# Java Development Standards

***v0.8 [mlf-971030]***

***Mark L. Fussell***

*1220 N. Fair Oaks Ave, #1314*
*Sunnyvale, CA 94089*
*408.734-9068*

*Mark.Fussell@ChiMu.com*
*www.chimu.com*

# *Table of Contents*

# Overview

This document describes the standards ChiMu Corporation uses for Java development. Documenting these standards is important for our own needs and our customers, but we provide these standards publicly to help other teams who are establishing or growing their own standards. These standards may also help developers to better understand good software design: they are especially designed to help a developer think about the issues (e.g. some of them are "radical"). In many cases you may object to our recommendations and decide on a different standard, but the acts of reasoning about the issues and documenting your solutions is even more important[1].

This document is organized with an initial introduction and then the summary of our standards. Following this is the main discussion, explanation, and details of the standards. The document ends with definitions and references.

[1] Although this document does not use the Pattern literary form, conceptually it has similar goals.

# Introduction

The two most important principles to consider for high-quality software development are:

1. To think from the client's point of view
2. To think from the maintainer's point of view

Understanding and considering these two customers' needs during development makes most of the difference between poorly designed and very nicely designed systems. Object-oriented techniques can help support both of these customers' needs, but the principles must always be on your mind. Codifying those needs into your standards will help.

Some of the guidelines may sound like more work than you as designer or coder would like to do, but over time they will pay off to your clients, your maintainers, and you yourself when you are filling those two roles. And in these two roles is where most of the time and energy for software development is spent, even for a one-person project. Anything we write today will be used and maintained for many days, months, and years ahead. Or so we hope.

## *Sources of Standards*

The design and development standards documented here are summaries and extractions from more general standards as well as our own experiences. The following sources document these general "software engineering" practices. The full references are at the end of the document.

For publicly available coding standards I recommend the following resources for Java, Smalltalk, and Eiffel. I choose these standards because Java is best thought of as simplified Smalltalk with interfaces and bare-bones static typing added to it. The syntax looks like 'C' or 'C++', but the semantics are much closer to a cross between a very diluted Eiffel and Smalltalk.

| | |
|---|---|
| Doug Lea's Java coding standards | [Lea] |
| Smalltalk Best Practice Patterns: Coding | [Beck 96] |
| Code Complete | [McConnell 93] |
| Object-Oriented Software Construction, 2nd Edition | [Meyer 97] |
| Eiffel, The Language | [Meyer 92] |
| Smalltalk with Style | [Skublics+KT 96] |

Software design and object oriented design have a much broader range of relevant topics and resources, but the following are some good introductions.

| | |
|---|---|
| Design Patterns | [Gamma+HJV 95] |
| Object-Oriented Analysis and Design with Applications | [Booch 94] |
| Object-Oriented Modeling and Design | [Rumbaugh+BPEL 91] |
| Information Modeling | [Kilov+R 94] |

Finally, in terms of relational databases, there are two primary authors (Codd and Date) and much of the recent discussions have been in Database Programming and Design. The following are some of the books that may be useful.

| | |
|---|---|
| The Relational Model for Database Management | [Codd 90] |
| An Introduction to Database Systems | [Date 95] |
| Relational Database Writing | [Date 95b] and the previous editions |
| The Object Database Standard: ODMG-93 | [Cattell+ 96] |
| Object-Relation DBMSs | [Stonebraker+M 96] |

There is a collection of terms and definitions at the end of this document, which form part of our standard development dictionary. Many of these terms are reconciled with:

| | |
|---|---|
| Directory of Object Technology | [Firesmith+E 95] |

# Standards Summary

## Ultimate Principles

| | |
|---|---|
| *Think from the Client's point of view* | Think about what a client needs and how a client will be using your functionality. Support their needs before yours. Protect their code from changes. Make sure they will know how to correctly use you system. |
| *Think from the Maintainer's point of view* | Think about what a maintainer needs to support the maintenance and enhancement process. Make sure they can understand your code's responsibilities and its implementation. Make life as simple as possible. |

## Interfaces and Classes

| | |
|---|---|
| *Interface with 'interface's* | Use interface as the glue throughout your code instead of classes. Interfaces focus on the client's needs: they define what functionality a client will receive without coupling them to the implementation. Variables, parameters, return values, and casts should all use interfaces. |
| *Avoid exposing classes* | Avoid exposing implementation classes at all. Provide creation methods on interfaces to other objects or on Packs. Only expose classes that are meant to be subclassed. |
| *Do not suffix interfaces* | Do not suffix or prefix interfaces. Interfaces describe functionality publicly and own the "normal" namespace.<br>***Example**: `Query, Table, Sequence`* |
| *Suffix classes with 'Class'* | To avoid collisions between interfaces and implementation classes, suffix your classes with 'Class'. Classes provide implementation and are both localized and secondary compared to interfaces. If it is an abstract class suffix it with 'AbsClass'.<br>An exception is a class called from the command line (i.e. main entrypoints) which does not need to be suffixed for user convenience.<br>***Example**: `QueryClass, TableAbsClass, JdkVectorWrapperClass`* |
| *Provide different interfaces for different clients* | Use different interfaces to document and control the functionality provided to different types of clients. For example, you can document the functionality provided to users of a subsystem through a public interface, which is separate from the interface used within the subsystem. |
| *Suffix Extended, Subsystem, and Package interfaces* | Use suffixes for extended (Xi), subsystem (Si), and package (Pi) interfaces and inherit in a hierarchy (Xi<-Si<-Pi). This allows you to identify which interfaces should be visible to whom, and also to know whether an internal cast will succeed.<br>***Example**: `Query, QueryXi, QuerySi, QueryPi`* |

*Provide creation methods*

Instead of having clients use 'new' provide creation methods to construct objects. Prefix these creation methods with 'new' and place them on interfaces of an appropriate object or on a 'Pack'.

***Example**: MetaPack.newMethodReference, Orm.newObjectMapperNamed_table*

## Naming

*Name things well*

Spend extra effort to choose good names for your Types and Methods. Choose intention revealing method names [Beck 96] and simple, appropriate class names. Always try out a name by using it in client code before committing to a name in real code.

*Standardize your naming*

Decide on the meaning of a word or pattern and then use it consistently. This is especially important for method naming patterns, which should be reused as much as possible.

***Example**: is…, setup…, new…, with*

*Use Woven Parameters*

Put underscores ('_') as placeholders for where parameters belong in a selector. Leave off any trailing underscores.

***Example**: atIndex_put, newDirectSlot_column_type*

## Organizing

*Use 'Pack' classes*

Every package should have a 'Pack' class that documents the package's functionality and provides a common place for creation methods and required static functionality.

***Example**: FunctorsPack /** Functors contains interfaces that support using object that encapsulate functions (aka Commands)… **/*

*Avoid 'static' methods*

Try to avoid making public members 'static'. Instead try to allocate the functionality to another appropriate object or use the Singleton pattern.

***Example**: VmPack.theVm().canSupportWeakReferences().*

*Consistently order your modifiers*

Use a consistent ordering of class and member modifiers. Put 'static' first to make conspicuous this different (non-OO) kind of member. Put access modifiers next so a client can tell if a method is visible. Follow access with the special modifiers. Finish with return type.

***Example**: public int, static protected void, private synchronized final Object*

*Augment access modifiers with comments*

Use comments to describe more precisely what type of access you are providing. Be explicit about 'package' visibility, mention if you expect 'protected' to really be 'progeny' (i.e. no package visibility), and specify if a method is 'public' but is only so to support a 'system' or 'package' interface (interfaces require methods to be public).

***Example**: public, /*subsystem*/ public, /*package*/ public, /*package*/, protected, /*progeny*/ protected, private*

*Categorize your methods* — Divide your methods into categories and organize your class structure around them.

## Abbreviation and Acronyms

*Initial cap acronyms* — Initial cap acronyms instead of putting them in all capitals (i.e. treat them as words).

***Example**: `JdbcConnection, JglCollection, FormPack`*

*Keep capitalization when abbreviating* — Abbreviate long names by reducing their internal words to the initial letter. A sequences of capitals indicates multiple words abbreviated in a row.

***Example**: `CompLangJava -> CLJava.`*

## Growing and changing standards

*Try it out* — Live with a standard for a while before deciding to scrap or change it. Let the goals of a standard grow into your habits so you fully understand its value to you.

*Prove performance* — Do not sacrifice a standard for performance reasons until you see the profiling numbers. Only optimize when it will quantifiably be worth the maintenance penalty.

*Take out the trash* — If a standard does not work for your team, create a new one or let the issue be context and programmer dependent until a new standard emerges.

# Coding and Design Standards

This chapter describes our standards for both coding and coding related design. I believe these to be inseparable: All object-oriented software development project I have dealt with require the "coder" to also be at least a partial designer, and the better the design skills the better the software.

## *Classes, Interfaces, Types and Protocol standards*

Interfaces and Classes are the dominant structuring mechanism for object-oriented programming. They determine how your system looks to clients and they help organize your implementation. It is important to consider the client's point of view first.

### Interfaces for interfacing

Use interfaces as the glue throughout your code instead of classes. Interfaces focus on the client's needs: they define what functionality a client will receive without coupling them to the implementation. Variables, parameters, return values, and casts should all use interfaces.

Classes are implementations of interfaces and should provide no public behavior beyond the interface itself (other than how to create and initialize an object of that class). Avoid exposing classes except when you want to provide the ability for a client to subclass.

Interfaces should be given no suffixes or prefixes: they have the "normal" name space. Classes are given a suffix of "Class" if they are meant to be instantiated or are given a suffix of "AbsClass" if they are an abstract class that provides inheritable implementation but is not complete and instantiable by itself.

#### Examples

All variables, return values, and parameters are typed to an interface (a Type).

```
    Point aPoint = GeometryPack.newPointX_y(3,5);
```

Methods commit to what type they return, but not the particular implementation class

```
    Point newPointX_y(float x, float y) {
        if ((x == 0) && (y ==0)) return ZERO_POINT;
        if (y == 0) return newPointR_theta(x,0);  //Just to show the
encapsulation
        return new PointXYClass(x,y);
    }
```

```
    Point newPointR_theta(float r, float theta) {
        if (r == 0) return ZERO_POINT;
        return new PointRThetaClass(r,theta);
    }
```

Similarly, methods specify the parameter type they need but do not restrict the implementation of that type:

```
    Rectangle newRectangle(Point c1, Point c2) {
        if (c1.equals(c2)) return new DegenRectangleClass(c1);
        return new RectangleClass(c1.x(), c1.y(), c2.x(), c2.y());
            //We know c1 can tell us x(), y(), but not if it
            //actually stores 'x' and 'y' as instance variables
    }
```

The only place where the Class itself is exposed is within the implementation of the factory method (as above) and the methods of the actual object.

Classes and abstract classes can implement the "unsuffixed" interface directly:

```
    public class PointAbsClass implements Point {...
```

or indirectly when a parent class implements it for them:

```
public class PointXYClass extends PointAbsClass {...
```

### Discussion

There are many benefits to using interfaces as the glue throughout your systems, the following are just two of the most important benefits. First, clients will not be coupled to the specific implementation, so you can have much more flexibility in evolving the implementation plus you can provide alternative implementations to support proxies, tracing, and performance variations. Second, you can use multiple inheritance among interfaces and between interfaces and classes, which can help with OO modeling and can support different access views of the same class (see below on Access Control).

The penalties of using interfaces everywhere are having to managing interfaces separately from classes and any performance penalty a particular JVM has for calling through interfaces. Managing separate files is overhead but it provides an important reward: you are focused on the client when modifying the 'interface' file. It is much harder to keep a 'client' focus when in a 'class' file with all the implementation details around.

The performance penalty depends on the particular VM. On some current VMs it varies between 30% overhead and 150% overhead over a straight message call[2]. Rarely is this "message overhead" a significant burden for an application, but in a few tight circumstances it may be and 'class-typing' can replacing the 'interface-typing'[3]. Because classes are conspicuous these cases will be easy to identify and reverse if other needs take precedence.

Overall, the advantages of using interfaces far outweigh the disadvantages, especially for the client and maintainer. The extra flexibility, object modeling improvement, and client consideration provided by interfaces improve programs significantly.

## Interface and Class naming

Make sure the name of an interface or a class matches the range of usage of the interface or class. If it is designed to be very general, choose a very simple name. If it is designed to be used in a more specific context, qualify it enough to describe the range of functionality that it is meant to have.

You should spend a fair amount of time making sure your interface and class names are very good. Try names out and fix them if they don't work well in actual use or if they do not fit well with other interfaces and classes in the system. This is especially valuable in the younger stages of a project – before other people have mentally and programmatically committed to a name.

## Types and protocols

We separate specifying an object's public interfaces in two ways: specifying what an object is (its Types) and specifying what an object responds to (its Protocols). These are both specified using Java interfaces, but protocols tend to be "mixed in" to the Types as opposed to having their own hierarchy. It is useful for modeling to make the distinction, and we to suffix the mixin protocol with "able" (Printable, Sortable).

[2] Although the percentage is significant, the actual overhead is very small in time: less than 1 microsecond on older VMs on a 100MHz machines and less than 1/10th a microsecond on newer VMs.

[3] The actual overhead should be verified through a performance profiler. Never assume something is a bottleneck until you actually see the numbers.

## Access Control

Java has 4 levels of access control[4]:

| | |
|---|---|
| Public | Visible to everyone |
| Protected | Visible only to the class and its subclasses (when each is acting as a subclass) |
| Private | Visible only to the class itself (no subclasses) |
| Package | Visible only to the classes within the same package |

We primarily use Public and Protected access control. Since we use interfaces to specify the public interface of a class, the only methods that are public and not in some interface are creation methods (constructors and initialization). We use package visibility if we want a method to be visible to other classes in the package to support implementation that is not part of the public behavior. We rarely use Private because we rarely find it necessary to consider a subclass as not having the ability to access its superclasses' implementation.

Our default order of access is:

| | |
|---|---|
| Protected | All instance variables and methods that are not part of the public interface to the class. |
| Public | Methods that are part of the public Type interface or are publicly needed to create the object |
| Package | Any non-public methods that other classes in the package need for implementation |
| Private | Special use. |

Although the above access control standards work fine on their own, they do not work well with interfaces which requires methods to be public. The following section overrides the current section because of this limitation in Java.

## Package Interfaces and Subsystem Interfaces

In a continued attempt to document and encapsulate types and classes, We now use interfaces for package access as well as public access. Unfortunately, Java does not allow you to specify that an interface contains methods that are only package accessible. When you create an interface, its methods are automatically public and that means the methods must be public in the class implementing those methods. So if we use an interface to specify the package interface, all those methods must be public on the class.

Fortunately, this is not a problem when considered with the other standards. Since no client of the class is expected to ever see the class itself, but must instead interact with the interface that is available to the client. So if a class only has access to a public interface, it can only use the methods specified in that interface and can not normally see or use methods (although also "public" on the class). A class that has the package interface (which is private to the package) will be able to use both the public and the "package public" interfaces.

The standard is still that the public interfaces have no suffixes. The naming convention for the additional Package and Subsystem interfaces is as follows:

| | |
|---|---|
| Pi | Package Interface: Interface *for within* Package . This specifies the interface/methods available to other classes in the same package. This is usually the most inclusive interface (other than for the class itself). |
| Si | Subsystem Interface: Interface *for within* Subsystem. This specifies the interface/methods available to other classes in the same subsystem but in a different package. This is more inclusive than the public interface but more restrictive than the package interface. |
| Xi | Extended Interface: An extension of an interface to provide more functionality than is |

[4] See [Gosling+JS 96] for more details on these access controls, especially the formal definition of the protected access level. Java has a more complicated concept of access than Smalltalk would because access is considered from the class perspective instead of from the object perspective. In Java a method being executed within one object can access another object's private and protected areas if the other object is of the same class. In Smalltalk this would not be allowed: an object would be able to access another object's methods only if they have public or package visibility. Java has trades the advantage of easy cloning for breaking encapsulation at the object level.

commonly needed. Only certain "sophisticated" clients will want to use an Xi interface.

In this case, these interfaces form a "perfect" hierarchy (Foo - FooXi - FooSi - FooPi) so moving both up and down the hierarchy is guaranteed to succeed for an instance of 'FooClass'.

## Constructors and Factory methods

For Java we always use Factories and Factory methods for "public" object construction. We use factory creation methods instead of straight constructors because they

- Allow more flexibility in creating a new object: maybe we just reuse an existing object
- Can have better names: "newTimeNow()" and "newTimeFromSeconds(...)" instead of "new Time()" and "new Time(...)"
- Provide better separation between interface and implementation: we can document the factory method in an interface
- Naturally flow into more sophisticated factory designs (See [Gamma+HJV 95])

Generally we try to use another appropriate object as the Factory for a particular class (a database object create Tables). For classes that have no other appropriate factory object we use the 'Pack' object as the factory.

## Pack Objects

It is very useful to be able to treat a Package as an object, so in every package we have a class named "<packageName>Pack" (e.g. java.util would have a "UtilPack" class). This provides clients with a single interface for creating objects, finding singleton objects, and any other "static" (non-object) behavior. It also prevents the problem of exposing a class from behind the interfaces: class constructors and static methods can be private to the package with the pack object exposing what ever functionality is needed. Clients of a package see only the public interfaces and the Pack's functionality and have no visibility to the implementation classes.

You can interact with a Pack in two ways: as either a singleton object or as a "static-side" virtual object. The singleton object is more flexible because you can pass the object around within a program (for example, to recurse a package hierarchy). The static object is more convenient for construction (e.g. "UtilPack.newDate()") but is less flexible and more coupled. Overall, convenience tends to win out because construction is the main use of a Pack (but remember that it is better to give Factory responsibilities to another suitable object than the Pack).

## Abbreviations note

Using initial capitalization for words implies abbreviations should be initial capitalized as well. This makes it easier to identify the word boundaries but is against normal abbreviation policies: in this approach abbreviations are inherently all lowercase and are only capitalized when the start of a word boundary. Effectively abbreviations are treated like normal 'words' (i.e. as if they had come into common usage for English). This also means that if a word is abbreviated to a single letter, there can be a series of capital letters that indicate a series of one letter abbreviated words. DomainStorageInformation can be abbreviated to DSInformation and would have three words ('D', 'S', 'Information'). The previous example of PointXYClass is four words ('Point', 'X', 'Y', 'Class). Numbers are considered to start their own word consisting of numbers followed by any non-capitalized letters. For example, 'Procedure2Arg' is three words ('Procedure', '2', 'Arg'), '49ers' is one word, and 'root4The49ers' is 4 words ('root', '4', 'the', '49ers').

# *Method Standards*

Although not as visibly dominant as Classes and Interfaces, methods form a potentially equally valuable organization of a programs behavior. By having well named methods that are used consistently a client can

correctly use and predict the functionality in interfaces and objects. This makes a potentially complex program much simpler and understandable.

## Standardized, woven, intention-revealing method names

We use intention revealing method names where parameter positions are interwoven into the name by the inclusion of underscores "_". Any underscores at the end of the name (before the open parenthesis) are omitted from the name. Generally a method's name will uniquely determine the number and types of parameters it requires, but if there are a large number of parameters, the signature may not be unique without including the number of parameters. It is a rare and very controlled condition for when we will have two methods with signature's that are unique except for the type of a parameter.

We have many standardized words (especially prefixes) used in naming methods which reveal what the method does. We *do not* use "get" as a prefix for asking about an attribute (see below for the discussion of attributes).

### Examples

| | |
|---|---|
| dictionary.atKey_put(key,value); | Woven parameters |
| person.name(); | GetAttribute |
| person.setName(newName); | SetAttribute (if necessary and atomic) |
| array.atIndex(index); | "at" is not overloaded to take both a key (Object) and an index (int) |
| person.isHappy(); | Returns Boolean |
| factory.paintCar_using(aCar,white,quickly) | Extra unwoven parameters |

## Intention revealing names

Having "intention revealing method names" is the primary rule for naming methods. Create a name that suggests what the method "provides for the caller", not how the method accomplishes this service. The clearer you can make your methods behavior by the name of the method, the easier it is for the client (who repeatedly uses your method) to understand your class or interface. This is one of the many incarnations of thinking from the client's perspective. See the coding pattern "Intention revealing selector" in [Beck 96].

## Standardized naming patterns

The next rule after picking intention revealing method names is to standardize the vocabulary used in the name. As much as possible, words should be used consistently and uniquely when part of a method. The following is an example subset of the standard meanings for method name parts and method categories (see the Source code format section below).

The following are common method prefixes

| **Prefixes** | **Category** | **Description** |
|---|---|---|
| is, can, has, will | Testing | Return a Boolean and test the state of the object |
| copy, copy<Property> | | Produce a copy of the current object. Suffix describes properties that the new object will have. |
| init, setup | Initializing | These methods are called before you can use an object. Only a single init function should be called which can then be followed by whatever setup methods you need to change the default configuration of the object. |
| doneSetup | Initializing | Some objects require calling a "doneSetup" to allow them to prepare themselves for normal, post initialization, interactions. |
| new | Creating | Create and return a new object from a factory that creates only a single type of object |
| new<Type> | Creating | Create and return a new object of a specific type |

| | | |
|---|---|---|
| as<Type>, as<Property> | Converting | Produce a copy of the object (if changes were required) converted to a different type or property. ‘asString’, ‘asUppercase’ |
| to<Type>, to<Property> | Converting | If the current object is mutable, modify it to satisfy the type or property. Otherwise this is synonymous with as<Type>. |

A few type specific prefixes are:

| | | |
|---|---|---|
| find | Searching | Retrieve a single object or null if unsuccessful |
| select | Searching | Retrieve multiple objects or an empty collection |
| value | Evaluating | Evaluate the function and return an object |
| execute | | Execute the procedure, no return value |
| add | | Add an object to a collection |

Non-prefix method name patterns

| | |
|---|---|
| with | This prefix is used to add subsequent parameters to a basic method when the parameters have no distinguishing features (an example is the "value, valueWith, valueWith_with" series for Function objects) |
| any | Return any object that satisfies the request (findAny) |
| all | Return all objects that satisfy the request (selectAll) |

Within each Type or domain area (Collections, Functors, SQL, Mapping, Domain models) there will be both reused vocabulary and new vocabulary.

## Java method signatures

Java selects the method to execute based on the name of the method, the number of parameters to the method, and the declared (not actual) types of the parameters to the method. [§8.4.2] Together these form the signature of the method.

To produce good code, the developer needs to make it easy for the client of the code to do what was intended. The developer, the client, and the compiler must all agree on what will happen when the program executes; it should be easy for a client to remember what methods are needed and how to call them. Allowing differences in certain features of a signature to change what method is called makes life harder on the client.

The first problem is with using the declared type of a variable to modify what method is called. This makes it harder for the caller to realize they are calling the wrong method because the information used to decide which method to call is spread over two locations: the actual method call and the variable declaration. For example, given the methods:

```
void doItTo(Person person);
void doItTo(Company company);
void doItTo(Object anyObject);
```

the call to a method:

```
doItTo(person);
```

is not enough to determine which method is called. Even if the object in the variable person is really a person (not a company), it will depend on whether the variable 'person' was declared as type Person or as only a general object. In general we consider this to be unacceptable, so we only use this feature (overloading of signature based on a parameters declared type) in very controlled circumstances (See below).

The second problem is having the number of parameters determine different behavior of methods. Although it is visible which version of the method you are calling (simply count the parameters), it is unclear what the purpose of the parameters are and what order they should be placed in. Although we do allow methods that are only distinguishable by the number of parameters, we try to avoid it.

## Principles for signatures

Our standard for method signatures is to have them be mostly distinguishable by name. Ideally two methods with the same name will take the same number and type of arguments. In infrequent cases the number of parameters will determine what method to call, but this primarily happens when a method has a significant number of parameters (4 or more) so the "Parameter positions interwoven into the method name" pattern is cumbersome.

The only time the type of a parameter will determine which method to call is to support using literal values (Strings, Numbers, and Booleans) as direct parameters. These types will always be incompatible with the "standard" type of the parameter. For example, if the standard method requires an object of type Person, say "Company::firePerson(Person p)", it is acceptable to have a method with an identical name that takes a String instead of a Person in that parameter position "Company::firePerson(String personName)". This is disapproved of, and it would be better to name the method "Company::firePersonNamed(String personName)". Usually this is only a problem when a set of related messages all have a significant number of parameters (4 or more), so the names are not likely to be unique.

## Parameter positions woven into the method name

A general problem with parenthesized parameters is that there is no reminder of the order of parameters. This is especially true of methods with more than two arguments, but can also be a problem even with only two parameters. Examples of these problems are:

```
hash.put(value,key)
    // put value at key? no, wrong order
statement.setObject(index,type,value)
    // set the entry at index to the type using value?
```

Generally Smalltalk has less of a problem with this because the parameters are woven into the method call, which encourages a much higher level of readability. The example of a method definition in Smalltalk is:

```
Dictionary>>at: key put: value
```

or to put it into close to Java terms:

```
public void at: Object key put: Object value {...};
```

Because of this message weaving, we have come up with a clearer name "at:put:" instead of just "put", both of what the method does and especially of which parameter is expected in what position. Even though Java does not allow this type of syntax, it would be nice to encourage these improved message names and to have the messages document for the client the order to put the parameters.

### First Solution

The first solution we used was to put underscores "_" as place holders to identify where a particular parameter is woven into the message send. For example:

```
at_put_(key,value)
```

would read as at_(first parameter)put_(second parameter) or "at (key) put (value)". The second example would be:

```
setIndex_to_asType_(index,value,type)
```

or "set index (index) to (value) as type (type)". This does a good job of specifying the meaning of the message, the number of parameters, and the specific positions of all the parameters. If you have a large number of parameters that you do not want to specifically mention/weave into the method name, you can use a double underscore to indicate two or more values. For example:

```
at_putStuff__(key,value1,value2,value3)
```

Although this was a very consistent and logically understandable solution, all the added underscores were annoying, especially because the underscores at the end of the name visibly separated the method name from the parameter list. Those trailing underscores were also the hardest to remember to put in, probably partially because they were not really part of the "spoken name" itself: it is easy to pause within a name, but hard to express a pause at the end of a name.

### Better Solution

A better working solution is a hybrid approach. Put in the underscores just as before, but drop all trailing underscores. The examples become:

```
at_put(key,value)
setIndex_to_asType(index,value,type)
at_putStuff(key,value1,value,value3)
```

So there are underscores to indicate the position of all parameters except the last one (and any unwoven ones). This means "put(value,key)" is also valid, because it has an intermediary form "put__(value,key)" and then the underscores are dropped. Although "put" is valid using the standard, it is still an inferior method name to "at_put" which reveals intention and its use better.

Because the trailing underscores have now been dropped, we can not tell how many parameters a method takes. We know the minimum: 0 if no underscores and 1+#underscores if there are any. Unfortunately we can not tell the difference between 0 and 1, or between 3 and 7. Luckily other standards or the obvious intention of caller make these not much of a problem: calling a method that requires no parameters (asking about an attribute) is so significantly different from calling a method that requires a parameter (setting an attribute) that they are not likely to be confused.

## Attributes

An attribute is a public property of an object that shows the state of the object. Frequently there is a minimal collection of attributes that uniquely determine the state of the object, but this should not be confused with the instance variables which may be used to store that state.

### No ‘get’ prefix for attributes

We do not use the prefix "get" to ask the value of an object’s attributes. A method sounding like a noun indicates an attribute, which distinguishes it from a method that changes the state of an object (which will use an active verb). This has several positive and a couple negative consequences.

On the positive side, leaving off the ‘get’ avoids a noisy and uninformative word[5]. Not using ‘get’ makes code read better in terms of objects and is consistent with the standards in most other OO languages. On the negative side, leaving off ‘get’ causes two problems. It is different from the standard Java idiom and it is different from the ‘pattern’ used by JavaBeans to automatically guess a class’s BeanInfo description. The second of these is solvable by using explicit BeanInfos, which provides a number of benefits in addition to naming flexibility.

Overall, considered in isolation, it is the better choice to avoid ‘get’. Considered in the context of Java’s idioms and libraries it might be better to use ‘get’, but it will depend on your particular context. For us the conceptual correctness and improved naming is worth the overhead. We do use ‘get’ for classes designed to be treated like basic data structures.

### Attribute setting

Not all attributes are changeable, in which case they will not have a setter. Not all attributes can be changed independently (state transitions should be as valid and complete as possible), in which case they will not have a setter, but will instead be changed as the result of another method. Finally, if an attribute is independently changeable it will have a setter with a signature of the attribute name prefixed with “set”, returning “void”, and taking a single argument.

## Avoid ‘static’

Rarely use ‘static’ members as part of a class or subsystems public interface. The keyword ‘static’ is the most powerful destructive force in Java. With one word you can turn Java from an object-oriented,

[5] Plus ‘get’ itself is not even the right word, which should instead be something like ‘give’, ‘your’, or ‘produce’.

interface-based language into a very poor compile-time binding and implementation binding. You can not use interfaces with 'static' functionality and you can not subclass to provide a different implementation.

The better option is to place the functionality on another appropriate object, which allows you to have all the interface and implementation benefits of OO programming. If there is currently no Type appropriate for the functionality, use the Singleton pattern [see Gamma+HJV 95]. With this pattern you would create a real interface and class that you expect only a single instance of. Then you can provide (through another object or the 'Pack') a way for clients to retrieve that object and interact with it normally from then on. The client is only bound to the interface, so you can swap out the implementation (or have multiple instances) without the client being harmed.

### Instance variables

An instance variable is an implementation detail of a class and should be "private" or "protected". Whether we wrap an instance variable with private get/set methods is dependent on the needs of the class. We never use the ability to access another object's (of the same class) instance variables except in "clone" type methods.

## *Source code format*

Almost none of this standards document is focused on the actual formatting of code. This is because the most important rules "be consistent" and "be logical" seem to suffice in most cases and the set of existing formatting standards suffice for the rest. The following sections discuss some higher level or less well covered considerations for Java code formatting.

### Sectioning

The following describes the general sectioning of interface and class files. There are also method categories within these major sections, and these are visible in source files but not in the JavaDoc API. Because classes are encapsulated within interfaces and "packs", the sectioning of a class file is only important for the class developer or maintainer and should be organized to help that person.

#### Section ordering for Interfaces

- Constants
- Methods

#### Section Ordering for Classes

- Object
  - Public
    - Constructors
    - Constants
    - Methods
  - Private
    - Constructors
    - Methods
    - Instance variables
- Static/Class
  - Public
    - Constants
    - Methods
  - Private
    - Constants
    - Methods
    - Instance variables

#### Subsectioning

Within each section are method categories that organize the methods of that section. These are equivalent to Eiffel features labels (without the export control) or Smalltalk protocols/method-categories (without having a nice browser). A category normally ends with 'ing'. Categories can have subcategories that organize on a more refined level. Categorizing is designed to make a class or interface file more organized

and readable, so you should try to stay in the happy middle between too few categories (e.g. none) and too many sections (e.g. one per method)

The following is an ordered list of categories and definitions:

| | |
|---|---|
| **Constructing** | A section and category. The constructors for the class. |
| **Initializing** | An additional method that should be applied directly after constructing the object. |
| **Setup** | Methods that can optionally be applied to an object but must be done immediately after construction and initialization and before using the object normally. |
| **Validating** | Check whether the current object is in an acceptable state (could also be under asking if this is possible after construction is finishing). |
| **Asking** | Asking the state of the current object without causing any (visible) side effects. A pure function. ISE Eiffel 'Query'. |
| **Testing** | An asking method that returns a Boolean value |
| **Querying** | More complicated asking questions. |
| **Printing** | Return a printable representation of the object |
| **Displaying** | Render an object (as text or graphics) on a display medium |
| **Creating** | Create new objects |
| **Copying** | Produce a copy of the current object with possibly different properties (e.g., a subset of a collection) |
| **Converting** | Transform the current object to another type of object |
| **Enumerating** | Perform an element-by-element operation on a collection |
| **Altering** | General changing of the state of an object. Altering methods can also return values so they are not guaranteed to be pure procedures although they frequently are. |
| **Adding** | Add elements to a collection |
| **Removing** | Remove elements from a collection |
| **Notifying** | Notify a dependent of a change |
| **Releasing** | Remove any dependents and references to other objects (this helps with garbage collection) |
| **Utilities** | Usually part of the private section. Just internal private methods that help get a job done. Not-elsewhere-classified. |

The following is an example of sectioning in code. This would look a bit more balanced if the method bodies were included.

```
//************************************************************
//(P)******************** Copying *************************
//************************************************************

public Object copy() {..}
protected void initFrom(CompanyClass company) {..}

//************************************************************
//(P)****************** Displaying ************************
//************************************************************

public String toString() {...}
public String info() {..}

//************************************************************
//(P)******************** Asking **************************
//************************************************************

public String name() {..}
public int revenue() {..}

//************************************************************
//(P)****************** Creating ************************
//************************************************************
```

```
    public Project newProject(String name, Date startDate) {..}

    //*********************************************************
    //(P)******************* Altering ************************
    //*********************************************************

    public void setName(String name) {..}
    public void setRevenue(int revenue) ..}
```

## Comments

We place the JavaDoc comments immediately above and inset relative to the method declaration. This is so it is easy to read the method declaration ***before*** reading the comment. A method should have a good, intention revealing selector, good parameter names, and a suitable return value type. This implies that the declaration itself is the best first source for documentation of the public use of the method.

```
       /**
        * Prepare the query so that its unbound variables can
        * be bound
        */
    public void prepareForBinding() {
```

In effect, we consider the comment to be inside and subservient to the declaration, but JavaDoc requires it to be before the declaration.

Because we try to have communicative method names and standard patterns for categories of method, we will frequently leave off method comments. We never repeat the information that can be found simply by reading the declaration.

We do not indent Class comments. Class comments have enough information that they are not optional on released software.

## Interfaces and comments

Interfaces are where the public protocols of a system are defined, so they provide the main documentation. Documentation is not repeated in a class for the methods that are defined in an interface. Only implementation documentation will be in a class.

## Keyword Order

Since we consider static methods to be a completely different kind of method (they are actually statically bound functions and procedures), this is the first qualifier mentioned. After this comes the access control (including the comment specifying more specific access than Java currently provides). This is followed by all the not-elsewhere-mentioned qualifiers. Finally we have the type specification.

1. [‘static’]
2. ‘public’ | ‘/*subsystem*/ public’ | ‘/*package*/ public’ | ‘/*package*/’ | ‘protected’ | ‘/*progeny*/ protected’ | ‘private’
3. [‘abstract’], [‘synchronized’], [‘final’], [‘native’], [‘transient’], [‘volatile’]
4. ‘void’ | <TypeName>

# Design Notation

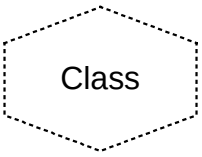
Design notation enhances communication among people. It is an important tool for both the design process itself and for communicating the finished design to clients and maintainers. Design notation needs to be standardized because it is shared among many people who must all interpret it consistently. On the other hand, the notation needs flexibility to grow and support the expression of newer ideas.

We use an object design notation that is similar to UML (see [Rational-1 http://www.rational.com/uml/]), but it has some variations that are improvements, additions, or legacies from earlier notations. We also consider the work by Kilov and Ross on information modeling [Kilov+R 94] to be very useful and important. Although [Kilov+R 94] does not require a specific visual notation, they do suggest one, which I merged into my notation.

The following describe my notation's differences from UML. If something is not mentioned it should be assumed that the UML notation is the correct notation.

## *Objects and Classes*

The most obvious difference between UML and my notation is I use hexagons instead of rectangles to identify objects and classes. This is to maintain the property from Booch notation of objects being distinctly recognizable and visible in diagrams. UML/OMT rectangles are neither distinct (other notations use rectangles to indicate tables, rows, layers, etc.) nor particularly visible in complex diagrams. These properties are very important for me.



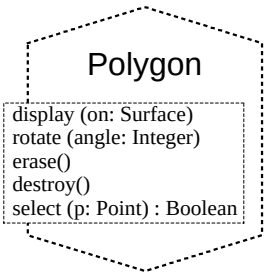

Rumbaugh did have a legitimate argument that Booch clouds were difficult to draw and even to electronically use (because they are hard to connect to). Booch and Rumbaugh came up with a very good (synergistic) solution of using hexagons in early versions of the UML, but later versions of the UML abandoned them. This was unfortunate and I decided not to make the same mistake.

### Class Names

The examples do not have the Java standard suffix of "Class" for classes. This is because it is unneeded: classes and types have distinct symbols, so do not share the same name space. When converting design class names to Java names, all classes will have the "Class" or "AbsClass" suffix added to them, and variables will be typed to Java interfaces (for the Type) instead of classes.

### Methods

The "compartment" for methods extends beyond the edge of the hexagon because methods are potentially publicly visible. Methods can either all be in one compartment, or separated into different compartments for the different interfaces to the class.

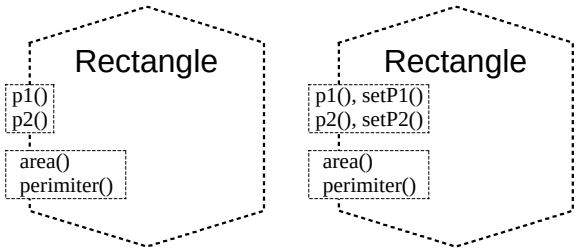



If placed within one compartment, different protocols (groups of methods) are organized like lists are in UML.

I use UML method annotations.

## Attributes

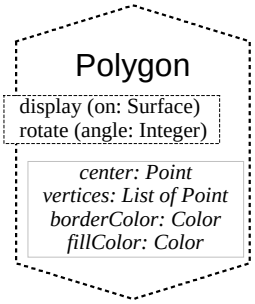
An attribute is a public property of an object that shows the state of the object. I do not make any distinction between attributes and other methods other than possibly showing them as a separate protocol from other methods. If attributes are independently changable, they will have a corresponding 'set' method.




Frequently there is a minimal collection of attributes that uniquely determine the state of the object, but this should not be confused with the instance variables which may be used to store that state.

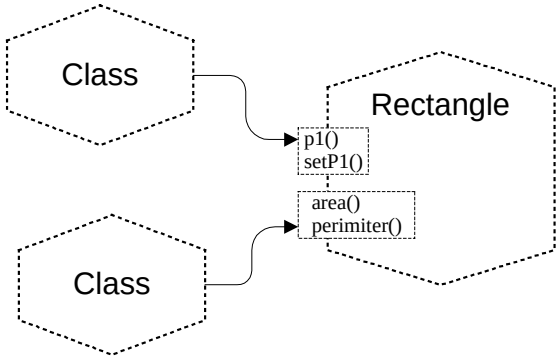
## Instance Variables

Instance variables are private to a class and a shown in a compartment completely within the hexagon and (usually) below the methods. This is inverted from UML where frequently instance variables are considered attributes and are shown before the methods. Stylistically they are centered and italicized.




## Message Sends

Inter-class interfaces and message sends can be shown by connecting arrows to the appropriate method of a class.




## Shorthand for hand drawings

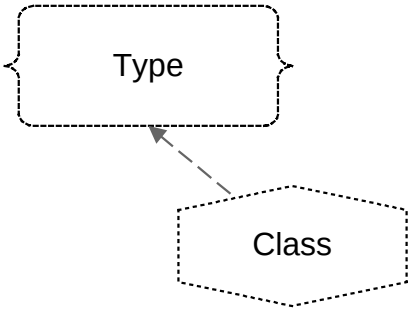
It is acceptable to fall back on just using a rectangle with three compartments (but having instance variables at the bottom) when drawing classes by hand.

# *Types*

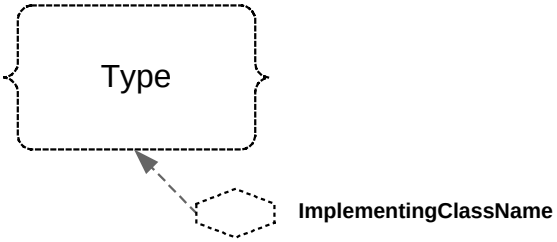
I use a separate notation for Types than for classes. Types are shown with a "Set-like" notation of a curly edged box. This distinguishes them from classes and makes them more distinct then just a rectangle again. Otherwise they have the same meaning as the rectangles in [Kilov+R 94].

Method (and Attribute) compartments can be specified for a Type, but instance variables can not be.


A Class (or Object) can be shown to implement a Type by an arrow pointing from the Class to the Type.




When the class is not important enough to be very large, its symbol can be shrunk and the name placed outside. This is frequently useful for giving example implementations of a Type in a complicated diagram.



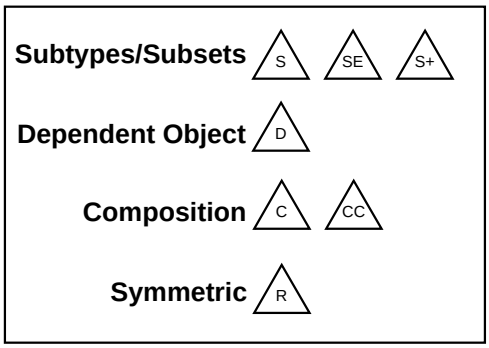

### Shorthand for hand drawings

It is acceptable to draw types as straight rectangles instead of the curly rectangles.

## *Relationships*

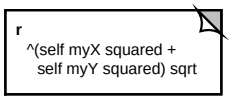
Generally I use UML notation when possible. *Information Modeling* [Kilov+R 94] has a richer set of relationships than UML and a cleaner decomposition of them, so in certain diagrams I will use them instead of the less precise UML notations.

### Kilov and Ross relationship types




## *Code Blocks*

I use the same dog-eared notation as *Design Patterns* [Gamma+HJV 95] for methods and other code annotations

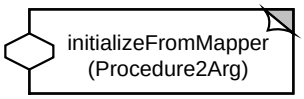


# *Other Notations*

This section shows some other more unusual notations that are also interesting.

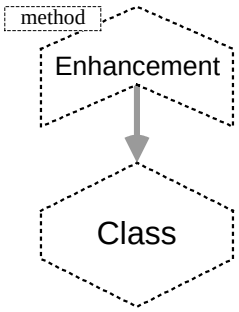
## Functor

Functors are objects that model operations that can be performed. Their notation is interesting because of its distinctive combination of other notations already existing: hexagons for classes and dog-eared notes for Code blocks.



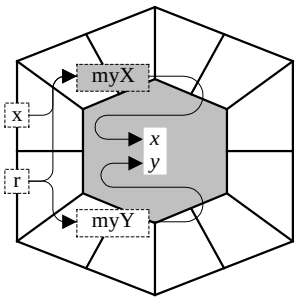

## Enhancement

An enhancement to a class is an added piece of functionality beyond the core class functionality. Enhancements allow you to manage portions of functionality of a cClass individually while still having all that functionality directly available on instance of the class. For example, you might want your presentation logic to be separate from your true (presentation independent) business domain logic. You could make the ability to create user-readable information an enhancement of your domain class. Since Java doesn't support enhancements this notation is only useful for conceptual modeling.




## Private Functionality

A class can be considered to have a public interface that talks to an inner class that manages the state of the object. This means all communication from the public interface will go through private methods to access state information, which allows the object to change its state representation (e.g. delegating to another object) without impacting the public methods.



# Standard Definitions

This section collects our standard project-independent definition of terms. Though these do include terms that are relevant to the areas of specialty we normally deal with (Information Systems), it is only at the highest level. The following collects terms and definition from many sources, but I compared and reconciled many of the terms with:

Directory of Object Technology [Firesmith+E 95]

In choosing and defining terms, I try to follow the following priority (best to worst). The best term:

1. Has a precise meaning in a known context (MetaClass for Smalltalk or CLOS)
2. Has common meaning in most OO discussions (Object)
3. Is infrequently used and I will define a precise meaning (Stub,Replicate,Forwarder)
4. Has overloaded or disputed meanings (Attribute)
5. Has a different, somewhat common, meaning than my definition (Slot)

| | |
|---|---|
| Object | An identifiable, encapsulated entity that is interacted with by sending messages. Objects have behavior, state, and identity (but see ValueObject for a variation). |
| Type | Specifies the public behavior and a conceptual grouping for objects that are members of the Type |
| Protocol | Specifies a collection of methods that together provide a higher level interface to an object. An object can **be** a Type, an object can **support** a Protocol, and a Type can **specify support for** a Protocol. Protocol and interface can be synonymous. Certain contexts suggest using one or the other to prevent conflict with a language term (e.g. Java's "interface"). |
| Class | Describes the types and the implementation for a set of objects. A class conforms to (or implements) a Type, Protocol, or interface. |
| Factory | An object that can create other objects. |
| Identity | The ability to tell an object apart from another object independent of whether their type and state is equal. |
| Immutable | Can not be changed after being created. |
| ValueObject | An object that does not have identity independent of its value. A ValueObject is immutable and should be considered identical to anything that it is equal to. Primitive data types in Smalltalk (such as most numbers and Symbols) are ValueObjects. Java Strings are very close to ValueObjects except they are not guaranteed to be identical for the same value (they would be if they did an automatic "intern()"). Java primitive types are not Objects. |
| AbstractDataType | Synonymous with ValueObject in the Object realm. |
| Attribute | A public property of an object that shows the state of the object. Frequently there is a minimal collection of attributes that uniquely determine the state of the object. |
| BasicAttribute | An attribute that takes its value from ValueObjects. This is as opposed to associations which connect two or more objects with identity. |
| Association | A defined relationship between two objects with identity. |

| | |
|---|---|
| Instance Variable | A private implementation to remember part of an object's state |
| ObjectShadow | The information needed to see that an object exists without any true representation of the real object. Relational databases could be considered to work with ObjectShadows: they record the information about an object but never have a real object to interact with. |
| Proxy | An object that stands in for another object (the RealObject) and manages the client interaction with the RealObject. |
| Forwarder | A proxy which immediately forwards messages, possibly over process and machine boundaries, to the RealSubject. |
| Replicate | A proxy which holds local state and performs local operations which are later propagated to the RealSubject |
| Stub | A proxy which is simply a placeholder for the RealObject and must become another type of proxy (for example, forwarder or replicate) when interacted with by a client. |
| RealIdentity | The identity of the RealObject that a proxy represents instead of the proxy's independent identity. For proxies we are rarely interested in their own identity, we just want to know the identity of the RealObject on the server. |
| IdentityKey | A value that defines the RealIdentity of a Proxy. |
| Binding | Associating a client object to a database object, which turns the client object into a Proxy |
| Builder | Builds up another object that can later be extracted |
| Writer | Writes information directly to another object (usually another writer or a Stream) |
| Reader | Reads information from another object (another Reader or a Stream) |
| Stream | Able to sequentially retrieve or store information |

.

# References


**Beck 96** Kent Beck. *Smalltalk Best Practice Patterns, Volume 1: Coding.* (also see writings at http://c2.com/ppr/titles.html)

**Booch 94** Grady Booch. *Object-Oriented Analysis and Design with Applications*. Benjamin/Cummings, Redwood City, CA, 1994.

**Brooks 75** Fred Brooks. *The Mythical Man-Month*. Addison-Wesley, Reading MA, 1975.

**Brown+W** Kyle Brown and Bruce G. Whitenack. "Crossing Chasms: A Pattern Language for Object-RDBMS Integration". http://www.ksccary.com/ORDBJrnl.htm

**Burbeck** Steve Burbeck, Ph.D. "Applications Programming in Smalltalk-80(TM): How to use Model-View-Controller (MVC)". http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html

**Cattell+ 96** R.G.G. Cattell, Editor. *The Object Database Standard: ODMG-93, Release 1.2.* Morgan Kaufmann, San Francisco, 1996.

**Codd 90** E.F. Codd. *The Relational Model for Database Management, Version 2.* Addison-Wesley, Reading, MA, 1990

**Coplien 92** James O. Coplien. *Advanced C++ Programming Styles and Idioms.* Addison-Wesley, Reading, MA, 1992.

**Coplien+S 95** James Coplien and Douglas Schmidt, Editors. *Pattern Languages of Program Design*. Addison-Wesley, Reading, MA, 1995.

**Date 95** C.J. Date. *An Introduction to Database Systems*. Addison-Wesley, Reading, MA, 1995.

**Date 95b** C.J. Date. *Relational Database Writings 1991- 1994.* Addison-Wesley, Reading, MA, 1995.

**Firesmith+E 95** Donald Firesmith, Edward Eykholt. *Dictionary of Object Technology: The Definitive Desk Reference.* SIGS Books, Inc., New York, NY, 1995.

**Gamma+HJV 95** Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Object-Oriented Architecture.* Addison-Wesley, Reading, MA, 1995.

**Goldberg+R 83** Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, Reading, MA, 1983.

**Gosling+JS 96** James Gosling, Bill Joy, Guy Steele. *The Java™ Language Specification*. Addison-Wesley, Reading, MA, 1996.

**Howard 95** Tim Howard. *The Smalltalk Developer's Guide to VisualWorks*. SIGS Books, New York, NY, 1995.

**Kilov+R 94** Haim Kilov and James Ross. *Information Modeling: An Object-Oriented Approach*. Prentice-Hall, Englewood Cliffs, NJ, 1994.

**Lea** Doug Lea. "Java Coding standards". http://gee.cs.oswego.edu/dl/html/javaCodingStd.html

**McConnell 93** Steve McConnell. *Code Complete: A Practical Handbook of Software Construction*. Microsoft Press, Redmond, WA, 1993.

**Meyer 92** Bertrand Meyer. *Eiffel, The Language.* Prentice-Hall, Englewood Cliffs, NJ, 1992.

**Meyer 97** Bertrand Meyer. *Object Oriented Software Construction, 2nd Edition.* Prentice-Hall, Englewood Cliffs, NJ, 1997.

**Rumbaugh+BPEL 91** James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William

Lorenson. *Object-Oriented Modeling and Design*. Prentice-Hall, Englewood Cliffs, NJ, 1991.

**Skublics+KT 96** Suzanne Skublics, Edward J. Klimas, David A. Thomas. *Smalltalk with Style*. Prentice Hall, Upper Saddle River, NJ, 1996

**Stonebraker+M 96** Michael Stonebraker with Dorothy Moore. *Object-Relational DBMSs, The Next Great Wave*. Morgan Kauffman, San Francisco, CA, 1996.

**Vlissides+CK 96** John Vlissides, James Coplien, and Norman Kerth, Editors. *Pattern Languages of Program Design 2.* Addison-Wesley, Reading, MA, 1996.