Fundamental Class Patterns in Java

Dirk Riehle, dirk@riehle.org, www.riehle.org

Classes are fundamental to object-oriented design and programming. In this paper, 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, Abstract Superclass, Narrow Inheritance Interface and 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 Overview

Figure 1 shows a roadmap to the patterns discussed in this paper. The solid arrows indicate how the patterns are applied one after another in the evolution of our example. The dashed arrows show which patterns are needed to realize the pattern from which the arrows come.

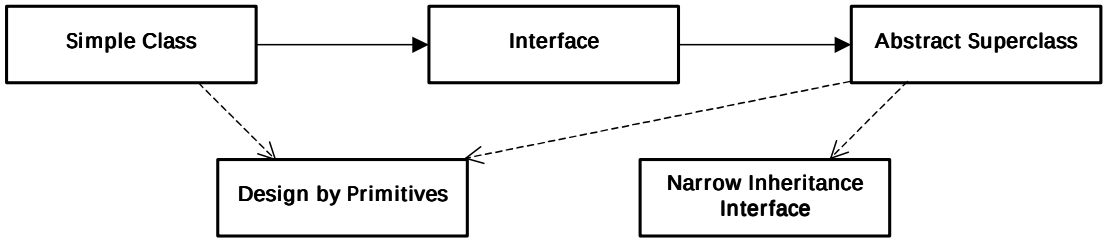

Figure 1: Roadmap to "Fundamental Class Patterns in Java."

We start out with the *Simple Class* pattern. It tells you to implement a domain concept as a single, typically simple, class, unless you have specific more complex requirements.

When you implement a Simple Class, you are likely to use *Design by Primitives*, which shows you how to build the class from primitive methods: you first devise a set of basic, "primitive", methods, and then implement more complex methods by composing the primitive methods.

Once you want to have variations of the same class, you will want use the *Interface* pattern to design a Java interface for your classes. Using an interface, clients will not know about specific implementation classes. This lets you change classes implementing the interface without affecting clients.

Eventually, if your implementation classes are getting more mature, you may want to abstract what is common into an *Abstract Superclass*. The abstract superclass pattern tells you how to organize algorithms and reusable methods in such a way that you can have different implementations of the same concept and still reuse code.

Abstract superclasses may be hard to understand and it may be difficult at first to implement a new subclass. You can use the *Narrow Inheritance Interface* principle to make it as simple as possible for users of your abstract superclass to implement a new subclass.

Name:	Simple Class.
Problem:	You need to design and implement a concept.
Solution:	Implement the concept as a single (simple) class.

Thumbnail 1: Simple Class pattern.

Name:	Design by Primitives.
Problem:	You need to implement a class.
Solution:	Separate primitive from complex methods and implement the complex methods using primitive methods.

Thumbnail 2: Design by Primitives pattern.

Name:	Interface.
Problem:	You need to design and implement a concept with different implementations.
Solution:	Use a Java interface to abstract from different implementations.

Thumbnail 3: Interface pattern.

Name:	Abstract Superclass.
Problem:	You want code reuse between different implementations of an interface.
Solution:	Use an abstract superclass from which concrete subclasses inherit algorithmic structure and reusable methods.

Thumbnail 4: Abstract superclass pattern.

Name:	Narrow Inheritance Interface.
Problem:	You want to make reusing an abstract superclass as simple as possible.
Solution:	Minimize the number of abstract inherited methods by new subclasses.

Thumbnail 5: Narrow Inheritance Interface pattern.

2 Name 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 “~/java/jvalue/Value.java” are “~”, “java”, “jvalue”, and “Value.java”. 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. 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 “~/java/jvalue/Value.java”, the file system first resolves “~” to your (Unix) home directory, then searches for a directory called “java”, followed by searching for a directory called “jvalue”, 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.

3 Simple Class

Let’s try to use Name objects to represent file names and other multi-part names. A simple thing that does the job is a single class with a field that holds the sequence of strings. A convenient representation of the sequence of strings is a List object of strings.

Listing 1 provides a snapshot of this class. The field components holds the list of strings.

```
public class Name {

    public List components;

    // get name component
    public String getComponent(int index) {
        return (String) components.get(index);
    }

    // set name component
    public void setComponent(int index, String component) {
        components.set(index, component);
    }

    // insert name component
    public void insert(int index, String component) {
        components.add(index, component);
    }

    // prepend name component
    public void prepend(String component) {
        insert(0, component);
    }

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

    // 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, called 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: Summary 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 and Abstract Superclass. 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.

4 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 list, etc. Naturally, as a lazy developer (which is to say as a good developer [4]), you start moving these repeated code fragments into methods of their own.

Listing 2 shows some methods that come into being this way.

```
public class Name {

    // primitive method: returns number of components in name
    public int getNoComponents() {
        return components.size();
    }

    // primitive method: asserts that given index is valid
    protected void assertIsValidIndex(int i) {
        assertIsValidIndex(i, getNoComponents());
    }

    // primitive method: asserts that given index is valid
    protected void assertIsValidIndex(int i, int upperLimit) {
        if ((i < 0) || (i >= upperLimit)) {
            throw new IllegalArgumentException("Invalid index!");
        }
    }

    // primitive method: returns name component at given index
    protected String doGetComponent(int i) {
        return (String) components.get(i);
    }

    // primitive method: inserts name component at given index
    protected void doInsert(int index, String component) {
        components.add(index, component);
    }
```

```
    ...
}
```

Listing 2: A set of primitive methods.

All methods of Listing 2 are so-called primitive methods because they do exactly one well-defined thing. Please notice that “doComponent” and “doInsert” 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 methods are protected so that they can be used only from inside the object.

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

```
public String getComponent(int index) {
    assertIsValidIndex(index);
    return doGetComponent(index);
}

public void insert(int index, String component) {
    assertIsValidIndex(index, getNoComponents() + 1);
    doInsert(index, component);
}

...
```

Listing 3: Working with primitive methods.

The “component” and “insert” methods compose the primitive methods to do their task. More complex methods like “getContextName” (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 methods as well.

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

Designing a class using primitive methods 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 methods. You want to avoid a fragile class in which changes to a method affect too many other methods. You want to make it as simple as possible, but not simpler.
Solution:	Separate more complex non-primitive methods from primitive methods. Determine the primitive methods that best help implement the class. Implement non-primitive methods using primitive methods.

Pattern 2: Summary of Design by Primitive pattern.

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

5 Interface

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. This way we get rid of the List object and reduce memory consumption of Name objects. For example, a string representing the Name "~/java/jvalue/Value.java" looks like "~#java#jvalue#Value.java" 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 List-based Name class, but it is also slower.

So you actually need both classes. You rename the existing Name class to ListName 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 ListName objects interchangeably. A client of your Name class does not want to write its code twice just to deal with different classes that effectively do the same thing.

Thus, you decide to use an *interface*.

In our example, you introduce an interface called Name. It declares all the functionality that is common to both StringName and ListName. You then make StringName and ListName implement Name. Using the Name interface, clients can now work with StringName and ListName objects without committing to any one of these two subclasses.

Listing 4 shows the Name interface and how StringName and ListName relate to it.

```
public interface Name {
    public String asString();
    public String asDataString();
    public String getComponent(int i);
    public Iterator getComponents();
    public char getDelimiterChar();
    public char getEscapeChar();
    public bool isEmpty();
    public bool isEqual(Name name);
    public int getNoComponents();
    public void append(String component);
    public void setComponent(int index, String component);
    public Name getContextName();
    public String getFirstComponent();
    public void insert(int index, String component);
    public String getLastComponent();
    public void prepend(String component);
    public void remove(int index);
    ...
};

public class StringName implements Name {
    protected String name;
    ...
}

public class ListName implements Name {
    protected List components;
    ...
}
```

Listing 4: The Name interface and its StringName and ListName implementations.

The Name interface decouples clients from Name implementations like StringName and ListName. 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 is captured as Pattern 3.

Name:	Interface.
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. Make implementation classes implement the interface.

Pattern 3: Summary of Interface pattern.

6 Abstract Superclass

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

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

```
// from StringName
void insert(int index, String component) {
    assertIsValidIndex(index, getNoComponents() + 1);
    doInsert(index, component);
}

// from ListName
void insert(int index, String component) {
    assertIsValidIndex(index, getNoComponents() + 1);
    doInsert(index, component);
}
```

Listing 5: Identical implementations of the “insert” methods in StringName and ListName.

There is some obvious opportunity for code reuse here.

So far, we only have an interface that defines how to access Name objects. It does not implement any code common to both StringName and ListName. In order to capture that common code, you introduce an abstract class AbstractName that you make the superclass of both StringName and ListName. You make AbstractName implement

the Name interface and remove the direct inheritance relationship between Name and StringName as well as Name and ListName. The resulting design is displayed in Figure 1 using UML notation.

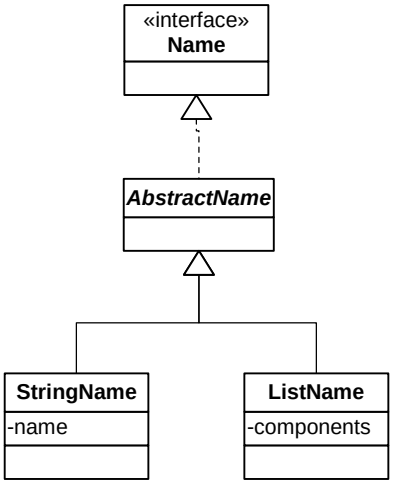

Figure 1: Design of Name class hierarchy.

AbstractName is an *Abstract Superclass.* An abstract superclass is a class that cannot (or should not) be instantiated and that is only partially implemented. Also, AbstractName does not provide any fields. This is left to subclasses like StringName and ListName. This way, you can introduce new subclasses of AbstractName without burdening them with unwanted fields. It is generally a good idea to push fields into the subclasses.

We still distinguish between the Name interface and the AbstractName superclass, 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 interface Name {
    ... // only abstract methods
}

public abstract class AbstractName implements Name {
    ... // mixture of abstract and regular methods but no state
}

public class StringName extends AbstractName {
    protected String name;
    ...
}

public class ListName extends AbstractName {
    protected List components;
    ...
}
```

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

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

```
String getComponent(int index) {
    assertIsValidIndex(index);
    return doGetComponent(index);
}

void insert(int index, String component) {
```

```
    assertIsValidIndex(index, getNoComponents() + 1);
    doInsert(index, component);
}

void remove(int index) {
    assertIsValidIndex(index);
    doRemove(index);
}
```

Listing 7: Some methods of the AbstractName abstract superclass.

This all works very well for those methods that really are the same between the two implementation classes StringName and ListName. But when you try to implement the primitive methods do you recognize that these differ significantly between the two classes. No way that you could abstract them into a shared method of the common abstract superclass! But still, you have to declare them on the level of the abstract superclass. The code in Listing 6 does not compile if AbstractName does not at least declare the primitive methods “doGetComponent”, “doSetComponent”, “doInsert”, and “doRemove”.

So you declare the methods that AbstractName relies on but can not implement as abstract methods. StringName and ListName must now implement these abstract methods. StringName and ListName are *concrete classes*, of which you can create instances.

The set of abstract methods that a class inherits from AbstractName is called the *inheritance interface* of AbstractName. It is display in Listing 8.

```
public interface Name {
    ...
    public int getNoComponents();
    public boolean isEqual(Name name);
    public String asDataString();
    public Name getContextName();
    ...
}

public abstract class AbstractName implements Name {
    ...
    protected abstract String doGetComponent(int index);
    protected abstract void doSetComponent(int index, String component);
    protected abstract void doInsert(int index, String component);
    protected abstract void doRemove(int index);
    protected abstract Name createName(String name);
    ...
}
```

Listing 8: The inheritance interface of AbstractName.

The inheritance interface of a class is the set of (typically abstract) methods 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 Superclass pattern.

Name:	Abstract Superclass.
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 superclass. Declare the variant functionality in the abstract superclass using abstract methods. Make implementations subclasses of the abstract superclass. Make the implementation subclasses implement the variant functionality.

Pattern 4: Summary of Abstract Superclass pattern.

By now, the difference between an interface and an abstract superclass should have become clearer. An interface defines an interface but imposes no baggage on its implementations. Implementations may vary significantly, as long as they implement the interface. An abstract superclass, in contrast, is a partial implementation that defines some shared behavior of its subclasses (typically, but not always, while implementing an interface). So Interface and Abstract Superclass go hand in hand but are different 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 abstract methods, doing all kinds of things. We address this problem next.

7 Narrow Inheritance Interface

After you moved all this code into AbstractName, you start consolidating StringName and ListName. They should be much simpler now, because you only have to implement about ten methods rather than the 25-30 you had to implement without an abstract superclass.

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 methods 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 methods need to be implemented, it becomes considerably easier to introduce new subclasses.

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

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

Two examples of the first case are the methods "asDataString" and "getContextName". 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 doSetComponent(int i, string c)” method that sets a name a name component c on index i. It can be implemented by first calling “doInsert” on a given index i and then “doRemove” 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 class AbstractName implements Name {
    ...

    // inheritance interface
    protected abstract String doGetComponent(int index);
    protected abstract void doInsert(int index, String component);
    protected abstract void doRemove(int index);
    protected abstract Name createName(String name);

    // other protected methods
    ...
}
```

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 in a subclass through faster implementations. For example, subclasses of AbstractName are likely to override the implementation of “doGetComponent”, because “doGetComponent” is a core primitive method that should be as fast as possible.

The definition of a narrow inheritance interface should be rooted in how subclasses use its abstract superclass. 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 method like “doGetComponent”.

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 superclass.
Context:	You are using an abstract superclass with many abstract methods. 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 abstract methods to its minimum by • using design by primitives; • providing default implementations of primitives where possible; • implementing all non-primitive methods using primitives..

Pattern 5: Summary of Narrow Inheritance Interface pattern.

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

8 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, and Abstract Superclass 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 to Abstract Superclass 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 and Abstract Superclass) 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.

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 fundamentals 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 method types and properties [5, 6]. As a consequence, the full context of the patterns remains elusive and we cannot nail down all the forces of when and when not use the patterns. For example, we haven't talked much about the importance of teamwork and how it determines when to prefer Interface 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.

9 Acknowledgments

I would like to thank Brad Appleton, Steve Berczuk, Patrizia Marsura, and John Vlissides for their feedback on an earlier version of this article. I would like to thank Kevlin Henney for his comments during PLoP 2001 shepherding.

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

[5] Dirk Riehle. "Method Types in Java." Java Report 4, 2 (February 2000). Page 22pp.

[6] Dirk Riehle. "Method Properties in Java." Java Report 4, 5 (May 2000).

10 Notes

AbstractName vs. CommonName?

Fields protected or private?

Method prefixes: as or to, nothing or is?