Pattern Hatching

Subject-Oriented Design

John Vlissides
C++ Report, February 1998

I've said it before and I'll say it again: A hallmark—if not *the* hallmark—of good object-oriented design is that you can modify and extend a system by adding code rather than by hacking it. In short, change is *additive*, not *invasive*. Additive change is potentially easier, more localized, less error-prone, and ultimately more maintainable than invasive change. Indeed, a system that requires invasive change may not be changeable at all when you don't or can't have the source.

Most of the patterns in *Design Patterns*[1] can help make invasive change unnecessary, or at least less necessary. STRATEGY, for example, lets you replace one algorithm with another. You can change the class that's instantiated by substituting one prototype for another (see PROTOTYPE). To express a grammar flexibly and extensibly, use INTERPRETER. Table 1.2 in the book lists a design aspect you can change noninvasively for each one of our 23 patterns.

You can't always get what you want

All well and good, except for one big assumption—that the right patterns have been designed into the software in the first place. If not, the going gets tough.

Some patterns— DECORATOR and ADAPTER come to mind—do support extension after the fact. But they can't work miracles. In particular, it's devilishly hard to change a system's behavior when you can't control which classes get instantiated. For a decorator or adapter to work, you must have something to decorate or adapt, right? If that something lives deep in the bowels of a design, it may be difficult or impossible to get at. Your only hope then is to trick the system into instantiating classes of your own making—classes that can adapt, decorate, or otherwise enhance existing ones.

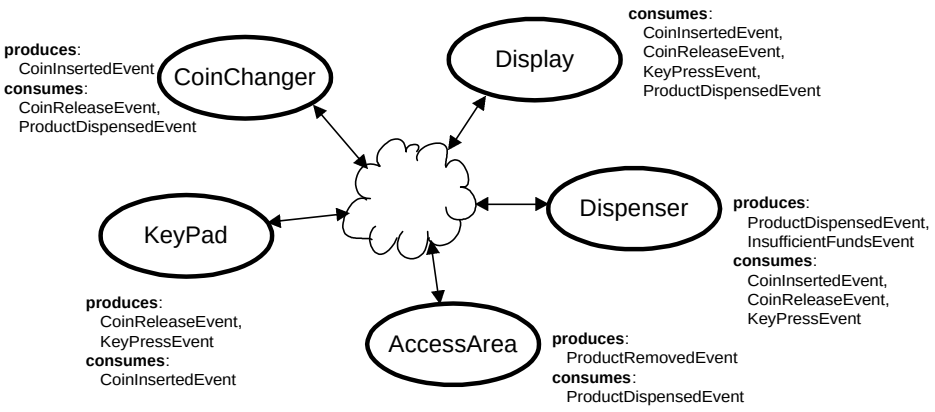

Figure 1

Consider again the vending machine example we've been looking at.[2,3,4] Figure 1 shows the classes that represent and control the various subsystems in the vending machine, along with their interconnections.

Now assume that the system was designed naïvely, with no patterns in mind. All connections between subsystem classes are hard-wired. That means, for example, that the Display class keeps explicit references to producers of events it cares about (CoinInsertedEvent, CoinReleaseEvent, KeyPressEvent, and ProductDispensedEvent). The implementation might look like this:

```
class Display {
public:
    virtual void show();
    virtual void clear();
    // ...
protected:
    virtual void beep();  // helper function called by show() and clear()
private:
    CoinChanger* _changer;
    Dispenser* _dispenser;
    KeyPad* _keyPad;
};
```

In such a hard-wired world, the CoinChanger, Dispenser, and KeyPad classes would in turn keep a reference to a `Display` so that they know whom to deliver events to. All other connections would have similar implementations.

The trouble with this approach of course is the proliferation of connections that, once implemented, are not easily changed. We're forced to perform extensive surgery on the system to add, say, a BillChanger class. `Display` would need modification to add a reference to a BillChanger object. One or more operations would have to change, too. The same is true of any class that is currently interested in CoinInsertedEvents.

What if we get creative and effect these changes with a decorator?

```
class DisplayDecorator {
public:
    DisplayDecorator (Display* d) { _display = d; }

    virtual void show () {
        // enhanced or extended implementation
    }
    virtual void clear () { _display->clear(); }
    // ... (assume only show() needs to change)
 private:
    Display* _display;
    BillChanger* _billChanger;
};
```

Such a decorator might do the trick, provided we *can* decorate the `Display` instance(s) in the system. In a perfect world, we could easily and noninvasively transform every constructor call `new Display()` into `new DisplayDecorator(new Display)` and—*shazam!*—displays know about bill changers.

In reality, however, this transformation is likely to require edits to existing code. Like the interconnections, the constructor call is almost certainly hard-wired, probably deep down in one or more method implementations. The type of display to create has to have been parameterized somehow, perhaps using C++ templates or a creational pattern like PROTOTYPE or FACTORY METHOD. Without some kind of parameterization, there's no way to control what gets created—and DECORATOR needs at least that much to work its magic. The same is true of ADAPTER and any other pattern I can foresee applying to this problem.

Mightier magic

Suspend disbelief for a moment and suppose you could extend `Display` by first specifying the extensions…

```
class Extensions {
public:
    virtual void show () {
        // enhanced or extended implementation
    }
 private:
    BillChanger* _billChanger;
};
```

…and then expressing how the extensions affect the original `Display` class:

```
override (class Display, (Display, Extensions));
```

Here, `override` means, "Combine existing classes (in the inner parentheses) to form a new class named by the first parameter." In this case we're combining the existing `Display` and `Extensions` classes to form a new, "composed" class that happens to be called `Display`. This new class will contain all the members of both `Extensions` and the original `Display`.

The second parameter denotes a list of classes, and the order in which they appear in the list is significant. If two or more of the classes define a member with the same name, then the composed class will wind up with the implementation of the class appearing *latest* in the list. In our example, the new `Display` will duplicate all of the original's member functions except `show()`, which will come from `Extensions`. As for the member variables, there's no name conflict between `Extensions` and the original `Display`. Therefore the composed class will incorporate every member of both classes—`_changer`, `_dispenser`, `_keyPad`, and `_billChanger`—all declared `private`.

The semantics here are a lot like standard C++ multiple inheritance, with three important differences:

1. Any reference to `Display` outside these classes will refer to the composed `Display` class.
2. Anything accessible to `Extensions` or to the original `Display` is also accessible to the composed `Display`, including private members.
3. `override` is just one of many possible ways to compose classes.

Subjectivity

Clearly this ain't vanilla C++. So what is it?

It's called "subject-oriented programming" (SOP), and it's the brainchild of Bill Harrison and Harold Ossher of IBM Research.[5,6] Their idea is to augment object-oriented languages with facilities for packaging code in interesting ways. Here are some of the problems SOP can address:

- A logically atomic object is implemented as a composition of objects, and the implementation shows through in undesirable ways. A good example is the classic "self problem," which crops up in the implementation of several of our patterns. DECORATOR, for one, can be a fine alternative to subclassing—as long as you realize that decorators work only for clients. Take the decorator code for the vending machine. Extending `beep()` by overriding it in `DisplayDecorator` is an exercise in futility. That's because clients can't call `beep()` (it's `protected`), and `Display` operations that call `beep()` will get `Display::beep()`, not `DisplayDecorator::beep()`. The "self problem" can rear its head when you use PROXY and ADAPTER, too.
- You want to avoid revealing concrete types to clients, preferring to have them deal with abstract interfaces exclusively. You are using ABSTRACT FACTORY, for example, whose intent is to free clients from specifying ConcreteProduct types. Or it's BRIDGE you're using, and you want to avoid exposing the client to Implementor classes.
- You want to extend classes noninvasively, as in the vending machine example. Perhaps you want to extend an interface without changing it, as is sometimes the case with the VISITOR pattern. Or

you've applied SINGLETON, and now you want to subclass the Singleton class. How do you make the static `Instance` operation instantiate this new subclass without changing the base class?

- There are several ways to partition your code depending on different needs. You might partition it one way to spread the work among your development team. You'd do it another way to accurately model your problem domain. Yet another partitioning would make the system more amenable to tool support, say, for third-party extension or end-user customization.

The model

What exactly is SOP? Conceptually, it's not much beyond what we already know and love about objects. A subject-oriented program has two parts:

1. Two or more **subjects**, each nothing more than a collection of class declarations.
2. One or more **composition rules**, which describe how to combine the subjects into instantiable classes or new subjects.

In the vending machine example, both the original `Display` class and the `Extensions` class are subjects. The `override` composition rule combines them to form the `Display` class that clients will end up using.

Composition rules are arguably *the* key mechanism in SOP. There are two basic kinds:

1. **Correspondence rules** specify what names mean in the composed subject. For example, suppose `Extensions` defines `beep()`. Does that mean *all* calls to `beep()` resolve to the `Extensions` version, or only client calls? Correspondence rules let you specify either case with ease.
2. **Combination rules** specify how to implement members in the composed subject. The `override` rule ensures that the composed `Display` subject gets its implementation of `show()` from the `Extensions` subject. Alternatively, you can say you want the new `show()` to be a sequential combination of `Extensions::show()` and the original `Display::show()`. To do that with DECORATOR, you'd have to code it explicitly.

The most useful composition semantics are achieved using a correspondence rule coupled with a combination rule. This has been institutionalized in a third set of composition rules called **correspondence-combination rules** (an unfortunate name for the most prevalent rules).

Anyway, it turns out that `override` is in this third set, being a combination of the `equate` correspondence rule and the `replace` combination rule. Once you `equate` classes, all calls to their operations will resolve to the composed subject's operations (thereby avoiding the "self problem" in the decorator example). Meanwhile, `replace` makes it clear that the composed subject should adopt the implementations of the *latest* subject in the rule's composition list. The other common rule of this ilk is `merge`. That's the same as saying `equate` and `join`, where `join` is the combination rule that yields sequential combinations of operations.

At minimum, a SOP-enabled programming environment will enforce these and other composition rule semantics statically, much as conventional compilers enforce the static semantics of C++. A more advanced environment would also support browsing, debugging, and revision control of subjects and composition rules just like current environments do for classes.

Subjects in action

So much for the basics. Now let's consider how SOP can help us avoid invasive change, either as a replacement for or an enhancement of a pattern-derived implementation.

We've already seen how we can enhance an existing class by combining two subjects to form a third with the `override` rule. We're about to use a similar technique in a subject-oriented implementation of ABSTRACT FACTORY. Our goal is to avoid any mention of concrete factory or product classes in client code.

To implement ABSTRACT FACTORY conventionally, you define the AbstractFactory and AbstractProduct interfaces…

```
class AbstractProductA;
class AbstractProductB;
// ...

class AbstractFactory {
public:
    virtual AbstractProductA* createProductA();
    virtual AbstractProductB* createProductB();
    // ...
};
```

…along with the families of ConcreteFactory and ConcreteProduct classes:

```
/**
 * Family #1
 */
class ConcreteProductA1 : public AbstractProductA { /* ... */ }
class ConcreteProductB1 : public AbstractProductB { /* ... */ }
// ...
class ConcreteFactory1 : public AbstractFactory {
public:
    virtual AbstractProductA* createProductA () {
        return new ConcreteProductA1();
    }
    virtual AbstractProductB* createProductB () {
        return new ConcreteProductB1();
    }
    // ...
};

/**
 * Family #2
 */
class ConcreteProductA2 : public AbstractProductA { /* ... */ }
class ConcreteProductB2 : public AbstractProductB { /* ... */ }
// ...
class ConcreteFactory2 : public AbstractFactory {
public:
    virtual AbstractProductA* createProductA () {
        return new ConcreteProductA2();
    }
    virtual AbstractProductB* createProductB () {
        return new ConcreteProductB2();
    }
    // ...
};
```

The challenge here is to keep client code from ever mentioning a class with a “1” or a “2” in its name. Otherwise you expose a concrete class to the client. Then you’d have to change the client when you define a new kind of concrete factory or product.

Hiding concrete products from the client is easy enough—the factory does it for us *de facto*. Clients deal with abstract classes almost exclusively, since the `AbstractFactory` interface refers to *nothing but* abstract products. The trouble lies in that little word, “almost.” There is one time when a concrete class needs mentioning: when you instantiate the concrete factory. But you don’t want to put code like `new ConcreteFactory2()` in the client.

While there are any number of conventional workarounds to this problem (making the AbstractFactory class an extensible Singleton, for example[1]), SOP offers a particularly simple alternative. To instantiate an abstract factory, we'll make it so that clients always write

```
AbstractFactory* factory = new AbstractFactory();
```

no matter which ConcreteFactory class they end up using. We specify that class at compile-time in a composition rule. For example, to make this code create instances of `ConcreteFactory2`, we tell our SOP-enabled compiler to `equate` the concrete factory and the abstract factory:

```
equate (class AbstractFactory, (AbstractFactory, ConcreteFactory2));
```

Thus the compiler will ensure that every explicit call to the `AbstractFactory` constructor will be transformed, type-safely, into a call to `ConcreteFactory2::ConcreteFactory2()`. Simple as that.

Interface extension

Now let's see how subjectivity can address another common problem with ABSTRACT FACTORY, that of extending the AbstractFactory interface to handle new kinds of products. I know of no statically type-safe workaround to this problem using straight C++. SOP, however, makes it pretty easy.

Let's assume the new category of product classes looks like this:

```
class ConcreteProductC1 : public AbstractProductC { /* ... */ }
class ConcreteProductC2 : public AbstractProductC { /* ... */ }
```

To extend the factory classes to handle these products, just `merge` each factory class with a subject that extends its implementation and/or interface:

```
class ExtensionC {   // extension for AbstractFactory
public:
    virtual AbstractProductC* createProductC();
};

class ExtensionC1 {  // extension for ConcreteFactory1
public:
    virtual AbstractProductC* createProductC () {
        return new ConcreteProductC1();
    }
};

class ExtensionC2 {  // extension for ConcreteFactory2
public:
    virtual AbstractProductC* createProductC () {
        return new ConcreteProductC2();
    }
};

/**
 * Composition rules
 */
merge (class AbstractFactory, (AbstractFactory, ExtensionC));
merge (class ConcreteFactory1, (ConcreteFactory1, ExtensionC1));
merge (class ConcreteFactory2, (ConcreteFactory2, ExtensionC2));
```

Spanning the gap

As a final example, consider how you might use subjects to address a code-partitioning problem. A while back in these pages I presented a fairly fleshed-out pattern called GENERATION GAP.[7] At issue was how to

separate generated code from hand-modifications so that regenerating the code wouldn't force you to reapply the modifications.

The motivating example in that pattern described a GUI builder that generated the code for an alarm clock widget. The builder implemented all the visual aspects of the widget but almost none of the behavioral aspects. Those had to be implemented by hand-modifying the generated code. GENERATION GAP offered an effective if nontrivial way to insulate the changes from the generated code, making it unlikely that regeneration would require reimplementing anything. The old-timers among you will recall the artfulness needed to treat this problem with standard OO techniques, including changes to the code generator to make it pattern-aware. SOP allows a more mundane approach.

Going back to the motivating example, the builder is free to generate a class that implements the clock widget without following any special conventions:

```
class Clock : public Widget {
public:
    Clock (const char*);

protected:
    Interactor* Interior();

    virtual void SetTime();
    virtual void SetAlarm();
    virtual void Snooze();

protected:
    Picture* _clock;
    SF_Polygon* _hour_hand;
    SF_Rect* _min_hand;
    Line* _sec_hand;
};
```

The programmer may then specify hand-modifications in a separate subject…

```
class HandModifications {
public:
    HandModifications();

    void Run();

    virtual void SetTime();
    virtual void SetAlarm();
    virtual void Snooze();

    virtual void Update();
private:
    void GetSystemTime(int& h, int& m, int& s);
    void SetSystemTime(int h, int m, int s);
    void Alarm();
private:
    float _time;
    float _alarm;
};
```

…and finally compose this subject with the generated subject using a composition rule—`override` in this case:

```
override (class Clock, (Clock, HandModifications));
```

Sometimes it may be more appropriate to use `merge` to compose the subjects. At other times you'll want to merge some operations and override others. You may find yourself reaching for the more esoteric composition rules on occasion.

The beginning

Whatever the need, subject-oriented programming facilities bring a whole new dimension to software design. There are potentially many problems that lend themselves to subject-oriented solutions. What's important to realize about SOP is that it's just a mechanism, not a solution by itself. You must understand how to apply it to the problems you face as a software designer. It's a new mechanism too, one we've just begun to explore. In time, SOP will probably engender idioms and patterns of its own. For now there's plenty to learn about how subjects can help us implement, enhance, and perhaps even obviate some of today's design patterns.

Acknowledgments

Peri Tarr and Harold Ossher schooled me ever so patiently in the ways of SOP. Thanks you two—and congratulations to the team for making subjectivity real!

References

[1] Gamma, E., R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, MA, 1995.
[2] Vlissides, J. "Type Laundering," *C++ Report*, February 1997.
[3] Vlissides, J. "Latter-Day Events," *C++ Report*, June 1997.
[4] Vlissides, J. "Multicast," *C++ Report*, September 1997.
[5] Harrison, W. and H. Ossher. "Subject-Oriented Programming (A critique of Pure Objects)," *Conference Proceedings*, published as ACM SIGPLAN Notices, 28(10):411–428, October 1993.
[6] IBM. *Subject-Oriented Programming.* http://www.research.watson.ibm.com/sop.
[7] Vlissides, J. "Generation Gap," *C++ Report*, November/December 1996.