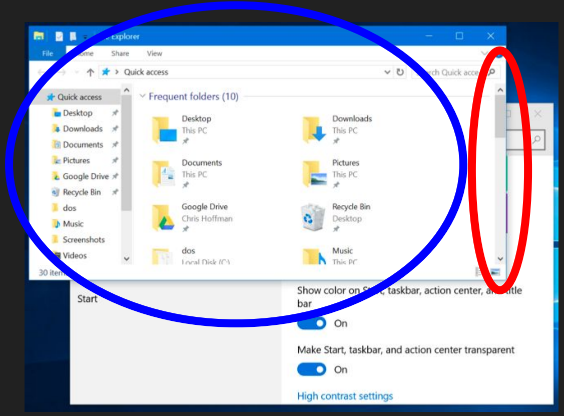
Abstract Factory Pattern

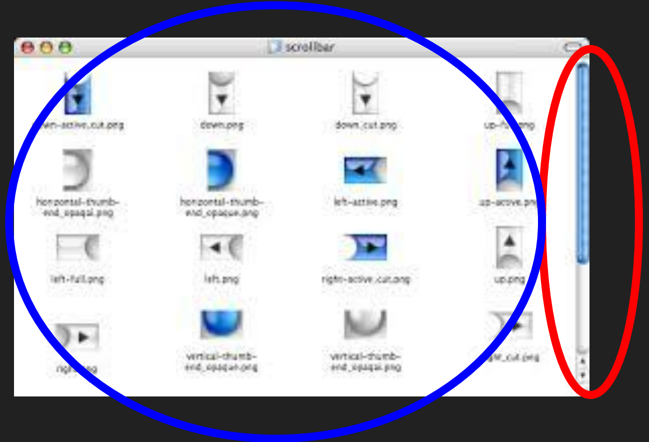
Creational, Object focused

Motivation

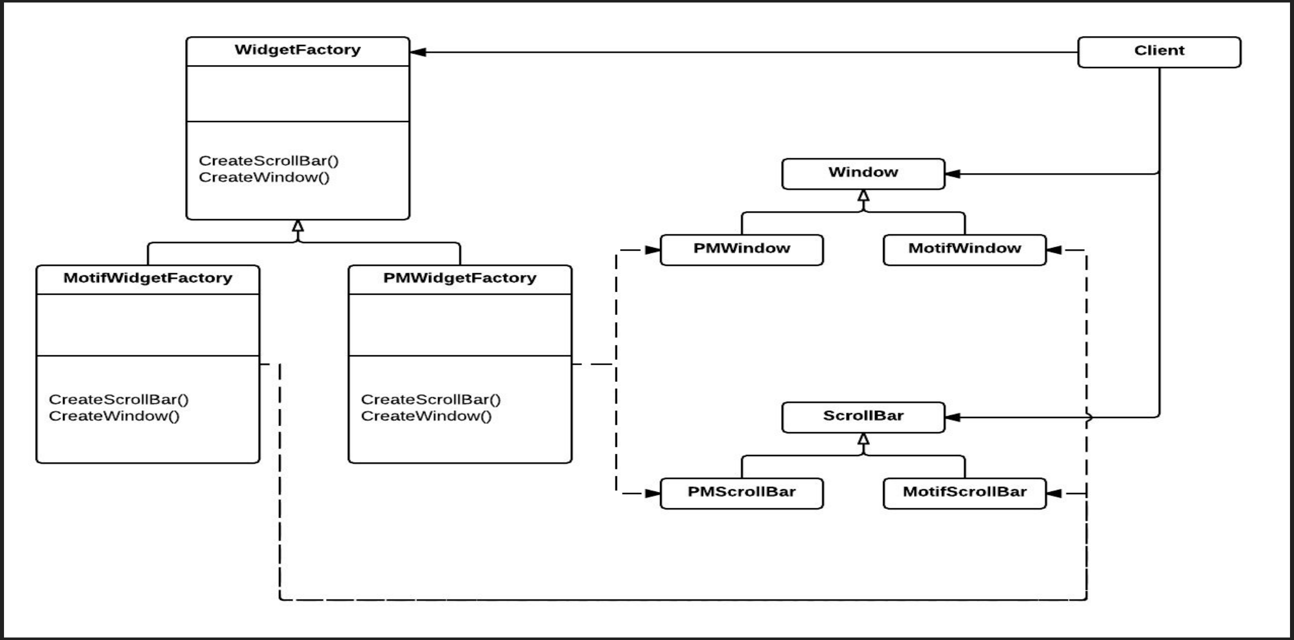
- Creating individual classes for every possible look-and-feel is intractable
- These classes would make it hard to change the look and feel later
- Think of a scroll bar
- Or the window itself



Show color on Start, taskbar, action center, and title bar
On
Make Start, taskbar, and action center transparent
On

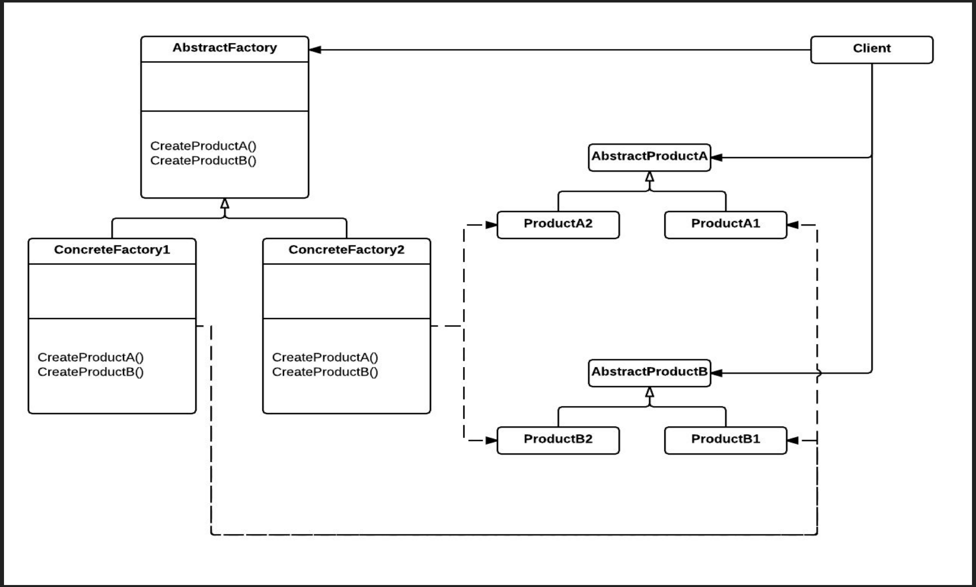


Motivation


WidgetFactory
CreateScrollBar()
CreateWindow()
Client
Window
PMWindow
MotifWindow
MotifWidgetFactory
CreateScrollBar()
CreateWindow()
PMWidgetFactory
CreateScrollBar()
CreateWindow()
ScrollBar
PMScrollBar
MotifScrollBar

Structure

- AbstractFactory
  - Declares interface for creating abstract product objects
- ConcreteFactory
  - Implements the interface for a type of product
- AbstractProduct
  - Declares interface for type of product
- ConcreteProduct
  - Defines a product object to be created
  - implements AbstractProduct interface
- Client



Consequences

- Pros:
  - *Isolates concrete classes.* Product class names are hidden from client code. Client interfaces through `AbstractProduct` interface
  - *Exchanging product families is easy*.
  - *Promotes consistency among products.*
- Cons:
  - *Supporting new kinds of products is difficult.* Requires redesigning the `AbstractFactory` class and all of its subclasses.

Example from “Design Patterns: Elements of Reusable Object-Oriented Software”

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides

Porting Lexi to new systems

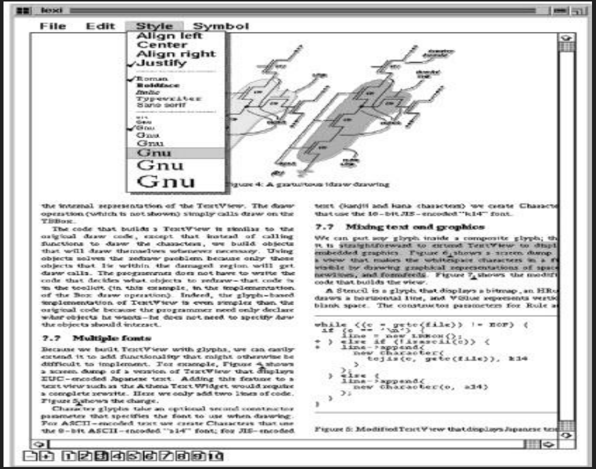


Figure 4: A gratuitous idraw drawing

the internal representation of the TextView. The draw operation (which is not shown) simply calls draw on the TBBox.

The code that builds a TextView is similar to the original draw code, except that instead of calling functions to draw the characters, we build objects that will draw themselves whenever necessary. Using objects solves the redraw problem because only those objects that lie within the damaged region will get draw calls. The programmer does not have to write the code that decides what objects to redraw—that code is in the toolkit (in this example, in the implementation of the Box draw operation). Indeed, the glyph-based implementation of TextView is even simpler than the original code because the programmer need only declare what objects he wants—he does not need to specify how the objects should interact.

7.? Multiple fonts

Because we built TextView with glyphs, we can easily extend it to add functionality that might otherwise be difficult to implement. For example, Figure 4 shows a screen dump of a version of TextView that displays EUC-encoded Japanese text. Adding this feature to a text view such as the Athena Text Widget would require a complete rewrite. Here we only add two lines of code. Figure 5 shows the change.

Character glyphs take an optional second constructor parameter that specifies the font to use when drawing. For ASCII-encoded text we create Characters that use the 8-bit ASCII-encoded "a14" font; for JIS-encoded text (kanji and kana characters) we create Characte[illegible] that use the 16-bit JIS-encoded "k14" font.

7.7 Mixing text and graphics

We can put any glyph inside a composite glyph; th[illegible] it is straightforward to extend TextView to displ[illegible] embedded graphics. Figure 6 shows a screen dump [illegible] a view that makes the whitespace characters in a f[illegible] visible by drawing graphical representations of spac[illegible] newlines, and formfeeds. Figure 7 shows the modifi[illegible] code that builds the view.

A Stencil is a glyph that displays a bitmap, an HRu[illegible] draws a horizontal line, and VGlue represents vertic[illegible] blank space. The constructor parameters for Rule a[illegible]

```
while ((c = getc(file)) != EOF) {
 if (c == '\n') {
     line = new LRBox();
+ } else if (!isascii(c)) {
+    line->append(
       new Character(
         tojis(c, getc(file)), k14
       )
     );
  } else {
     line->append(
       new Character(c, a14)
     );
  }
}
```

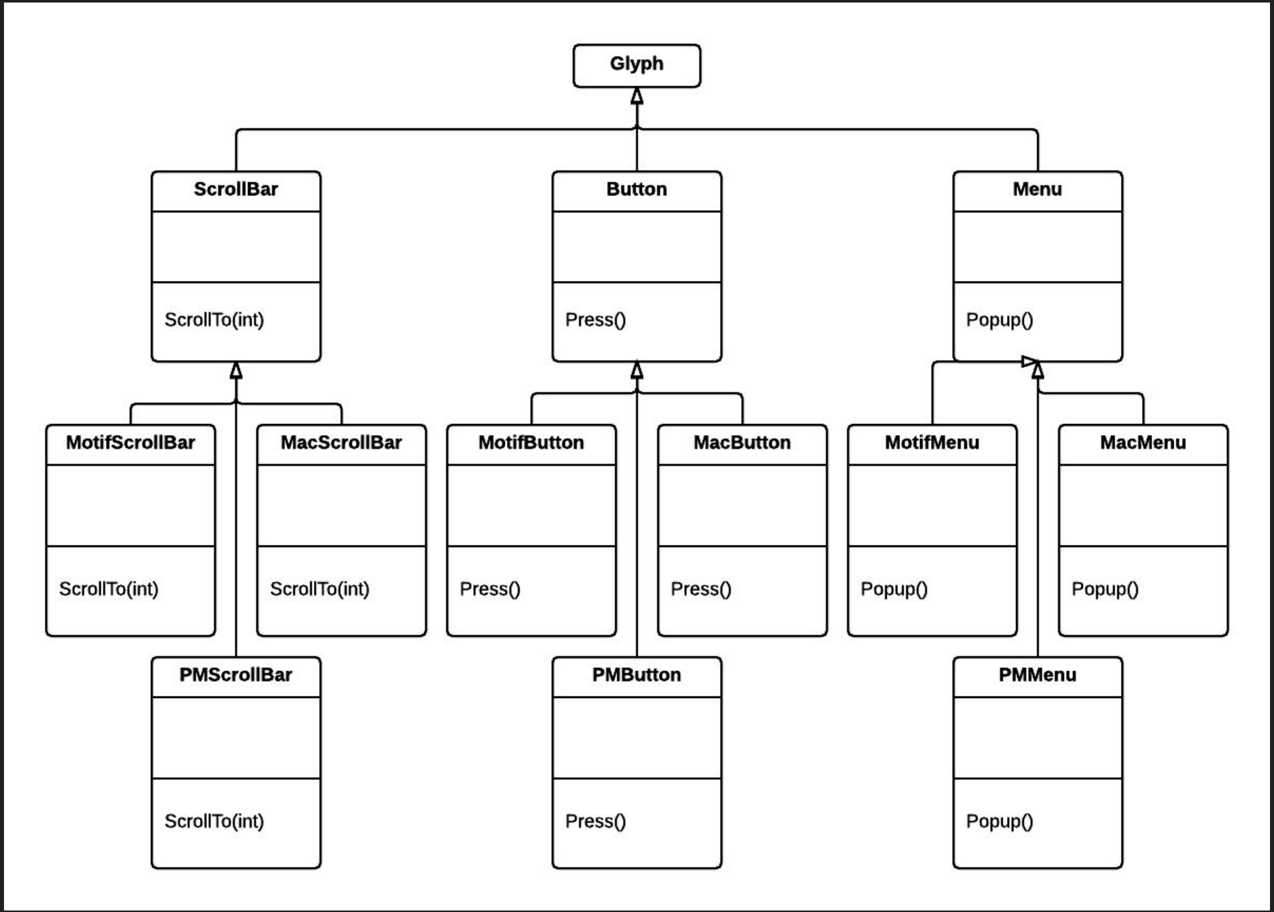
Figure 5: Modified TextView that displays Japanese te[illegible]

Supporting Multiple look-and-feel standards

- Problem:
    - Achieving portability across hardware and software platforms is a **major problem**
    - Retargeting Lexi to a new platform should not require a major overhaul
    - Adding support for new standards **should** be easy
    - Lexi's look and feel should be able to be dynamically changed

Assumptions

- Two sets of widget Glyph classes to implement multiple look-and-feel standards
  - Abstract Glyph subclasses for each category of widget Glyph
    - ScrollBar
    - Buttons
  - Concrete sublcasses for each abstract subclass (MacScrollBar, Win10Button)


Glyph
ScrollBar
ScrollTo(int)
Button
Press()
Menu
Popup()
MotifScrollBar
ScrollTo(int)
MacScrollBar
ScrollTo(int)
MotifButton
Press()
MacButton
Press()
MotifMenu
Popup()
MacMenu
Popup()
PMScrollBar
ScrollTo(int)
PMButton
Press()
PMMenu
Popup()

C++ Implementation

- Can’t hard code constructors into C++

  `MacMenu* mac_menu = new MacMenu();`
- Now if we change it to Motif

  `MotifMenu* motif_menu = new MotifMenu();`
- We have to change this for every instance:

  Find/Replace `mac_menu` => `motif_menu`
- This would also need to be done with all buttons and scroll bars

How can we generalize this?

C++ Implementation

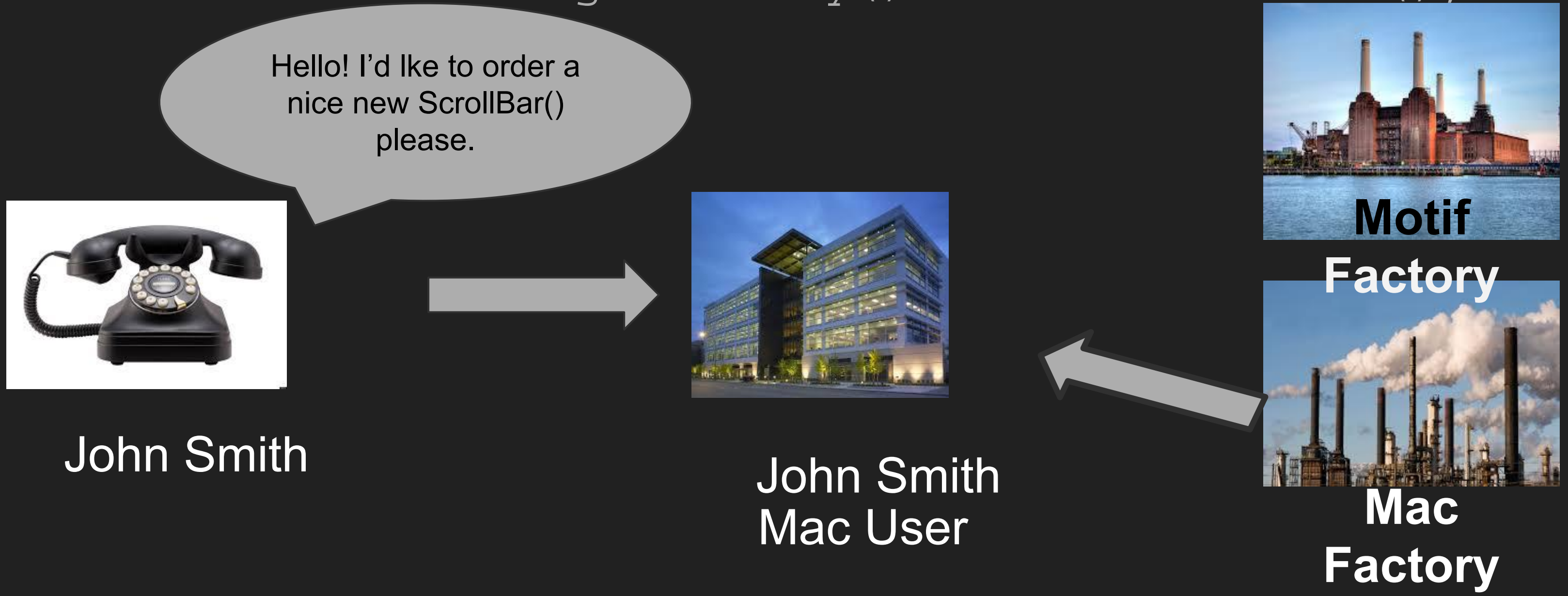
- Generalized to standardize the interface

```
ScrollBar* scroll_bar = new MacScrollBar();
```

C++ Implementation

- Generalized to minimize look-and-feel dependencies

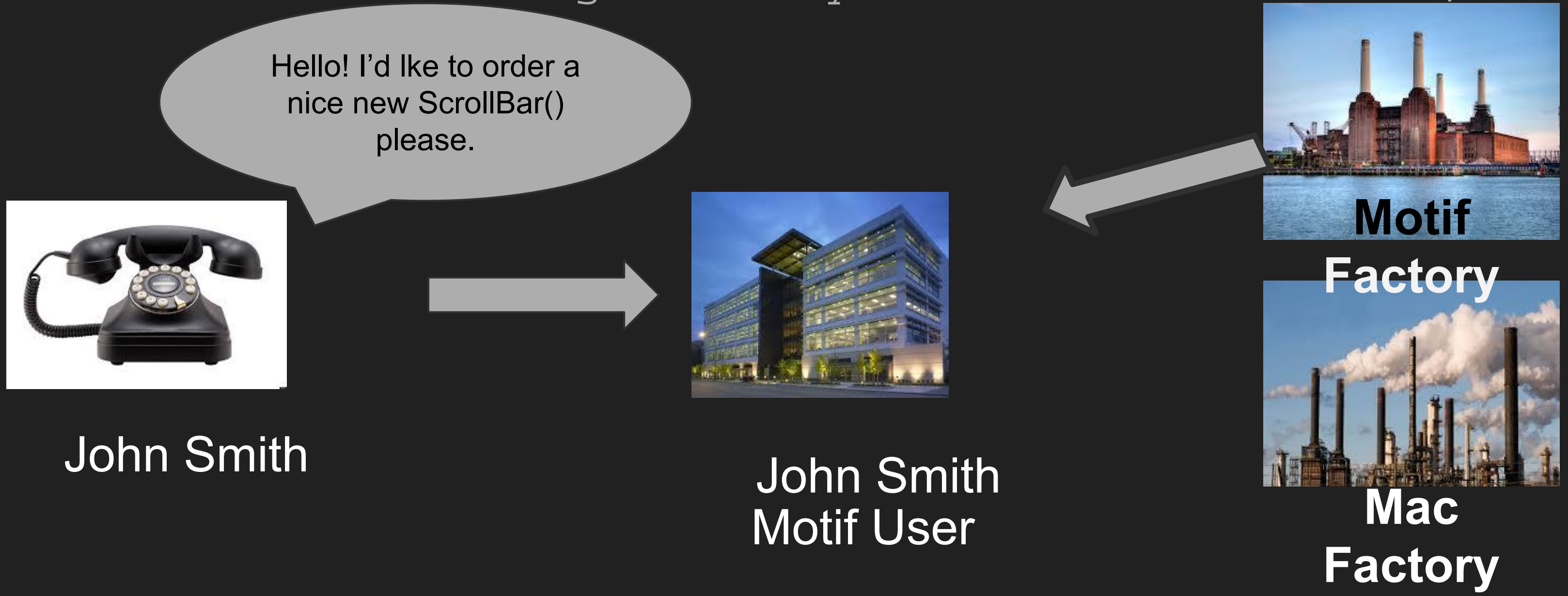
```
ScrollBar* scroll bar = guiFactory()->CreateScrollBar();
```



C++ Implementation

- Generalized to minimize look-and-feel dependencies

```
ScrollBar* scroll bar = guiFactory()->CreateScrollBar();
```



C++ Implementation

- Generalized to minimize look-and-feel dependencies

```
Button* button = guiFactory()->CreateButton();
```

Hello! I'd lke to order a nice new Button() please.
John Smith
John Smith Motif User
Motif Factory
Mac Factory

Run-time look-and-feel selection

```
GUIFactory* guiFactory;
const char* styleName = getenv("LOOK_AND_FEEL");
if(strcmp(styleName, "Mac") == 0) {
  guiFactory = new MacFactory;
} else if (strcmp(styleName, "Win10") == 0) {
    guiFactory = new Win10Factory;
} else {
  guiFactory = new DefaultGUIFactory;
}
```

Abstract Factory Pattern

- **Factories** and **products** are the key participants in the **Abstract Factory**
- This pattern captures how to create families of related product objects without instantiating classes directly
- It is most appropriate when the number and general kinds of product objects stay constant, and there are differences in specific product families
- We choose between families by instantiating a particular concrete factory and using it to create consistent products thereafter
- We can swap entire families by replacing the concrete factory
- **Abstract Factory** pattern emphasizes *families* of products