ」という設計を使って完全なSynth XML文書を作れるようになったはずです。Synthの継承、階層モデルには、スタイル・タグを作るためにさらに強力な方法があるのですが、そんなものを使わなくても、完全なルックを作ることはできるのです。Synthが知られるにつれ、爆発的な数のスキンがSwingのUIコミュニティーに現れることを期待したいものです。何百ものルックが選べるようになれば、Swingアプリケーションにつきものの「恐ろしく」「醜い」というレッテルは、永遠に消え去ることでしょう。

ダウンロード

内容	ファイル名	サイズ
Sample code	synth.jar	21KB

著者について

Michael Abernethy

Michael Abernethyは現在、WebSphere System Management Functional テスト・チームのリーダーとして働いています。以前は4年間WebSphere Servicesチームで働いており、WebSphere上の企業アプリケーションの作成や展開を行っていました。暇な時にはSwingやUI開発で遊んでいます。