The Specification of the Sibling Design Pattern

Adam Brown[†] Richard Cardone[†] Sean McDirmid[‡] Calvin Lin[†]

[†]*Department of Computer Sciences*
The University of Texas at Austin
Austin, TX 78712 USA
{abrown, richcar, lin}@cs.utexas.edu

[‡]*School of Computing*
University of Utah
Salt Lake City, UT 84112 USA
mcdirmid@cs.utah.edu

Pattern Name and Classification

Sibling - Class Structural

Intent

Use inheritance between nested classes to increase code modularity in single-inheritance languages with nested types. Want code relating to a piece of functionality affecting several classes contained in one place.

Motivation

Suppose that we want to build a graphical user interface (GUI) library that allows us to include only those *features* that are needed. Possibilities for *features* include support for a color display, handling of keyboard events, and handling of mouse events. By allowing the classes in the library to contain only needed features, we avoid the exponential explosion of classes that occurs when all possible combinations of features must be embodied in different classes. Let us further suppose that we are doing this in a language which supports single-inheritance and *nested types*. A *nested type* is a type whose definition occurs inside the definition of another type.

Our overall goal for the GUI library is to increase code modularity. Ideally, all of the code implementing a feature resides in one place. One possible solution uses a class to represent the feature, which we refer to as the *outer class*, and nested classes, one for each widget type, to implement the changes to that widget for the feature. For example, a `Base` outer class would contain the basic funcationality for the different widgets. A `Color` outer class would provide support for drawing the widgets to a color display. `KeyEvent` and `MouseEvent` outer classes would add handling of keyboard and mouse input, respectively.

For the sake of simplicity, we will only look at two of the widgets in a typical GUI library, `Button`s and `Component`s. The `Component` widget type is the abstract superclass of all GUI widgets and allows both

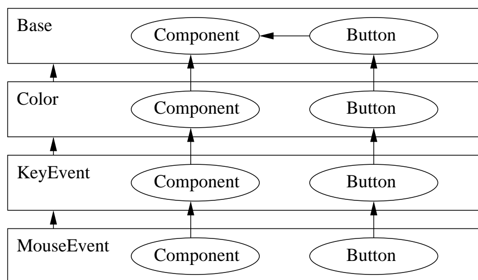


Figure 1: Incorrect Class Hierarchy

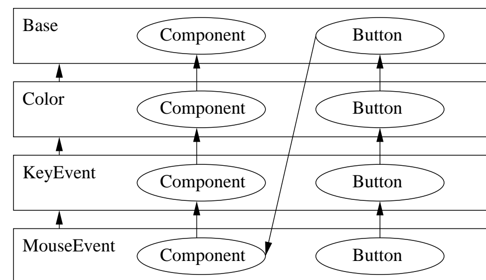


Figure 2: Sibling Pattern Hierarchy

application and library code to treat widgets generically. Since, `Button` is a widget type, it must inherit from `Component`. Naïvely, we might think that the class hierarchies for `Button` and `Component` are represented by Figure 1. However, in this hierarchy, the funcationality that is added to the `Component` hierarchy by outer classes other than `Base` is not available to the `Button` hierarchy. In fact, since no class inherits from a subclass of `Base::Component`, it is useless for any class to extend `Base::Component`.

The inheritance relationships that we want are shown in Figure 2. In this hierarchy, `Base::Button` derives from `MouseEvent::Component`. In this way, `Button` inherits all of the functionality added to the `Component` hierarchy by `Color`, `KeyEvent` and `MouseEvent`. The inheritance link between `Base::Button` and `MouseEvent::Component` represents the core idea of the *Sibling pattern*. The *Sibling pattern* occurs when a class (`Base::Button`) inherits from the most specialized subclass (`MouseEvent::Component`) of its *sibling*, a similarly nested class (`Base::Component`). By using Sibling, we can view each outer class as a *layer* adding functionality to the nested classes.

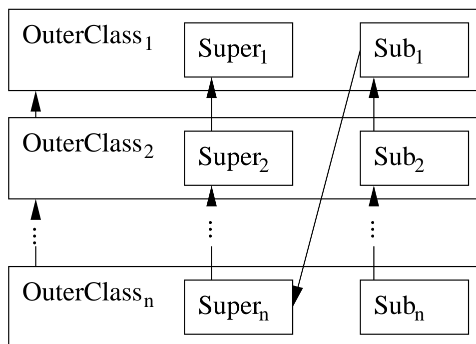
Applicability

The Sibling pattern is most applicable in languages with single-inheritance and support for nested types. The usefulness of Sibling is reduced in languages which provide multiple-inheritance which allows a class to derive from both its sibling in the same layer and the similarly named class nested in another layer. Additionally, Sibling is easiest to realize in languages which allow changes to the class hierarchy without changing class definitions [3, 9]

Use Sibling

- to modularize code for related classes when the changes made to a class should be inherited by classes in other hierarchies.
- when a large number of independent extensions are possible. If all combinations of features are possible, then an explosion of classes could occur. Sibling allows only those combinations which are used to be instantiated and ensures that the classes inherit the proper behavior for the combined features.

Structure



Participants

- OuterClass$_i$
  - contains the classes which are extended by the given layer.
- Super$_i$

- defines the changes to the superclass hierarchy at the given layer.

- Super_n
  - defines the most-derived type in the superclass hierarchy.
- Sub_1
  - defines the base type in the subclass hierarchy.
  - inherits from the most-derived type in the superclass hierarchy, namely Super_n.
- Sub_i
  - defines the changes to the subclass hierarchy at the given layer.

Collaborations

- Methods and fields in the class hierarchies are accessed according to language and inheritance semantics.
- Refinement is handled by performing actions before and after forwarding method calls up the inheritance hierarchy. Fields and methods may also be added to the class in a given layer.
- End-user code references Super_n and Sub_n. These classes exhibit the combined behavior for all combined features.

Consequences

The Sibling pattern offers the following benefits:

1. *Improves software engineering.* The Sibling pattern allows related changes to multiple classes to reside in one logical unit, namely the outer class. By combining the desired features, in their respective outer classes, we can obtain a specialized version of the class library with only those features required. Inheritance relations between nested classes in the same layer are resolved by the Sibling pattern.
2. *Multiple Inheritance.* As previously mentioned, similar behavior can be obtained by using multiple inheritance. The inheritance diagram in Figure 3 represents the equivalent possible using multiple inheritance. However, the Sibling pattern achieves this behavior without the complications that occur when a class inherits from a super-class along multiple paths.

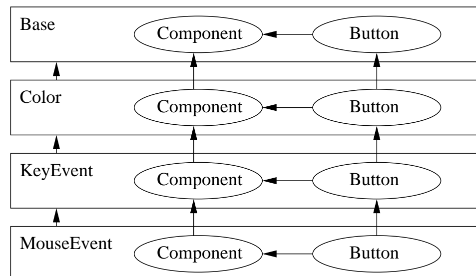


Figure 3: Multiple-inheritance Equivalent

The following issue must be considered when applying the Sibling pattern:

1. *Language Support.* The Sibling pattern requires that the language support nested types. Additionally, languages which only support fixed class hierarchies, that is languages which do not allow changes to the class hierarchy without changing the definition of classes, discourage the use of layers of nested classes. It is for this reason that the pattern is seldom seen in these languages.

Implementation

The Sibling pattern requires that the type of the leaf class in a hierarchy be available in classes that make up the hierarchy. The Sibling pattern can be implemented in OO languages that support nested classes by using a distinguished name for the leaf class in the hierarchy. This is the form that the code in the Sample Code section takes. Languages that support direct addressing of the leaf class, without using a distinguished name, provide greater flexibility in implementing the pattern. Some language techniques that provide this support include virtual types [9] and the **This** implicit type parameter in Java Layers [5, 2]. [7] demonstrates a way to use C++ templates along with a `Config` class to achieve a static reference to the leaf class.

Sample Code

The following code demonstrates an example use of the Sibling pattern in C++. We use the distinguished name form of the Sibling pattern here for clarity. This organization reduces the ease with which different combinations of features can be used in a library, because any changes in functionality would require changing the source code for one or more layers to update the inheritance hierarchy.

We continue the example of a GUI toolkit from the Motivation section. We start with the `Base` layer. The nested classes in the `Base` layer simply know how to draw the widget.

```
class Base {
  class Component {
    public:
      Component();
      virtual void Draw();
      // ...
  };
  class Button: public MouseEvent::Component {
    public:
      Button();
      virtual void Draw();
      // ...
  };
};
```

From Figure 2, we know that `Base::Button` must derive from `MouseEvent::Component`. The nested classes in the `Color` layer extend those in the `Base` layer to draw in color. The `Draw()` methods in `Color::Component` and `Color::Button` set the drawing pens to the correct color and then rely on the actual drawing code in their superclass.

```
class Color: public Base {
  class Component: public Base::Component {
    private:
      Color foreColor;
      Color backColor;
    public:
      Component();
      virtual void Draw() {
        setForeground(foreColor);
        setBackground(backColor);
        Base::Component::Draw();
      }
  };
  class Button: public Base::Button {
    private:
      Color buttonColor;
    public:
      Button();
      virtual void Draw() {
        setButtonColor(buttonColor);
        Base::Button::Draw();
      }
  };
};
```

`KeyEvent` extends the API of its nested classes by adding methods for event handling.

```
class KeyEvent: public Color {
  class Component: public Color::Component {
    public:
      Component();
      virtual void Handle(Event e);
  };
  class Button: public Color::Button {
    public:
      Button();
      virtual void Handle(Event e) {
        if ( e.type() == Key::Press && (Key e).key() == Key::Return )
          Push(e);
        Color::Button::Handle(e);
      }
      virtual void Push(Event e) {
        // perform action
      }
  };
};
```

`MouseEvent` enables `Component` and `Mouse` to handle a wider array of events, namely events generated by mice. Here we see that `MouseEvent::Button::Handle()` first checks to see if it can handle the generated event and then passes the event up to `KeyEvent`. In this way, an event can make it all the way up the class hierarchy to `KeyEvent::Component::Handle()`.

```
class MouseEvent: public KeyEvent {
  class Component: public KeyEvent::Component {
    public:
      Component();
      virtual void Handle(Event e);
  };
  class Button: public KeyEvent::Button {
    public:
      Button();
      virtual void Handle(Event e) {
        if ( e.type() == Mouse::Click )
          Push(e);
        KeyEvent::Button::Handle(e);
      }
  };
};
```

Finally, as a convenience, the nested classes of the leaf node in the outer class hierarchy have `typedefs` to *unnest* them and make them top-level entities.

```
typedef MouseEvent::Component Component;
typedef MouseEvent::Button Button;

Button button = new Button();
button->setBackColor(Color::Red);
```

Known Uses

Our initial discovery of the Sibling pattern occurred during the design of the Fidget GUI library [4]. The running example throughout this paper borrows from the design and implementation of Fidget.

The Jakarta Tool Suite (JTS) is a generator for domain-specific languages [1]. JTS is written in a variant of Java, called Jak, and uses the distinguished name form of the Sibling pattern.

Jiazzi [8] is a system for developing software components from existing Java classes. The *Open Class pattern* that Jiazzi supports is similar in structure and usage to the Sibling pattern. The open class pattern is a way for Jiazzi to simulate open classes [6], which allow existing classes to be updated with new methods without updating the classes' source code.

Related Patterns

The Sibling design pattern is a specific instance of the Most-Specialized Type in a Hierarchy Design Pattern [9].

References

[1] Don Batory, Bernie Lofaso, and Yannis Smaragdakis. JTS: tools for implementing domain-specific languages. In *Proceedings Fifth International Conference on Software Reuse*, pages 143–53, Victoria, BC, Canada, 2–5 1998. IEEE.

[2] Richard Cardone. *Language and Compiler Support for Mixin Programming*. PhD thesis, The University of Texas at Austin, 2002.

[3] Richard Cardone, Don Batory, and Calvin Lin. Java Layers: Extending Java to support component-based programming. Technical Report TR2000-11, Dept. of Computer Sciences, The University of Texas at Austin, June 2000.

[4] Richard Cardone, Adam Brown, Sean McDirmid, and Calvin Lin. Using mixins to build flexible widgets. In *Aspect-Oriented Software Development*, April 2002.

[5] Richard Cardone and Calvin Lin. Static virtual types in Java Layers. Technical Report CS-TR-00-25, The University of Texas at Austin, 2000.

[6] Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *OOPSLA 2000 Conference on Object-Oriented Programming, Systems, Languages, and Applications, Minneapolis, Minnesota*, volume 35(10), pages 130–145, 2000.

[7] Krzysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications.* Addison-Wesley, 1st edition, 2000.

[8] S. McDirmid, M. Flatt, and W. Hsieh. Jiazzi: New-age components for old-fashioned Java. In *Proc. of OOPSLA*, October 2001.

[9] Kresten Krab Thorup. Genericity in Java with virtual types. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP '97 — Object-Oriented Programming 11th European Conference, Jyväskylä, Finland*, volume 1241, pages 444–471. Springer-Verlag, New York, NY, 1997.