Working with Classes and Interfaces

Dirk Riehle

SKYVA International
www.skyva.com, www.skyva.de

riehle@acm.org, www.riehle.org

Classes are fundamental to object-oriented design and programming in C++. In this article, we take a look at five fundamental patterns of designing and using classes. We use a story, the evolution of a seemingly simple class, to illustrate the Simple Class, Design by Primitives, Interface Class, Abstract Base Class, and Narrow Inheritance Interface patterns. This story and the ensuing discussion provide us with some insight on what makes up a pattern and a good description thereof.

1 Name example

Consider the following example: we frequently give names to objects so that we can store the objects under these names and retrieve them later. Names may be as simple as a single string, and they may be as complex as a multi-part URL. Names we frequently use are class names, file names, and URLs. Such names typically consists of several parts, called name components. The name components of the file name “~/cpp/source/Main.C” are “~”, “cpp”, “source”, and “Main.C”. Generally speaking, we can view a name as a sequence of name components.

In our example, we represent each name as a Name object and each name component as a string object.

Name objects do not exist without a purpose: they are always part of a naming scheme. We can manage objects named by a Name object by using an appropriate naming scheme. For example, viewing a name as a sequence of name components lets us manage named objects in a hierarchical fashion, which is the most common (and convenient) management scheme I know of. Again, examples of Name objects that are interpreted hierarchically are class names (with possible namespaces), file names, and URLs.

For example, to look up the file “~/cpp/source/Main.C”, the file system first resolves “~” to your (Unix) home directory, then searches for a directory called “cpp”, followed by searching for a directory called “source”, and so on. This is a recursive descent into a directory hierarchy. Whether we represent the original name as a sequence of components or not, the lookup algorithm requires these name components one after another.

So here we go.

2 Simple Class

You decide you need Name objects to represent file names and other multi-part names. A simple thing that does the job is a single class with a member variable that holds the sequence of strings. A convenient representation of the sequence of strings is a vector object of strings.

Dirk Riehle. “Working with Classes and Interfaces.” C++ Report, March 2000.

Listing 1 provides a snapshot of this class. The member variable fComponents holds the vector of strings. The prefix ‘f’ stands for field and is a common convention I use for all non-static member variables of a class [4].

```
class Name {

protected:

    vector<string> fComponents;

public:

    // get name component
    virtual string component(int index) {
        ASSERT((i >= 0) && (i < fComponents.size()));
        return fComponents[index];
    }

    // set name component
    virtual void component(int index, string component) {
        ASSERT((i >= 0) && (i < fComponents.size()));
        fComponents[index]= component;
    }

    // insert name component
    virtual void insert(int index, string component) {
        ASSERT((i >= 0) && (i < fComponents.size()));
        fComponents.insert(fComponents.begin()+index, component);
    }

    // prepend name component
    virtual void prepend(string component) {
        insert(0, component);
    }

    // code for append, remove, etc. functions
    ...

    // code for constructors and destructor
    ...
};
```

Listing 1: Snapshot of initial Name class.

Designing and implementing the Name class is trivial. Still, there is a pattern behind it, which I call Simple Class. It is displayed as Pattern 1.

Name:	Simple Class.
Problem:	You need to design and implement a concept.
Context:	One implementation is sufficient, no other is needed. Changes to the implementation may affect clients You want to make it as simple as possible, but not simpler.
Solution:	Implement the concept as a single class.

Pattern 1: Thumbnail of Simple Class pattern.

Does Simple Class deserve the status of patternhood? After all, it is very simple and obvious.

The decision to call Simple Class a pattern depends on its relationship with other patterns. In the following sections, we will see further patterns like Interface Class and Abstract Base Class. When we compare Simple Class with these patterns, we see that designing and implementing a domain concept as a Simple Class is truly a pattern, because it is a proper abstraction from a recurring solution to a problem in a context.

Before we continue this discussion, let’s take a look at another pattern of good class design.

3 Design by Primitives

While implementing the Name class, you find yourself writing the same code repeatedly. You repeat code that checks for a valid index, you repeat code that puts a string as a name component into the vector, etc. Naturally, as a lazy developer (which is to say as a good developer [5]), you start moving these repeated code fragments into member functions of their own.

Listing 2 shows some member functions that come into being this way.

```
class Name {

public:

    // primitive function: returns number of components in name
    virtual noComponents() {
        return fComponents.size();
    }

    ...

protected:

    // primitive function: asserts that given index is valid
    virtual void assertIsValidIndex(int i) {
        assertIsValidIndex(i, noComponents());
    }

    // primitive function: asserts that given index is valid
    virtual void assertIsValidIndex(int i, int upperLimit) {
        ASSERT((i >= 0) && (i < upperLimit));
    }

    // primitive function: returns name component at given index
    virtual string basicComponent(int i) {
        return fComponents[i];
    }

    // primitive function: inserts name component at given index
    virtual void basicInsert(int index, string component) {
        fComponents.insert(index, component);
    }

    ...
}
```

Listing 2: A set of primitive member functions.

All member functions of Listing 2 are so-called primitive functions because they do exactly one well-defined thing. Please notice that “basicComponent” and “basicInsert” do not check whether the index passed in is actually a valid index. They rely on the calling context to ensure this precondition. As a consequence, most primitive member functions are protected so that they can be used only from inside the object.

Let us examine how we work with primitive functions. Listing 3 shows two examples.

```
virtual string component(int index) {
    assertIsValidIndex(index);
    return basicComponent(index);
}

virtual void insert(int index, string component) {
    assertIsValidIndex(index, noComponents()+1);
    basicInsert(index, component);
}

...
```

Listing 3: Working with primitive member functions.

The “component” and “insert” functions compose the primitive functions to do their task. More complex functions like “contextName” (which assembles all name components except for the last one in one new name), or “asDataString” (which assembles all name components into a single string of a storable format), use these primitive functions as well.

Hence, we can distinguish primitive member functions from the more complex functions that use them. Typically, the more complex functions provide the bulk of useful object functionality to clients. A set of primitive member functions is well-chosen, if the more complex member functions can easily use and compose them.

Designing a class using primitive member functions is called *Design by Primitives*, and it is a common pattern of good class design. It is displayed as Pattern 2.

Name:	Design by Primitives.
Problem:	You need to implement a class.
Context:	You expect to evolve the class. You want it to be easy to add new member functions. You want to avoid a fragile class in which changes to a function affect too many other functions. You want to make it as simple as possible, but not simpler.
Solution:	Separate more complex non-primitive member functions from primitive member functions. Determine the primitive member functions that best help implement the class. Implement non-primitive member functions using primitive member functions.

Pattern 2: Thumbnail of Design by Primitive pattern.

Design by Primitives is even more important in the context of the Abstract Base Class and Narrow Inheritance Interface patterns. As we will see, Design by Primitive is a precondition for reusing classes easily through inheritance.

4 Interface Class

The Name class turns out to be a popular class. Not only can you represent file names with it but also class names, Internet domain names, even URLs. The Name class is all over the place. Because it is so easy to use, it is used a lot, and there are plenty of Name objects at runtime. Profiling your applications tells you that Name objects are outnumbering most other objects (except, perhaps, for string and a few others). This gets you thinking about how to reduce memory consumption of Name objects.

Obviously, we can improve the Name implementation by representing a name as a single string that contains all name components. That way we get rid of the vector object and reduce memory consumption of Name objects. For example, a string representing the Name “~/cpp/source/Main.C” looks like “~#cpp#source#Main.C” using ‘#’ as a delimiter char (much like ‘/’ and ‘\’ are traditional delimiter characters for file names). A name component gets enclosed between ‘#’ delimiter chars or the start or end markers of the string.

But this approach has its downside: unless you start storing additional information, accessing a name component of a Name object is much slower than before. You have to search through the string until you reach the desired index position. Then you have to create a name component string before you can return it to the client. Thus the string-based Name class may be more memory-efficient than the vector-based Name class, but it is also slower.

So you actually need both classes. You rename the existing Name class to VectorName and implement a new class based on the string scheme and call it StringName. This lets you choose whichever class you need. However, you also want to use StringName and VectorName objects interchangeably. A client of your Name class does not want to write its code twice just to deal with different classes that do effectively the same thing.

Thus, you decide to use an *Interface Class*.

An interface class is a class that consists solely of pure virtual member functions.

In our example, you introduce an interface class called Name. It declares all the functionality that is common to both StringName and VectorName as pure virtual member functions. You then make StringName and VectorName inherit from Name and make them implement the pure virtual member functions. Using the Name interface class, clients can now work with StringNames and VectorNames without committing to any one of these two subclasses.

Listing 4 shows the Name interface class and how StringName and VectorName inherit from it.

```
class Name {
public:
    virtual string asString() =0;
    virtual string asDataString() =0;
    virtual string component(int i) =0;
    virtual Iterator components() =0;
    virtual char delimiterChar() =0;
    virtual char escapeChar() =0;
    virtual bool isEmpty() =0;
    virtual bool isEqual(Name* name) =0;
    virtual int noComponents() =0;
    virtual void append(string component) =0;
    virtual void component(int index, string component) =0;
    virtual GenericName* contextName() =0;
    virtual string firstComponent() =0;
    virtual void insert(int index, string component) =0;
    virtual string lastComponent() =0;
    virtual void prepend(string component) =0;
    virtual void remove(int index) =0;
    ...
};

class StringName : public virtual Name {
protected:
    string fName;
    ...
}

class VectorName : public virtual Name {
protected:
    vector<string> fComponents;
    ...
}
```

Listing 4: The Name interface class and the StringName and VectorName implementation classes.

The Name interface decouples clients from Name implementations like StringName and VectorName. Not only can you change your existing implementations without affecting clients, you can also introduce new and better implementations without breaking client code.

The concept of interface classes is described as Pattern 3.

Name:	Interface Class.
Problem:	You need to design and implement a concept with different implementations.
Context:	You want to give clients freedom of choice (for selecting a specific implementation). You want to give clients freedom from choice (by not having to care about implementations). You want to change implementations without affecting clients. You want to introduce new implementations without making clients notice. You want to separate implementations from their clients. You want to make it as simple as possible, but not simpler.
Solution:	Determine the functionality of the concept separately from its implementations. Represent the functionality as an interface class (a class with only pure virtual functions). Make implementation classes inherit and implement the interface class.

Pattern 3: Thumbnail of Interface Class pattern.

With pure virtual base classes, you can make your implementation classes implement several different interfaces at once. I usually avoid inheriting implementations from more than one class, but I think inheriting from several interface classes is just fine.

5 Abstract Base Class

Clients of the Name interface class can now handle Name objects using the interface without bothering with implementations. But you still have to provide implementations. StringName had a lot of code that looked similar to VectorName code. In fact, if StringName and VectorName use similar primitive member functions, chances are the member functions based on the primitive functions are similar or even identical.

For example, the implementation of “void StringName::insert(int, string)” and “void VectorName::insert(int, string)” might look like displayed in Listing 5.

```
// from StringName.C
void StringName::insert(int index, string component) {
    assertIsValidIndex(index, noComponents()+1);
    basicInsert(index, component);
}

// from VectorName.C
void VectorName::insert(int index, string component) {
    assertIsValidIndex(index, noComponents()+1);
    basicInsert(index, component);
}
```

Listing 5: Identical implementations of the “insert” member function in StringName and VectorName.

It’s time to reuse code through inheritance.

So far, we only have an interface class that defines how to access Name objects. It does not implement any code common to both StringName and VectorName. In order to capture that common code, you introduce an abstract class AbstractName that you make the base class of both StringName and VectorName. You make AbstractName implement the Name interface class and remove the direct inheritance relationship between Name and StringName as well as Name and VectorName. The resulting design is display in Figure 1 using UML notation.

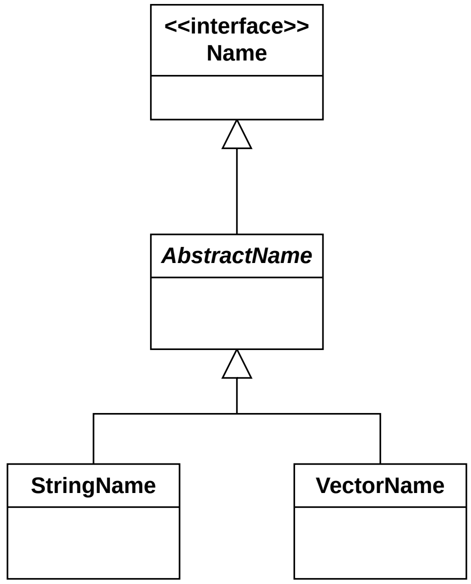

Figure 1: Design of Name class hierarchy.

AbstractName is an *Abstract Base Class.* An abstract base class is a class that can not be instantiated and that is only partially implemented. Also, AbstractName does not provide any member variables. This is left to subclasses like StringName and VectorName. This way, you can introduce new subclasses of AbstractName without burdening them with unwanted member variables. It is generally a good idea to push member variables into the subclasses.

We still distinguish between the Name interface class and the AbstractName base class, because after having introduced two different subclasses of AbstractName, we reckon there might be other Name implementations in the future. These new implementations might not fit under AbstractName but might have to implement Name directly.

The class definitions, shown in Listing 6, reflect these considerations.

```
public Name {
    ... // only pure virtual member functions
}

public AbstractName : virtual public Name {
    ... // mixture of pure virtual and regular member functions but no state
}

public StringName : public virtual AbstractName {
protected:
    string fName;
    ...
}

public VectorName : public virtual AbstractName {
protected:
    vector<string> fComponents;
    ...
}
```

Listing 6: Definition of the Name, AbstractName, StringName, and VectorName classes.

After analyzing the commonalties of StringName and VectorName you decide which member functions to move to AbstractName. This is possible for most non-primitive member functions like “component”, “insert”, “remove”, etc. Listing 7 shows three example member functions.

```
string AbstractName::component(int index) {
    assertIsValidIndex(index);
    return basicComponent(index);
}
```

```
void AbstractName::insert(int index, string component) {
    assertIsValidIndex(index, noComponents()+1);
    basicInsert(index, component);
}

void AbstractName::remove(int index) {
    assertIsValidIndex(index);
    basicRemove(index);
}
```

Listing 7: Some member functions of the AbstractName abstract base class.

This all works very well for those member functions that really are the same between the two implementation classes StringName and VectorName. But when you try to implement the primitive member functions do you recognize that these differ significantly between the two classes. No way that you could abstract them into a shared function of the common abstract base class! But still, you have to declare them on the level of the abstract base class. The code in Listing 6 does not compile if AbstractName does not at least declare the primitive functions “basicComponent”, “basicInsert”, and “basicRemove”.

So you declare the member functions that AbstractName relies on but can not implement as pure virtual functions. StringName and VectorName must now implement these pure virtual functions. StringName and VectorName are *concrete classes*, of which you can create instances.

The set of pure virtual member functions that a class inherits from AbstractName is called the *inheritance interface* of AbstractName. It is display in Listing 8.

```
public Name {
    ...

public:
    virtual int noComponents() =0;
    virtual bool isEqual(Name* name) =0;
    virtual string asDataString() =0;
    GenericName* StringName::contextName() =0;
    ...
}

public AbstractName : public virtual Name {

protected:
    virtual string basicComponent(int index) =0;
    virtual void basicComponent(int index, string component) =0;
    virtual void basicInsert(int index, string component) =0;
    virtual void basicRemove(int index) =0;
    virtual Name* newName(string name) =0;
    ...
}
```

Listing 8: The inheritance interface of AbstractName.

The inheritance interface of a class is the set of (typically pure virtual) member functions that a subclass has to implement to be a concrete readily instantiable class. The inheritance interface may become rather large, as Listing 8 shows. However, implementing this interface is a rather small price given that you inherit a large amount of shared code that your subclasses don’t have to implement anymore.

Pattern 4 describes the Abstract Base Class pattern.

Name:	Abstract Base Class.
Problem:	You need to ensure identical behavior of concept implementations where functionality is identical, and provide different behavior, where functionality is different.
Context:	You want to avoid redundant code. You want to ease adding other implementations. You want to make it as simple as possible, but not simpler.
Solution:	Separate variant functionality of the implementations from invariant functionality. Implement the invariant functionality as shared functionality in an abstract base class. Declare the variant functionality in the abstract base class using pure virtual functions. Make implementations subclasses of the abstract base class. Make the implementation subclasses implement the variant functionality.

Pattern 4: Thumbnail of Abstract Base Class pattern.

By now, the difference between an interface class and an abstract base class should have become clearer. An interface class defines an interface but imposes no baggage on its implementations. Implementations may vary significantly, as long as they implement the interface. An abstract base class, in contrast, is a partial implementation that defines some shared behavior of its subclasses (typically, but not always, while implementing an interface). So Interface Class and Abstract Base Class go hand in hand but are different patterns. Most pattern descriptions of abstract base class that I have seen confuse these two patterns. By separating interfaces from partially reusable implementations, you gain freedom in design and implementation.

The remaining question is how to best determine the inheritance interface. Our example shows a wealth of pure virtual member functions, doing all kinds of things. We address this problem next.

6 Narrow Inheritance Interface

After you moved all this code into AbstractName, you start consolidating StringName and VectorName. They look so much simpler now, because you only have to implement about ten member functions rather than the 25-30 you had to implement without an abstract base class.

However, our little story has sharpened your understanding of class evolution. You wonder what happens if you want to introduce a third subclass, perhaps based on a string array, of if another person wants to introduce his or her own subclasses. Suddenly, an inheritance interface of ten member functions doesn't look so good any more.

To minimize work for introducing a new subclass, the inheritance interface should be as small as possible. If only a few member functions need to be implemented, it becomes considerably easier to introduce new subclasses.

The AbstractName inheritance interface reveals that it can reduced further. There are two ways to do so:

- you can provide simple implementations of non-primitive member functions.
- you can implement some of the primitive functions using other primitive member functions.

Two examples of the first case are the member functions "asDataString" and "contextName". Both can be implemented generically by iterating over the name, picking each component and glueing them together for their respective purpose. These generic implementations are slow, but they work.

An example of the second case is the "void basicComponent(int i, string c)" member function that sets a name a name component c on index i. It can be implemented by first calling "basicInsert" on a given index i and then "basicRemove" on the index i+1.

Using these two techniques, you reduce the inheritance interface of AbstractName to its bare minimum. It is shown in Listing 9. Such an inheritance interface is called a *Narrow Inheritance Interface.*

```
public AbstractName : public virtual Name {
    ...

protected:
    // inheritance interface
    virtual string basicComponent(int index) =0;
    virtual void basicInsert(int index, string component) =0;
    virtual void basicRemove(int index) =0;
    virtual Name* newName(string name) =0;

    // other protected member functions
    ...
}
```

Listing 9: The narrow inheritance interface of AbstractName.

The purpose of a narrow inheritance interface is to minimize the effort needed to change the underlying implementation of a subclass or to introduce a new subclass.

If the implementations that our little reduction technique produced are too limited or too slow, you can always replace them through faster implementations. For example, subclasses of AbstractName are likely to override the implementation of "basicComponent", because "basicComponent" is a core primitive member function that should be as fast as possible.

The definition of a narrow inheritance interface should be rooted in how subclasses use its abstract base class. Making it easy to implement new subclasses is an important goal, but if you want subclasses to be fast, you may as well decide not to provide default implementations of core primitive member functions like "basicComponent".

So, we have arrived at our last pattern, the Narrow Inheritance Interface pattern. It is displayed as Pattern 5.

Name:	Narrow Inheritance Interface.
Problem:	You need to minimize efforts to introduce new subclasses of an abstract base class.
Context:	You are using an abstract base class with many pure virtual member functions. You expect existing subclasses to evolve and new subclasses to enter the system. You want to make it as simple as possible, but not simpler.
Solution:	Reduce the number of pure virtual member functions to its minimum by • using design by primitives; • providing default implementations of primitives where possible; • implementing all non-primitive member functions using primitives..

Pattern 5: Thumbnail of Narrow Inheritance Interface pattern.

Typically, the Narrow Inheritance Interface pattern ties in well with the Design by Primitives pattern. Our primitive member functions are prime candidates for a narrow inheritance interface, because they are the only member functions that deal with member variables, which are typically introduced only by subclasses.

7 Conclusions

What have we gained, next to writing up five fundamental class design patterns?

First, an observation about vocabulary. All pattern names are based on common vocabulary and common usage. None of the terms was unknown or invented by me. As developers, we have a huge amount of terms with specific meanings that are waiting to be analyzed for their patternhood.

Second, an observation about pattern relationships. The Simple Class, Interface Class, and Abstract Base Class patterns are on a different level than the Design by Primitives and Narrow Inheritance Interface patterns. The first three patterns are alternatives for designing and implementing a domain concept; you pick whichever pattern suits your specific problem best. The succession from Simple Class to Interface Class to Abstract Base Class is not a necessary one and only a consequence of my choice to present the pattern as a story.

This observation also gives us a hint about whether Simple Class is a pattern. In my opinion, it is a true pattern, because it represents one of several alternatives. Simple Class only seems trivial, if we don't take the alternatives (Interface Class and Abstract Base Class) into account. Now that we know these alternatives, every decision to use Simple Class becomes a decision for simplicity but against flexibility and ease of evolution. Such a decision is only justified if the context allows for it. Hence, Simple Class is a true abstraction from a solution to a problem in a specific type of context.

Third, an observation about composability. Ken Auer's patterns for "Reusability through self-encapsulation" [1] present similar patterns, illustrated using Smalltalk. His patterns form a linear succession suggesting that you apply one pattern after another. In another instance, Bobby Woolf presents the Abstract Class pattern [2], which is a combination of several aspects of all five patterns of this article. I view Bobby's pattern as a compound pattern that tries to bring everything together in one description.

In this article, I have deliberately kept the patterns separate without implying a specific order of application (even though the chosen sequence of pattern instantiations seems to be...um...a common pattern itself). I have done this so that you can both apply the patterns stand-alone and flexibly compose them. The patterns present basics of our design and implementation vocabulary and much like composing words when speaking, we compose them when designing.

Finally, an observation about context. These patterns may seem to stand alone, but of course they do not. There are higher-level patterns, like those from the Design Patterns book [3], and lower-level patterns of member function types and properties. As a consequence, the full context of the patterns remains elusive and we can not nail down all the forces of when and when not use the patterns. For example, I haven't talked much about the importance of teamwork and how it determines when to prefer Interface Class over Simple Class.

So, whether to apply one pattern or another always depends on your experience and informed judgement. It all depends on context, and in our open world, context is open-ended and infinite.

8 Acknowledgments

I would like to thank Brad Appleton, Steve Berczuk, Patrizia Marsura, and John Vlissides for their feedback on this article.

References

[1] Ken Auer. "Reusability through Self-Encapsulation." Pattern Languages of Program Design 1. Addison-Wesley, 1995. Page 505-516.

[2] Bobby Woolf. "Abstract Class." Pattern Languages of Programming 1997. WUSTL Technical Report 97-34. Washington University at St. Louis, 1997.

[3] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

[4] Taligent Inc. Taligent's Guide to Designing Programs; Well-Mannered Object-Oriented Design in C++. Addison-Wesley, 1995.

[5] Larry Wall, Tom Christiansen, Randal Schwartz, and Stephen Potter. Programming Perl (2nd Edition). O'Reilly, 1997.