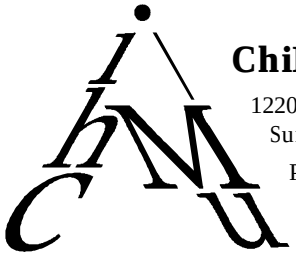
ChiMu OO and Java Development: Guidelines and Resources

v1.8 [mlf-980228]



ChiMu Corporation

1220 N. Fair Oaks Ave, #1314
Sunnyvale, CA 94089

Phone: 408 734-9068
Email: info@chimu.com

www.chimu.com

ChiMu OO and Java Development: Guidelines and Resources

Table of Contents

1 Overview

This document provides guidelines and resources for developers and teams building Object-Oriented (OO) and Java based software systems. The materials include:

An Introduction to OO terminology and concepts
Guidelines for OO/Java systems
Definitions for common OO terms
References to other good OO resources

This document is meant to both provide an integration of the many resources available for OO and also to serve as a jumping point into these other resources.

1.1 Prerequisites

Some familiarity with OO and Java is required for this material to be understandable. There are many introductory books on Java that focus on different types of reader background (and new books are added every month). For OO itself you may want to consider one of the following books:

Designing Object-Oriented Software	**Wirfs-Brock+WW 90**
Object-Oriented Analysis and Design with Applications.	**Booch 94**
Object-Oriented Modeling and Design.	**Rumbaugh+BPEL 91**
Object-Oriented Software Engineering: A Use Case Driven Approach.	**Jacobson+CJO 92**

For Java-oriented OO, consider:

Java Design: Building Better Apps & Applets	**Coad+M 96**
Understanding UML: The Developer's Guide	**Harmon+W 98**

There are also many materials available on the WWW. See the JavaSoft web site or Yahoo links under Java.

1.2 Related Resources

This document does not define standards but provides resources and guidelines that can be chosen as part of a team's standards. A common approach would be to define a separate standards document — along with useful supporting materials like code templates, UML examples, and CASE templates — that priorities guidelines, makes them more concrete, and fills in gaps that are important to a development team. ChiMu uses this approach internally and recommends it to clients.

1.3 Attributions

This document collects information from several different sources, which are indicated by footnotes and citations. The Guidelines chapter aggregates several authors' works together. To make the attribution easier to maintain after changes, the sources for a guideline is indicated underneath the guideline, with the first entry being the primary copyright owner for the particular wording. Subsequent entries are conceptual originators for the guideline. Unless otherwise noted, the original source is this document or another of ChiMu's documents.

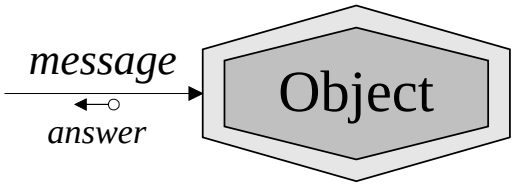
Special notice and thanks go to Doug Lea for allowing the inclusion of his guidelines within this document. See [Lea-1] for the original version.

2 Objects and Object-Orientation

This chapter introduces core concepts to OO and defines terms to provide a common foundation for reading the rest of the resources in this document. OO concepts and terms have been unifying over the last few years, but there are still differences and imprecisions in meanings. This chapter introduces the definitions and priorities for OO concepts that are used in this document and on ChiMu projects. See the resources mentioned in the Overview for a full introduction to OO concepts.

2.1 Objects

The core concept to OO is Objects. **Objects** are encapsulated entities that can be interacted with only by sending them messages. A **Message** is a stimulus sent to an object with a name and any parameters (as Objects) that the message requires. A message will cause the receiving Object to return an answer or possibly return nothing.



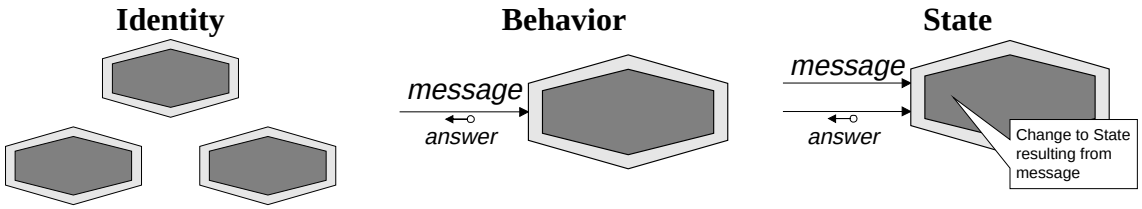
Because Objects are encapsulated they have a visible exterior (that to which you can send messages and from which answers are returned) and a hidden interior (the implementation of the functionality). Object's can also be viewed as having two core properties: Identity and Behavior.

Identity is the ability to tell one object from another object independently of whether they currently appear to be the same (i.e. have the same behavior).

Behavior is the response of an object to a stimulus. The only stimulus you can send to an object is a Message, so the behavior of an object is the answers it gives to the current and future messages.

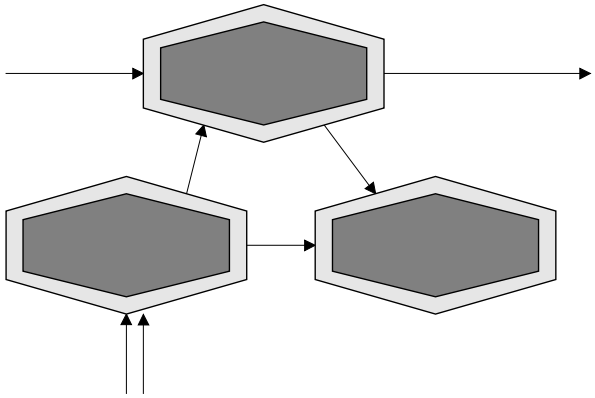
The above are the two core properties to an Object, but it is convenient to add a third (derived) property:

State is an abstraction to describe and simplify understanding an object's behavior. An object's behavior can be described as the answer it gives to the current message (which is determined by the current State) and the change to its State caused by the current message.



Summary

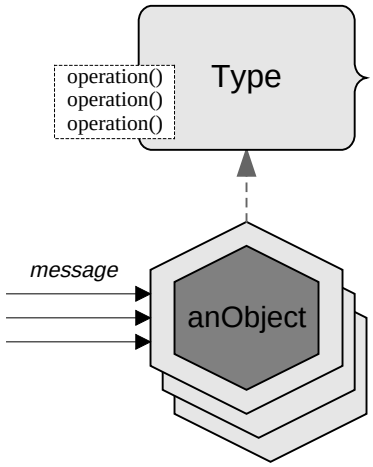
OO is building software out of Objects that send Messages to each other.



All the other concepts in OO are to make understanding, managing, or implementing these Object systems easier. Two of the most important concepts are Types and Classes, but there are many other important concepts, techniques, views, and methodologies that can help in the construction of Object software systems. Types and Classes will be discussed next and some of the other concepts are detailed in the Definitions chapter.

2.2 Types

Although OO systems can be built out of objects sending messages, any significant sized system would require so many objects that it would be impossible to understand them all individually. To handle this problem we can consider commonality of objects. The most important commonality to identify is a common exterior. This is important because Objects can have many clients (senders of messages) and a simplification of exteriors will benefit them all as well as the implementers of the objects.

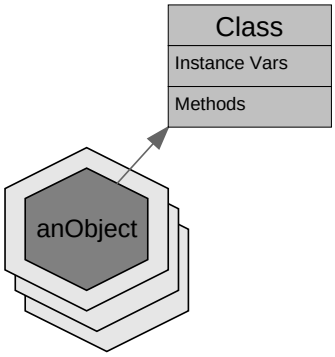


A **Type** describes a common exterior of a set of Objects. A Type makes understanding objects easier because their are much fewer Types than Objects (objects are mostly similar to each other), and objects can be grouped and understood by their similar behavior. A Type defines Operations for its set of Objects. An **Operation** specifies that the object has an ability to respond to a particular message and specifies the contract/requirement for that message (for both the sender and the receiver). An **Interface** is a description of a Type that only focuses on Operations.

2.3 Classes

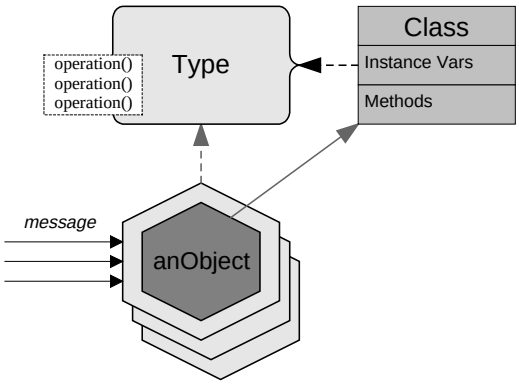
Where Types describe common exteriors of objects, Classes provide a common implementation for similar objects. Classes make it is easier to develop and manage the interiors of objects.

A Class has **Methods** that provide implementations for operations. These methods are shared among all objects (called **Instances**) that are implemented by that Class. A Class also has **Instance Variables**, which provide a way to store encapsulated state information for a particular object. Instance variables are completely hidden within the Object, but they enable two objects of the same Class to have different external Behavior.



2.4 Types vs. Classes

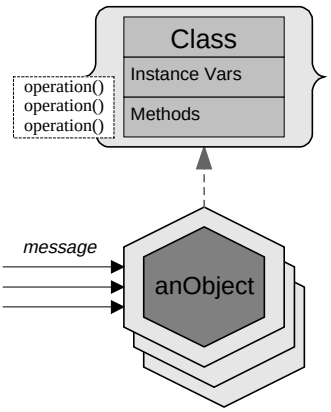
It is important to conceptually separate Classes from Types even though it is a common (bad) practice to use the term Class for both ideas. Types describe common exteriors of objects, classes provide a common implementation of objects. Neither of these are required for OO systems (some OO languages have one, the other, or neither), but both can be very useful for both understanding and managing a system's design and implementation.



2.5 Java

Java supports both Types and Classes through Java interfaces and Java classes. Java interfaces are a close equivalent to a Type: They specify the operations an Object understands, but do not restrict how those operations are implemented. Java interfaces support multiple inheritance and a Java class can implement multiple interfaces at one time.

Java classes are a Type (an exterior) combined with a Class (an implementation): A Java class can be used to specify the operations an object understands, but it also provides an implementation of the operations (or restricts the implementations to inherit from the specifying Java class). Java classes only support single inheritance and Java interfaces can not "extend" a Java class.



Java does not have a very sophisticated OO type model (no covariant or contravariant overrides) and Java also incorporates many non-OO features like primitive data types and static (compile-time bound) "methods". These limitations and non-OO features can distract or interfere with building good OO systems, so developers have to make extra efforts.

2.6 Other Concepts

The above is meant to be some basic definitions for OO that are very important and make understanding the subsequent guidelines and other resources a bit easier. For more information about other important OO concepts and more detailed explanations of the above concepts, see the references in the Guidelines, Definitions, and the References chapters.

3 Guidelines

This chapter contains guidelines that progress from very general OO guidelines to more Java specific development guidelines. Many of these guidelines are extracted from or based on other sources and the source to the guideline is noted immediately below the guideline or in footnotes.

3.1 Introduction

Development guidelines provide two high-level benefits: they reinforce techniques to design and code better and they provide consistency even when there is no clear best choice. This chapter provides guideline recommendations that focus on the first of these benefits: all of the guidelines in here are meant to help build better software. A separate (Standards) document can then chose which of these guidelines a team would like to use or the consistency can be more informally maintained by team interaction and code reviews.

These guidelines are simply recommendations. A particular guideline may be extremely beneficial for some teams and counter-productive for others. Try the guidelines out to see how well they work for your team's particular needs. Sometimes the guidelines conflict because they have advantages in different contexts. A standards document would be expected to define which guidelines a team chooses, or which are acceptable depending on the circumstance. Or the standards could be determined by the team's experiences and culture.

Sources of Guidelines

The guidelines documented here are summaries and extractions from more general guidelines, design principles, methodologies, and development experiences. The following sources document these general "software engineering" practices.

Software design and object oriented design are enormous subjects and there are numerous books and other resources covering them. See the References chapter of this document for a good collection of sources and the Definitions chapter for some discussions of what they focus on. The following list contains a few good introductions to OO related design:

Object-Oriented Analysis and Design with Applications	Booch 94
Designing Object-Oriented Software	Wirfs-Brock+WW 90
Object-Oriented Modeling and Design	Rumbaugh+BPEL 91
Design Patterns	Gamma+HJV 95
Information Modeling	Kilov+R 94

For Java development guidelines, we recommend the following resources from the Java, Smalltalk, and Eiffel literature. We chose these other languages because Java is best thought of as simplified Smalltalk with interfaces and bare-bones compile-time 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. The literature of good design techniques for Eiffel and Smalltalk is also more mature than Java literature.

Java Design: Building Better Apps & Applets	Coad+M 96
Doug Lea's Java coding standards	Lea-1
Smalltalk Best Practice Patterns: Coding	Beck 96
Code Complete	McConnell 93
Object-Oriented Software Construction, 2nd Edition	Meyer 97
Eiffel, The Language	Meyer 92

A major part of all standards and guidelines are good definitions for terms. See the chapter "Definitions" later in this document and the following for suggestions:

Directory of Object Technology	Firesmith+E 95
Analysis Patterns: Reusable Object Models.	Fowler 97

3.2 Overall Concepts

The following are three different perspectives on overall principles to good design and implementation for OO. None of these are a panacea: OO is a simple concept, but good design takes lots of learning and experience. These are meant to bring out some simple concepts that sometimes get lost.

Objects: The Core Concept of OO

The core concept of OO is that systems are built out of Objects with a clearly defined exterior and a completely opaque implementation. Objects are just like Cells, Computer Components, and People. Objects can only be interacted with by sending them messages, and a system performs its operations through the behavior associated with those messages. Sometimes just remembering this core metaphor can radically improve a design. Techniques like CRC sessions and anthropomorphizing can especially help.

Two Principles

Two important principles to consider for high-quality software development are to:

Think from the client's point of view

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. As the developer yourself, you will rarely forget your own needs.

Kent Beck

The following maxims are from Kent Beck's Smalltalk Best Practice Patterns [Beck 96]. These describe properties that should be in good OO designs and good OO code:

Once and only once	In a program written with good style, everything is said once and only once.
Lots of little pieces	Good code invariably has small methods and small objects.
Replaceable objects	Good style leads to easily replaceable objects.
Movable objects	…objects can be easily moved to new contexts.
Isolated rates of change	…don't put two rates of change together.

3.3 Guidelines Summary

The following table summarizes the guidelines for easier reference

3.4 General OO Guidelines

The following guidelines are statements of very general OO principles that provide overall guidance. These principles should help produce more specific guidelines and can be used as the restarting point if a particular guideline does not seem to work very well for a team.

Types

Types categorize and describe the exteriors of objects. In Java, a Type is represented as either an interface (a pure Type) or a class (a Type combined with an implementation). This document will use the term Type when considering the exterior properties of a class or an interface.

Name Types Well

> Spend the time to name a Type correctly and concisely. The name should match the range of usage for the Type: if it is very general, choose a very simple name; and if it is only applicable in a specific context, qualify it to describe that limitation. Sometimes naming a Type can be very easy because it exists as part of the business concepts (an Employee) or part of the solution (a Window). But always make sure the name really matches the concept. And really care about the name: "Be a poet for a moment."[*]

Rationale

Type names provide the main glossary and conceptual skeleton for any OO system.

Details and Examples

You should also try names out and fix them if they don't work well in actual use or if they do not fit well with names in the system. This is especially valuable in the earlier stages of a project – before the team has mentally and programmatically committed to a name.

ChiMu 97e, Beck 96

Only Expose Responsibilities

Only expose the features that you want to be responsible for. A Type provides a contract to all its customers that it will need to maintain "for life", so never publicly expose something you do not want to be responsible for maintaining. Make sure all instance variables and implementation specific methods are hidden from clients and do not become part of your responsibilities.

Rationale

This is one of the foundations of OO and software engineering.

ChiMu 97e

Operations

Operations formally define the exterior of an Object: what messages it can understand and the contract it agrees to if you send it that message[†]. Operations must be understandable in the context of the current Type (Interface or Class) and should be consistent across multiple Types. This requires continuous OO thinking, strong efforts toward standardization, and vigilant semantic verification. There are many more operations than their are Types, so it is a much more difficult task to name and organize them well.

[*] [Beck 96: Simple Superclass Name]
[†] [Meyer 96]

Choose Intention Revealing Operation Names

Choose "intention revealing operation names"* is the primary rule for naming operations. Always create a name that suggests what the operation "provides for the caller", not how a method could accomplish this service.

Rationale

The clearer you describe the behavior of the operation within its name, the easier it is for the clients (who repeatedly use those operations) to understand your class or interface. This is one of the many incarnations of thinking from the client's perspective.

ChiMu 97e, Beck 96

Have Uniquely Named Signatures

Ideally, make each operation have a unique name if it has different behavior or a different number or parameters. This will allow a client to know what this particular operation does and how many parameters it requires simply from the operation name. Especially avoid creating operations with the same name and the same number of arguments but different argument types.

Rationale

Otherwise, the human reader has to act like a compiler and figure out which operation (of several identically named ones) is the proper one to call given all the variable types involved. Because this is all done at compile-time, usually the result is not what anyone would want. See "Avoid overloading methods on argument type" for a good example in Java.

ChiMu 97e

Standardize Naming Patterns

Standardize the vocabulary used in operation names. As much as possible, words should be used consistently and uniquely when part of an operation.

Rationale

This makes understanding operations easier and supports precisely describing new functionality. These benefits become especially important as a system grows.

Details and Examples

The following is an example subset of the standard meanings for operation name parts and operation categories (see the Source code format section below).

The following are common operation prefixes

Prefixes	Category	Description
is, can, has, will	Testing	Return a Boolean and test the state of the object
new	Creating	Create and return a new object from a factory that creates only a single type of object
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.

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
add		Add an object to a collection

* See [Beck 96: Intention Revealing Selector] for a fuller description.

Non-prefix operation name patterns

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. Try to manage these forces well.

ChiMu 97e

Categorize your Operations

Group operations into Categories and reflect those categories in your Interfaces. If a language or tool does not support operation categories, use whatever documentation is available (comment dividers, notes, etc.).

Rationale

By grouping operations into meaningful categories it will be easier to understand the operations and to read the implementations. This organizational assistance is something akin to "subtyping" of operations. Some example categories are:

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

These categories should align with and reinforce operation naming patterns.

ChiMu 97e

Classes

Classes provide an implementation for a Type, so they should focus on the needs of implementers and maintainers[*]

Methods

Methods implement operations within a given class. As such, the rules for operations determine the name and other externally visible properties of a method. The rest of the

Compose Your Methods

After you have defined the public operations that a class has to perform, you will need to implement those operations with methods. Focus on communication and maintainability when implementing methods. "Divide your program into methods that perform one identifiable task. Keep all of the operations in a method at the same level of abstraction. This will naturally result in programs with many small methods, each a few lines long."[†]

ChiMu 97e, [Beck 96]

[*] If you choose to combine the two (have Classes be used as Types) then you will need to consider everyone's needs together

[†] [Beck 96: Composed Method]

Make Execution Structure Obvious

Try to make the execution structure of your method visible when looked at quickly. Standardize on a few good structures so a reader will quickly be able to survey the functionality and identify where to look for interesting features.

ChiMu 97e

3.5 Java-Oriented Guidelines

The following guidelines are more Java specific but also try to expose the general principles as much as possible.

Types, Interfaces, and Classes: Design

Interface with Interfaces

Use interfaces as the glue throughout your code instead of classes: define interfaces to describe the exterior of objects (i.e. their Type) and type all variables, parameters, and return values to interfaces.

Rationale

The most important reason to do this is that interfaces focus on the client's needs: interfaces define what functionality a client will receive from an Object without coupling the client to the Object's implementation. This is one of the core concepts to OO.

Details and Examples

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).

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. Java classes then become implementations of Java 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.

ChiMu 97e

Create Different Interfaces for Different Types of Clients

Provide different interfaces to support different types of clients and to prevent exposing responsibilities to clients who should not see it.

Rationale

Provides a more understandable system for a particular client's perspective and makes maintenance impacts more visible.

ChiMu 97e

Use Interfaces over Abstract Classes

If you can conceive of someone else implementing a class's functionality differently, define an interface, not an abstract class. Generally, use abstract classes only when they

are ‘‘partially abstract”; i.e., they implement some functionality that must be shared across all subclasses.

Rationale
Interfaces are more flexible than abstract classes. They support multiple inheritance and can be used as ‘mixins’ in otherwise unrelated classes.

Lea-1

Rarely declare a class final

Declare a class as final only if it is a subclass or implementation of a class or interface declaring all of its non-implementation-specific methods. (And similarly for final methods).

Rationale
Making a class final means that no one ever has a chance to reimplement functionality. Defining it instead to be a subclass of a base that is not final means that someone at least gets a chance to subclass the base with an alternate implementation. Which will essentially always happen in the long run.

Lea-1

Consider writing template files

Consider writing template files for the most common kinds of class files you create: Applets, library classes, application classes.

Rationale
Simplifies conformance to coding standards.

Lea-1

Operation and Method Design

The terms “operation” and “method” are used interchangeably (when referring to external specification) depending on who produced the guideline.

Weave Parameters Positions into the Operation Name

Put underscores "_" into operation names as placeholders for where a particular parameter is woven into the message send. Leave off any trailing underscores.

Examples and Details
For example:

```
at_put(key,value)
```

would read as at_(first parameter)put(second parameter) or "at (key) put (value)". A 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 operation name, they can be added at the end. For example:

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

Rationale
This improves operation/method names and helps clients know what order to put parameters into the parenthesis.

ChiMu 97e

Define return types as void

Define return types as void unless they return results that are not (easily) accessible otherwise (i.e., hardly ever write “return this”).

Rationale

While convenient, the resulting method cascades (a.meth1().meth2().meth3()) can be the sources of synchronization problems and other failed expectations about the states of target objects.

Lea-1

Avoid overloading methods on argument type

Avoid overloading methods on argument type. (Overriding on arity is OK, as in having a one-argument version versus a two-argument version). If you need to specialize behavior according to the class of an argument, consider instead choosing a general type for the nominal argument type (often Object) and using conditionals checking instanceof. Alternatives include techniques such as double-dispatching, or often best, reformulating methods (and/or those of their arguments) to remove dependence on exact argument type.

Rationale

Java method resolution is static; based on the listed types, not the actual types of argument. This is compounded in the case of non-Object types with coercion charts. In both cases, most programmers have not committed the matching rules to memory. The results can be counterintuitive; thus the source of subtle errors. For example, try to predict the output of this. Then compile and run.

```
class Classifier {
    String identify(Object x)  { return "object"; }
    String identify(Integer x) { return "integer"; }
}
```

```
class Relay {
    String relay(Object obj) {
        return (new Classifier()).identify(obj);
    }
}
```

```
public class App {
    public static void main(String[] args) {
        Relay relayer = new Relay();
        Integer i = new Integer(17);
        System.out.println(relayer.relay(i));
    }
}
```

Lea-1

Write methods that only do “one thing”

Write methods that only do “one thing”. In particular, separate out methods that change object state from those that just rely upon it. For a classic example in a Stack, prefer having two methods Object top() and void removeTop() versus the single method Object pop() that does both.

Rationale

This simplifies (sometimes, makes even possible) concurrency control and subclass-based extensions.

Lea-1

Packages and Modularity

Have a "Pack" Class for all Packages

Create a Class named "<packageName>Pack" for each Java package. Put documentation about the Package and any functionality that applies to the package as a whole into the Pack.

Rationale

Packages are not represented in Java in any tangible manner: they are more a hierarchical naming convention. By having a real Class for each package you have a standard place to put package documentation and functionality. This makes understanding a package easier and can also support better encapsulation of the package's functionality.

ChiMu 97e

Use Factories for Creating Objects

Use Factories and Factory methods for "public" object construction: have an object be responsible for construction instead of having clients directly call "new AClass()".

Rationale

The reason to use factory creation methods instead of straight constructors is because they:

- Allow more flexibility in "creating" a new object: the implementation can just reuse an existing object if the semantics make sense.
- 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])
- The implementation can take advantage of inheritance since it is a "normal" object method.

The Factory can either be an existing appropriate object (e.g. a database object is the factory for database Tables), or a specific Factory type. For classes that have no other appropriate factory object we use the 'Pack' object as the factory.

ChiMu 97e

*Minimize * forms of import*

Minimize * forms of import. Be precise about what you are importing. Check that all declared imports are actually used.

Rationale

Otherwise readers of your code will have a hard time understanding its context and dependencies. Some people even prefer not using import at all (thus requiring that every class reference be fully dot-qualified), which avoids all possible ambiguity at the expense of requiring more source code changes if package names change.

Lea-1

When sensible, consider writing a main for the principal class

When sensible, consider writing a main for the principal class in each program file. The main should provide a simple unit test or demo.

Rationale

Forms a basis for testing. Also provides usage examples.

Lea-1

The class with main should be separate from those containing normal classes.

For self-standing application programs, the class with main should be separate from those containing normal classes.

Rationale
Hard-wiring an application program in one of its component class files hinders reuse.

Lea-1

Method Implementation

Declare a local variable where you know its initial value

Declare a local variable only at that point in the code where you know what its initial value should be.

Rationale
Minimizes bad assumptions about values of variables.

Lea-1

Use a new local variable

Declare and initialize a new local variable rather than reusing (reassigning) an existing one whose value happens to no longer be used at that program point.

Rationale
Minimizes bad assumptions about values of variables.

Lea-1

Assign null to any reference variable that is no longer being used

Assign null to any reference variable that is no longer being used. (This includes, especially, elements of arrays.)

Rationale
Enables garbage collection.

Lea-1

Avoid assignments (‘‘=”) inside if and while conditions

Avoid assignments (‘‘=”) inside if and while conditions.

Rationale
They are almost always typos. The java compiler catches cases where ‘‘=” should have been ‘‘==” except when the variable is a boolean.

Lea-1

Ensure that there is ultimately a catch for all unchecked exceptions

Ensure that there is ultimately a catch for all unchecked exceptions that can be dealt with.

Rationale
Java allows you to not bother declaring or catching some common easily-handlable exceptions, for example java.util.NoSuchElementException. Declare and catch them anyway.

Lea-1

Embed casts in conditionals

Embed casts in conditionals.

Details and Examples

For example:

```
C cx = null;
if (x instanceof C) {
    cx = (C) x;
} else {
    evasiveAction();
}
```

Rationale

This forces you to consider what to do if the object is not an instance of the intended class rather than just generating a ClassCastException.

Lea-1

Documentation

Make code self documenting before commenting it

Try to make code as self document as possible before resorting to commenting it. This can both to improve the design and better describe an existing design than using comments.

Rationale

Specifications and code within a programming language are always more precise and useful than comments. If the code can describe itself better, this provides a constant reinforcement to future development (the clients and maintainers of this code). Comments are just auxiliary information and (although useful) should be a second choice.

ChiMu 97e, McConnell 92

Provide comments that augment, not repeat, program code

Make comments augment, not repeat, information available in Java syntax itself.

Rationale

Statements made in the programming language are precise, communicative, and guaranteed to be "true" (the program does what it says it does). If comments repeat information already specified they provide nothing and they are likely to become out of date and incorrect.

Details and Examples

Place the JavaDoc comments for methods 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 operator name, 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.

```
      /**
       * Find a person with the particular name.
       *@return null if can not find the person
       */
   public Person findName(String name);
```

Consider method comments to be inside and subservient to the declaration (although JavaDoc requires it to be before the declaration).

ChiMu 97e, McConnell 92

Use Interfaces for Public Documentation

If you "Interface with Interfaces", put public documentation in the Interface and implementation documentation in the class. Do not repeat documentation between the two files.

Rationale

Clients should only be looking at the public documentation and then the Class file can focus on implementation needs. Repeating information just makes it likely to get out of synch.

ChiMu 97e

Specify a standard keyword order

Specify a standard keyword order for your method modifiers.

Rationale

This is mostly a programming convention, but it can help team members to quickly understand other members programs. It can also help the mental classification of methods and reinforce other guidelines (e.g. avoid 'static').

Details and Examples

The following is a suggested order. Static methods are a completely different kind of method (they are actually statically bound functions and procedures), so 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>

ChiMu 97e

Augment javadoc keywords

Consider augmenting standard javadoc keywords (author, version, see, param) with additional descriptive keywords (require, ensure).

Rationale

Provides a more descriptive definition of the contract your class is providing to its clients. The compiler and running program can not use this information (without additional tools), but it supports the human communication among the developer, the clients, and future developers. See [Meyer 97] for the principles behind design by contract.

Details and Examples

If you are using standard javadoc, do not use the same '@' format for the new keywords (they will disappear), but instead capitalize them with a colon. The following is an example of augmenting keywords.

```
      /**
       * Remove and return the top element.
       *<P>REQUIRE: notEmpty()
       *<P>ENSURE:  NEW(count()) = OLD(count()) - 1
       */
  public Object pop();
```

ChiMu 97e

Class Implementation

Never declare instance variables as public

Never declare instance variables as public.

Rationale

The standard OO reasons. Making variables public gives up control over internal class structure. Also, methods cannot assume that variables have valid values.

Lea-1

Minimize statics

Minimize statics (except for static final constants).

Rationale

Static variables act like globals in non-OO languages. They make methods more context-dependent, hide possible side-effects, sometimes present synchronized access problems. and are the source of fragile, non-extensible constructions. Also, neither static variables nor methods are overridable in any useful sense in subclasses.

Lea-1

Prefer protected to private

Generally prefer protected to private.

Rationale

Unless you have good reason for sealing-in a particular strategy for using a variable or method, you might as well plan for change via subclassing. On the other hand, this almost always entails more work. Basing other code in a base class around protected variables and methods is harder, since you you have to either loosen or check assumptions about their properties. (Note that in Java, protected methods are also accessible from unrelated classes in the same package. There is hardly every any reason to exploit this though.)

Lea-1

Minimize reliance on implicit initializers

Minimize reliance on implicit initializers for instance variables (such as the fact that reference variables are initialized to null).

Rationale

Minimizes initialization errors.

Lea-1

Prefer abstract methods to those with default no-op implementations

Prefer abstract methods in base classes to those with default no-op implementations. (Also, if there is a common default implementation, consider instead writing it as a protected method so that subclass authors can just write a one-line implementation to call the default.)

Rationale

The Java compiler will force subclass authors to implement abstract methods, avoiding problems occurring when they forget to do so but should have.

Lea-1

Avoid giving a variable the same name as one in a superclass

Avoid giving a variable the same name as one in a superclass.

Rationale
This is usually an error. If not, explain the intent.

Lea-1

Use final and/or comment conventions for instance variables

Use final and/or comment conventions to indicate whether instance variables that never have their values changed after construction are intended to be constant (immutable) for the lifetime of the object (versus those that just so happen not to get assigned in a class, but could in a subclass).

Rationale
Access to immutable instance variables generally does not require any synchronization control, but others generally do.

Lea-1

Avoid unnecessary instance variable access and update methods

Avoid unnecessary instance variable access and update methods. Write get/set-style methods only when they are intrinsic aspects of functionality.

Rationale
Most instance variables in most classes must maintain values that are dependent on those of other instance variables. Allowing them to be read or written in isolation makes it harder to ensure that consistent sets of values are always used.

Lea-1

Minimize direct internal access to instance variables inside methods

Minimize direct internal access to instance variables inside methods. Use protected access and update methods instead (or sometimes public ones if they exist anyway).

Rationale
While inconvenient and sometimes overkill, this allows you to vary synchronization and notification policies associated with variable access and change in the class and/or its subclasses, which is otherwise a serious impediment to extensibility in concurrent OO programming. (Note: The naming conventions for instance variables serve as an annoying reminder of such issues.)

Lea-1

Ensure that non-private statics have sensible values

Ensure that non-private statics have sensible values even if no instances are ever created. (Similarly ensure that static methods can be executed sensibly.) Use static intitializers (static { ... }) if necessary.

Rationale
You cannot assume that non-private statics will be accessed only after instances are constructed.

Lea-1

Consider whether any class should implement Cloneable and/or Serializable.

Consider whether any class should implement Cloneable and/or Serializable.

Rationale
These are ‘‘magic” interfaces in Java, that automatically add possibly-needed functionality only if so requested.

Lea-1

Whenever reasonable, define a default (no-argument) constructor

Whenever reasonable, define a default (no-argument) constructor so objects can be created via Class.newInstance().

Rationale
This allows classes of types unknown at compile time to be dynamically loaded and instantiated (as is done for example when loading unknown Applets from html pages).

Lea-1

Overriding

If you override Object.equals, also override Object.hashCode

If you override Object.equals, also override Object.hashCode, and vice-versa.

Rationale
Essentially all containers and other utilities that group or compare objects in ways depending on equality rely on hashcodes to indicate possible equality. For further guidance see Taligent’s Java Cookbook

Lea-1

Override readObject and WriteObject if a Serializable class relies on process state

Override readObject and WriteObject if a Serializable class relies on any state that could differ across processes, including, in particular, hashCodes and transient fields.

Rationale
Otherwise, objects of the class will not transport properly.

Lea-1

Explicitly define clone()

If you think that clone() may be called in a class you write, then explicitly define it (and declare the class to implement Cloneable).

Rationale
The default shallow-copy version of clone might not do what you want.

Lea-1

Miscellaneous

Generally prefer long to int, and double to float

Generally prefer long to int, and double to float. But use int for compatibility with standard Java constructs and classes (for the major example, array indexing, and all of the things this implies, for example about maximum sizes of arrays, etc).

Rationale
Arithmetic overflow and underflow can be 4 billion times less likely with longs than ints; similarly, fewer precision problems occur with doubles than floats. On the other hand, because of limitations in Java

atomicity guarantees, use of longs and doubles must be synchronized in cases where use of ints and floats sometimes would not be.

Lea-1

Use method equals instead of operator == when comparing objects

Use method equals instead of operator == when comparing objects. In particular, do not use == to compare Strings.

Rationale

If someone defined an equals method to compare objects, then they want you to use it. Otherwise, the default implementation of Object.equals is just to use ==.

Lea-1

Prefer declaring arrays as Type[] arrayName rather than Type arrayName[].

Prefer declaring arrays as Type[] arrayName rather than Type arrayName[].

Rationale

The second form is just for incorrigible C programmers.

Lea-1

Concurrent Programming

Doug Lea specializes in Concurrent Programming. See his book [Lea 96] for more information on Concurrent Programming in Java.

Declare all public methods as synchronized

Declare all public methods as synchronized; or if not, describe the assumed invocation context and/or rationale for lack of synchronization.

Rationale

In the absence of planning out a set of concurrency control policies, declaring methods as synchronized at least guarantees safety (although not necessarily liveness) in concurrent contexts (every Java program is concurrent to at least some minimal extent). With full synchronization of all methods, the methods may lock up, but the object can never enter in randomly inconsistent states (and thus engage in stupidly or even dangerously wrong behaviors) due to concurrency conflicts. If you are worried about efficiency problems due to synchronization, learn enough about concurrent OO programming to plan out more efficient and/or less deadlock-prone policies.

Lea-1

Prefer synchronized methods to synchronized blocks.

Prefer synchronized methods to synchronized blocks.

Rationale

Better encapsulation; less prone to subclassing snags; can be more efficient.

Lea-1

Always embed wait statements in while loops that re-wait

Always embed wait statements in while loops that re-wait if the condition being waited for does not hold.

Rationale

When a wait wakes up, it does not know if the condition it is waiting for is true or not.

Lea-1

Use notifyAll instead of notify or resume.

Use notifyAll instead of notify or resume.

Rationale

Classes that use only notify can normally only support at most one kind of wait condition across all methods in the class and all possible subclasses. And unguarded suspends/resumes are even more fragile.

Lea-1

Always document the fact that a method invokes wait

Always document the fact that a method invokes wait

Rationale

Clients may need to take special actions to avoid nested monitor calls.

Lea-1

3.6 Methodology, Notation, and CASE Guidelines

The following are some guidelines for working with methodologies, notations and CASE tools.

Use UML

Use UML because it is the standard for OO notation.

Rationale

With deference to all the great work by other methodologists, engineers, and “thinkers”, UML has won. The probability that someone understands what you are doing is significantly enhanced by having a (good-enough) core shared notation. Start with UML and then enhance it for needs it does not address.

ChiMu 97e

Go beyond UML

If UML does not express a concept well, extend it or modify it and then document how your extension relates to UML and other methods.

Rationale

Communication is the most important part of any notation. Make sure a standard does not hinder communicating concepts that are important to building good OO software. Although UML incorporates several peoples’ work together, other important concepts were left out (e.g. UI, Coordinator, Entity, Interface, Extensions, etc.) and should not be lost if they are important to your development process.

Details and Examples

UML is extendable via the stereotype functionality: you can further refine UML concepts by annotating a UML object with a guillemot surrounded phrase («coordinator»). Further, you can then define a new icon for the new refined concept. This, for example, allows you to use Jacobson concepts by adding «interface», «control», «entity» stereotypes and then using Jacobson icons for these new concepts.

ChiMu 97e

Don’t draw models for everything

Don’t draw models for everything; instead, concentrate on the key areas. It is better to have a few diagrams that you use and keep up-to-date than to have many forgotten, obsolete models.

Fowler 97b

Link your design diagrams to javadoc

If you create a diagram to explain an important concept for your design, connect the diagram into the Javadoc documentation.

Rationale

Javadoc is THE standard reference source for Java programming, so any documentation available within it will be much more likely to find.

Lea-2

Use a smart drawing tool

Use a smart drawing tool to make UML and other software design diagrams easier to create and modify.

Rationale

Smart drawing tools are the first logical step for creating software models above the simplicity and flexibility of the simple pen. They provide excellent support for creating, modifying, and printing readable diagrams quickly through their stencils, smart line connectors, and excellent drawing capabilities. Most smart diagram tools now have stencils for UML, Objectory, ER Diagrams, UI widgets, and many other types of software notations. Some tools even have a simple understanding of UML notation rules.

Details

There are many smart drawing tools available, and listings of them can be found on the web at:

1. http://www.yahoo.com/Computers_and_Internet/Software/Reviews/Titles/Business/Flow_Chart/
2. http://www.yahoo.com/Business_and_Economy/Companies/Computers/Software/Graphics/
3. http://www.yahoo.com/Business_and_Economy/Companies/Computers/Software/Graphics/Flow_Charting/

ChiMu 97e

Use a CASE tool

If your diagramming needs go beyond a smart drawing tool, carefully consider the different CASE tools on the market and decide which (if any) meet your needs the best compared to the drawing tools.

Rationale

For some projects, smart drawing tools will not be enough and CASE tools may be a more appropriate solution. Smart drawing tools are very powerful but they do not understand the meaning of what they portray. This restricts their abilities to:

4. Guide/enforce the user to create correct diagrams
5. Update changes between different diagrams
6. Support creating new diagrams based on the existing knowledge
7. Create automatic reports
8. Forward generate to code

All of these items might be useful for the project team to maintain its models. Be aware that all tools that are smarter are also less flexible and are never “smart enough”, so you need to make a very careful consideration of the tradeoffs with each tool and which of the above items are actually useful (in practice) to your project.

ChiMu 97e

Avoid taking liberties with a CASE tool

Avoid taking liberties with a CASE tool or using it as a general diagramming tool.

Rationale
All CASE tool diagrams impact the conceptual model in the “repository”, and if even one diagram is done strangely/loosely that model will be damaged. This could prevent others from understanding the model, or lead to a severe misunderstanding of the model. Make sure the repository is correct from all views.

Details
It is better to capture information that the CASE tool can not in a different (external to the tool) form. Put as much in the tool as possible and then organize and refer to this external information.

ChiMu 97e

Recognize the limits of CASE tools

Make sure you recognize what a CASE tool prevents you from expressing and determine when that is and is not acceptable. Use hand drawings, drawing tools, or just text when you your CASE tool would hinder important communication.

Rationale
CASE tools provide many benefits but they can sometimes interfere with the goal of good communication. No CASE tool can completely support all of UML, let alone all the useful ways to communicate precisely. The communication is more important than the tool.

ChiMu 97e

Drive CASE tools from the appropriate direction

Drive CASE tools from the appropriate direction. Before a significant amount of code is written, CASE tools can be driven forward (requirements to analysis to design to implementation). After code exists, drive design information from the code.

Rationale
The forward direction is the ideal, but a CASE repository is of no use later in a project if it does not reflect reality. Code is the reality; so CASE information must be generated from the code. The changes can then be reviewed as part of code review and acceptance. Previous designs can be kept as snapshots and new future designs can again drive code changes.

ChiMu 97e

3.7 Final Guidelines

Try it out

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

Rationale
Only by trying a guideline can you really understand how and whether it benefits you.

ChiMu 97e

Prove Performance

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

Rationale
Optimization is always harmful and rarely beneficial unless you have numbers to back it up. A better design (which these guideline try to encourage) will allow more precise and effective optimizations later when the numbers come in.

ChiMu 97e

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.

Rationale

Having standards that interfere with good system design is worse than no standard at all.

ChiMu 97e

Do not require 100% conformance to rules of thumb!

Do not require 100% conformance to rules of thumb such as the ones listed here!

Rationale

Java allows you program in ways that do not conform to these rules for good reason. Sometimes they provide the only reasonable ways to implement things. And some of these rules make programs less efficient than they might otherwise be, so are meant to be conscientiously broken when performance is an issue.

Lea-1

4 Definitions

Having a common glossary of terms is important for accurate, precise, and concise communication among team members. The following definitions are ChiMu's distilling and reconciliation of the many great concepts and work that have been contributed to OO*. Many of the sources for these terms are mentioned in the References section of this document. A few particular notable sources are:

The Dictionary of Object Technology [Firesmith+E 95].
UML: The Unified Modeling Language [Rational 98]
Design Patterns, especially [Gamma+HJV 96]

4.1 Categorized

The following sections categorize definitions by starting with the core OO concepts (Objects) and growing to more domain-specific definitions.

Object

The core concept to OO is Objects. Everything else in OO is to make understanding, managing, or implementing Objects easier. The definition of Object is the foundation for all other definitions.

Definitions

Object	An identifiable, encapsulated entity that can only be interacted with by sending messages.
Message	A stimulus sent to an object with a name and any parameters (as Objects) that the message requires. A message will cause the receiver Object to return an answer or nothing. In most Object Languages, the sender has to wait for the answer before continuing.
Identity	The ability to tell one object from another object independently of whether their appearance (behavior) is identical.
Behavior	The response of an object to a stimulus. An object's behavior is the answers it gives to messages both now and in the future. (See State).
State	An abstraction to describe and simplify understanding an object's behavior. An object's behavior can be described as the answers it gives to current messages (which are determined by the current state) and the changes to its state caused by these messages.

More Information

The best source for the concept of Object is Alan Kay's writings [Kay 95] and Smalltalk itself [Goldberg+R 83, Squeak]. You could also look at Simula (the progenitor of Smalltalk), but it has other concepts mixed into the language besides Objects. Some methodologies focus on Objects more than others [Wirfs-Brock+WW 90, Wilkinson 95], but all of the main OO methodology books (e.g. [Booch 95, Rumbaugh+BPEL 91, Jacobson+CJO 92]) have objects at the core.

Type

Types allow you to think about the commonality of objects' exteriors. They are the first conceptual abstraction above Objects and immediately provide an enormous amount of ability to reasoning about

* Fortunately, during the last few years a lot of unification has occurred and reconciling differences is less difficult.

Objects. The amount of abstraction and formality associated with Types can depend on of the project or the current perspective.

Definitions

Type	Describes a common exterior (public behavior) of a set of Objects. Can also be used to conceptually group and understand objects by their similar behavior.
Operation	A description of the ability for an object to respond to a particular message and the contract/requirement for that message.
Interface	A description of a Type focused on the Operations that the objects can respond to.
Is-A	An object is a Type if an Object supports all the exterior requirements of that Type.
Extend	To define a new Type (called a Subtype) in terms of an existing type (called a Supertype). The new Subtype will have the same contract (operations) as the Supertype but can add new functionality: as either new operations or enhancement in capability to existing operations.
Subtype	A Type that extends another Type (called the Supertype).
Supertype	A Type that has been extended by another Type.

More Information

Many methodologies and authors do not make a clear distinction between Type and Class (usually referring to both a "Class" and the context determines what was really meant), so this can cause confusion. The following discuss Types independently of Classes [Kilov+R 94, Fowler 97, Jacobson+CJO 92, Cook+D 94], but all of the main methodology books do focus on Types during analysis [Booch 94, Fowler 97, Rumbaugh+BPEL 91]. The description of contracts and interfaces in [Meyer 97] is the standard and an excellent reference on formal Types and their integration into OOP (but you have to be willing to accept many differences in terminology). Java supports the separation of Type from Class through interfaces and classes, see [Coad+M 96, Gosling+JS 96].

Class

Classes provide a common implementation for similar objects. They describe the interior of objects so it is easier to work with many objects of many different Types. It is important to conceptually separate Classes from Types even though it is a common (bad) practice to use Class for both ideas.

Definitions

Class	Provides a common implementation for a set of objects.
Method	An implementation of an operation for a particular object/class.
Instance Variable	A way to store encapsulated state information for a particular object. Instance variables are completely hidden within the Object, but they enable two objects of the same Class to have different external Behavior.
Instance	An object is an Instance of a Type if an object supports all the exterior requirements of that type (see "Is-A"). An object is an instance of a Class if it is implemented by that class.
Extent	The collection of all instances of a Type or Class.
Inherit	To define a new Class in terms of an Existing Class (the Superclass) by starting with the Superclass's implementation and overiding or adding to it.

More Information

All the methodologies focus on Classes, so any of them would be good for more information. Many are bad at separating the concept of Class from Type (see above under "Type"). Class is the most common term in OO Design and Analysis

Relationship Modeling

The above Type and Class concepts are just the beginning of the tools that can be used to reason about and implement objects. Relationship Modeling adds on the ability to describe relationships among Objects and Types of Objects.

Definitions

Link	A connection between two objects which allows one or both to know about the other object. By default links are assumed to be bidirectional in analysis, but they can be defined to be only traversable in one direction.
Traverse	To move from one object to another by a Link. If a Link is traversable from an Object than that Object can get to (knows about) the other Object.
Association$_1$	A relationship between two Types that allows or requires Objects of those Types to be Linked.
Role	The name of the "position" within a relationship an Object or Type holds. For example, a binary association has two roles that distinguish the two participants in the relationship.
Association$_2$	An Association$_1$, but must be between Types which have Objects with Identity. See Attribute and ValueObject.
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 (most numbers, 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.
Immutable	Can not be changed after being created. Immutable objects can not be changed after they are created and fully initialized.
Attribute$_1$	A public property of an object that shows an aspect of the state of the object. Frequently there is a minimal collection of attributes that uniquely determine the state of the object. See also Property.
Attribute$_2$	See BasicAttribute.
Attribute$_3$	See Instance Variable.
BasicAttribute	An Attribute$_1$ that takes its value from ValueObjects. This is as opposed to associations, which connect two or more objects with identity. A BasicAttribute is traversable only from the Object to the ValueObject.
Property	Synonym for Attribute$_1$ and sometimes for Attribute$_2$.
Feature	The Eiffel term for Operation where Operation includes both methods and attributes$_1$.

More Information

UML [Fowler 97, Rational 98] is now the primary source for the base concepts of Relationship Modeling in OO. More detailed modeling is done in other methods (for example, see [Kilov+R 95]). Many sources provide concepts for and examples of relationship modeling in particular domain areas (see the Patterns references). The biggest problem with Relationship modeling terminology is the overloaded and fuzzy meaning of "attribute", so be careful to consider what meaning was intended in a given context.

Patterns

The following list has definitions for terms that were codified into OO language through well-known Design Patterns. There are many more Patterns than the following definitions, but these are among the most common and accepted terms.

Definitions

Adapter	An object that can convert an Interface of one Class to the interface another Object expects.
Factory	An object that can create other objects.
Functor	An object that models an operation.
Observable	An object that can be Observed. See Observer.
Observer	An object that "looks at" another object (the Observable) and can respond to events in the Observable without the Observable being knowledgeable about the Observer.
Prototype	An object that is used as a template for creating other Objects.
Proxy	An object that stands in for another object (the RealObject) and manages the client interaction with the RealObject.
Singleton	An object that is the only instance of a Class.
Strategy	An object that encapsulates an algorithm to be used with an Object.
Visitor	An object that represents an operation that can be performed on the elements of an Object structure (frequently a hierarchy or sequence).

More Information

See [Gamma+HJV 96] for the full patterns to most of these definitions. Some definitions may also be covered in other Pattern resources and Patterns are frequently the source of new terminology.

Pattern: Proxy

The Proxy pattern is important to Information Systems, so the following provides some more details.

Definitions

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 acts as 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

More Information

See [Gamma+HJV 96] for the basic proxy concept. See information on distributed processing (e.g. [GemStone 95]) and Object-Relational Mapping (e.g. [Fussell 96, Fussell 97a]) for details on the Proxy variations.

Pattern: Functor

The Functor pattern is common to all types of OO Systems. The following provide some more definitions.

Definitions

Functor	"An object that models an operation" [Firesmith+E 95]. For a Java implementation, a basic functor is an Interface with a single, generic, operation.
Procedure$_f$	A functor that does not return a value
Function$_f$	A functor that returns an Object
Predicate$_f$	A functor that returns a boolean
Getter$_f$	A functor that is designed to retrieve a value from an object (the first parameter)
Setter$_f$	A functor that stores into an object (the first parameter) a value (the second parameter)

More Information

Note that almost all of the definitions above also have a more general meaning, so to be specific you would need to append "Functor" to the end to be precise (e.g. "a Predicate Functor is a functor that returns a boolean")

Architecture

The following terms are related to system architecture.

Definitions

Architecture	A system's concepts, structures, and interactions. Also, the description of a system's desired architecture before construction.
Framework	A strong partition of generalized functionality common to many parts of an application. Also, more formally, a collection of interacting classes that describes most of the behavior a client requires and can be subclassed and parameterized to customize and complete the functionality.
Layer	A logical, horizontal division of a system that provides a particular system abstraction to the client above the layer.
Module	A base level subsystem: one which does not contain any other subsystems

Partition	A vertical division of a system into areas of related functionality.
Subsystem	A division of a system into a cohesive unit of functionality (tightly related classes and internal subsystems) with a public interface and a private implementation.
Tier	A level on a hierarchy of processes over which a system is divided.

More Information

See the resources under Architecture (e.g. [Shaw+G 96, Fussell 96]) and also some of the large-picture methodology and project-management books (e.g. [Booch 96, Jacobson+CJO 92]).

Object-Oriented Information Systems

Object-Oriented Information Systems use Objects (called DomainObjects, BusinessObjects, or EntityObjects) to capture the knowledge, operations, and rules about a business within a computer.

Definitions

ObjectBase	.An ObjectBase captures the knowledge, operations, and rules required to usefully represent a particular part of the world in a computer. An ObjectBase contains all the objects that represent a particular state of your DomainModel and all the knowledge contained therein. Also called an ObjectSpace.
DomainModel	All the static rules, constraints, and operations that apply to DomainObjects. The DomainModel can either be conceptual to help understand the behavior of DomainObjects or it can be implemented as DomainClasses.
DomainObject	An object which captures knowledge about a domain. DomainObjects allow a computer to inspect, imply, modify, and "reason" about that information in either very simple ways (the facts) or more complex ways (the rules and implications).

More Information

See [Fussell 96].

Other Terms

The following are some currently unclassified terms.

Definitions

Registry	An object that remembers other objects and can search through and retrieve them through one or more properties. Usually the objects within a Registry are all of the same type.
Container	A Registry but without the implication of a primary registration property (e.g. a "key").
ExtentRegistry	A Registry that contains all the objects of a Type (the Extent of the Type). An ExtentRegistry is a close equivalent to a RelVar or Table in the context of Objects.
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.

Java Terms

The following terms are very common in Java and not mentioned elsewhere.

Definitions

Bean An Object that knows about its own properties (can introspect) and has several other capabilities. Any Java Object can be a Bean but some Objects have more Bean functionality.

More Information

See the Java JDKs and information

4.2 Alphabetical

The following table contains all the previous mentioned definitions combined and sorted alphabetically.

Adapter	An object that can convert an Interface of one Class to the interface another Object expects.
Architecture	A system's concepts, structures, and interactions. Also, the description of a system's desired architecture before construction.
Association$_1$	A relationship between two Types that allows or requires Objects of those Types to be Linked.
Association$_2$	An Association$_1$, but must be between Types which have Objects with Identity. See Attribute and ValueObject.
Attribute$_1$	A public property of an object that shows an aspect of the state of the object. Frequently there is a minimal collection of attributes that uniquely determine the state of the object. See also Property.
Attribute$_2$	See BasicAttribute.
Attribute$_3$	See Instance Variable.
BasicAttribute	An Attribute$_1$ that takes its value from ValueObjects. This is as opposed to associations, which connect two or more objects with identity. A BasicAttribute is traversable only from the Object to the ValueObject.
Bean	An Object that knows about its own properties (can introspect) and has several other capabilities. Any Java Object can be a Bean but some Objects have more Bean functionality.
Behavior	The response of an object to a stimulus. An object's behavior is the answers it gives to messages both now and in the future. (See State).
Binding	Associating a client object to a database object, which turns the client object into a Proxy
Class	Provides a common implementation for a set of objects.
Container	A Registry but without the implication of a primary registration property (e.g. a "key").
DomainModel	All the static rules, constraints, and operations that apply to DomainObjects. The DomainModel can either be conceptual to help understand the behavior of DomainObjects or it can be implemented as DomainClasses.
DomainObject	An object which captures knowledge about a domain. DomainObjects allow a computer to inspect, imply, modify, and "reason" about that information in either very simple ways (the facts) or more complex ways (the rules and implications).
Extend	To define a new Type (called a Subtype) in terms of an existing type (called a Supertype). The new Subtype will have the same contract (operations) as the Supertype but can add new functionality: as either new operations or enhancement in capability to existing operations.
Extent	The collection of all instances of a Type or Class.
ExtentRegistry	A Registry that contains all the objects of a Type (the Extent of the Type). An ExtentRegistry is a close equivalent to a RelVar or Table in the context of Objects.
Factory	An object that can create other objects.
Feature	The Eiffel term for Operation where Operation includes both methods and attributes$_1$.
Forwarder	A proxy which immediately forwards messages, possibly over process and machine boundaries, to the RealSubject.
Framework	A strong partition of generalized functionality common to many parts of an application. Also, more formally, a collection of interacting classes that describes most of the behavior a client requires and can be subclassed and parameterized to customize and complete the functionality.
Function$_f$	A functor that returns an Object
Functor	An object that models an operation.
Functor	"An object that models an operation" [Firesmith+E 95]. For a Java implementation, a basic functor is an Interface with a single, generic, operation.
Getter$_f$	A functor that is designed to retrieve a value from an object (the first parameter)
Identity	The ability to tell one object from another object independently of whether their appearance (behavior) is identical.
IdentityKey	A value that defines the RealIdentity of a Proxy.
Immutable	Can not be changed after being created. Immutable objects can not be changed after they are created and fully initialized.
Inherit	To define a new Class in terms of an Existing Class (the Superclass) by starting with the Superclass's implementation and overiding or adding to it.
Instance	An object is an Instance of a Type if an object supports all the exterior requirements of that

type (see “Is-A”). An object is an instance of a Class if it is implemented by that class.

Instance Variable A way to store encapsulated state information for a particular object. Instance variables are completely hidden within the Object, but they enable two objects of the same Class to have different external Behavior.

Interface A description of a Type focused on the Operations that the objects can respond to.

Is-A An object is a Type if an Object supports all the exterior requirements of that Type.

Layer A logical, horizontal division of a system that provides a particular system abstraction to the client above the layer.

Link A connection between two objects which allows one or both to know about the other object. By default links are assumed to be bidirectional in analysis, but they can be defined to be only traversable in one direction.

Message A stimulus sent to an object with a name and any parameters (as Objects) that the message requires. A message will cause the receiver Object to return an answer or nothing. In most Object Languages, the sender has to wait for the answer before continuing.

Method An implementation of an operation for a particular object/class.

Module A base level subsystem: one which does not contain any other subsystems

Object An identifiable, encapsulated entity that can only be interacted with by sending messages.

ObjectBase .An ObjectBase captures the knowledge, operations, and rules required to usefully represent a particular part of the world in a computer. An ObjectBase contains all the objects that represent a particular state of your DomainModel and all the knowledge contained therein. Also called an ObjectSpace.

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.

Observable An object that can be Observed. See Observer.

Observer An object that “looks at” another object (the Observable) and can respond to events in the Observable without the Observable being knowledgeable about the Observer.

Operation A description of the ability for an object to respond to a particular message and the contract/requirement for that message.

Partition A vertical division of a system into areas of related functionality.

Predicate$_f$ A functor that returns a boolean

Procedure$_f$ A functor that does not return a value

Property Synonym for Attribute$_1$ and sometimes for Attribute$_2$.

Prototype An object that is used as a template for creating other Objects.

Proxy An object that stands in for another object (the RealObject) and manages the client interaction with the RealObject.

Proxy An object that stands in for another object (the RealObject) and manages the client interaction with the RealObject.

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.

Registry An object that remembers other objects and can search through and retrieve them through one or more properties. Usually the objects within a Registry are all of the same type.

Replicate A proxy which holds local state and performs local operations which are later propagated to the RealSubject

Role The name of the “position” within a relationship an Object or Type holds. For example, a binary association has two roles that distinguish the two participants in the relationship.

Setter$_f$ A functor that stores into an object (the first parameter) a value (the second parameter)

Singleton An object that is the only instance of a Class.

State An abstraction to describe and simplify understanding an object’s behavior. An object’s behavior can be described as the answers it gives to current messages (which are determined by the current state) and the changes to its state caused by these messages.

Strategy An object that encapsulates an algorithm to be used with an Object.

Stub A proxy which acts as a placeholder for the RealObject and must become another type of proxy (for example, forwarder or replicate) when interacted with by a client.

Subsystem A division of a system into a cohesive unit of functionality (tightly related classes and internal subsystems) with a public interface and a private implementation.

Subtype A Type that extends another Type (called the Supertype).

Supertype A Type that has been extended by another Type.

Tier A level on a hierarchy of processes over which a system is divided.

Traverse To move from one object to another by a Link. If a Link is traversable from an Object than that Object can get to (knows about) the other Object.

Type	Describes a common exterior (public behavior) of a set of Objects. Can also be used to conceptually group and understand objects by their similar behavior.
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 (most numbers, 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.
Visitor	An object that represents an operation that can be performed on the elements of an Object structure (frequently a hierarchy or sequence).

5 Reading and References

5.1 Categorized Reading

Architecture

System Architecting: Creating and Building Complex Systems	**Rechtin 91**
"Software Architecture Bibliography"	**SEI**
Software Architecture	**Shaw+G 96**
Software Architecture and Design: Principles, Models, and Method.	**Witt+BM 94**

Information Modeling

Information Modeling: An Object-Oriented Approach.	**Kilov+R 94**
An Introduction to Database Systems.	**Date 95**
The Relational Model for Database Management, Version 2.	**Codd 90**
The Object Database Standard: ODMG 2.0.	**Cattell+D 97**
Object-Oriented Analysis and Design with Applications.	**Booch 94**
Object-Oriented Modeling and Design.	**Rumbaugh+BPEL 91**
Analysis Patterns: Reusable Object Models.	**Fowler 97**
Modern Database Systems: The Object Model, Interoperability, and Beyond	**Kim 95**

Relational Modeling and Databases

An Introduction to Database Systems.	**Date 95**
The Relational Model for Database Management, Version 2.	**Codd 90**
Relational Database: Selected Writing	**Date 86**
Relational Database Writings, 1985-1989	**Date 90**
Relational Database Writings, 1989-1991	**Date 92**
Relational Database Writings, 1991-1994	**Date 95b**
A Guide to The SQL Standard	**Date+D 97**

Object-Oriented Design and Analysis

Object-Oriented Analysis and Design with Applications	**Booch 94**
Java Design: Building Better Apps & Applets	**Coad+M 96**
Designing Object Systems: Object-Oriented Modeling with Syntropy	**Cook+D 94**
Dictionary of Object Technology: The Definitive Desk Reference.	**Firesmith+E 95**
Design Patterns: Elements of Object-Oriented Architecture.	**Gamma+HJV 95**
Object-Oriented Software Engineering: A Use Case Driven Approach.	**Jacobson+CJO 92**
The Art of the Metaobject Protocol.	**Kiczales+RB 91**
Object Oriented Software Construction, 2nd Edition.	**Meyer 97**

Object-Oriented Modeling and Design. **Rumbaugh+BPEL 91**
Designing Object-Oriented Software **Wirfs-Brock+WW 90**

UML

UML Distilled: Applying the Standard Object Modeling Language **Fowler 97**
Understanding UML: The Developer's Guide **Harmon+W 98**
Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design **Larman 97**
UML: Unified Modeling Language, Version 1.1 **Rational 98**

CRC

Using CRC Cards: An Informal Approach to Object-Oriented Development **Wilkinson 95**
The CRC Card Book **Bellin+S 97**
Designing Object-Oriented Software **Wirfs-Brock+WW 90**

Patterns

Design Patterns: Elements of Object-Oriented Architecture **Gamma+HJV 95**
Pattern Languages of Program Design **Coplien+S 95**
Pattern Languages of Program Design 2. **Vlissides+CK 96**
Pattern Languages of Program Design 3 **Martin+RB 98**
A System of Patterns: Pattern-Oriented Software Architecture **Buschmann+MRSS 96**
CORBA Design Patterns. **Mowbray+M 97**
Analysis Patterns: Reusable Object Models. **Fowler 97**

Object-Oriented Programming and Languages

History of Programming Languages – II. **Bergin+G 96**
Smalltalk-80: The Language and its Implementation. **Goldberg+R 83**
The Java™ Language Specification. **Gosling+JS 96**
Eiffel, The Language **Meyer 92**
Object Oriented Software Construction, 2nd Edition. **Meyer 97**
Object-Oriented Programming: The CLOS Perspective. **Paepcke 93**
LISP. **Winston+H 81**
Structure and Interpretation of Computer Programs. **Abelson+S 96**
Smalltalk Best Practice Patterns, Volume 1: Coding **Beck 96**

Programming

Structure and Interpretation of Computer Programs. **Abelson+S 96**
Code Complete: A Practical Handbook of Software Construction **McConnell 93**

Project Management

Object Solutions: Managing the Object-Oriented Project **Booch 96**

The Mythical Man-Month	**Brooks 75**
Constantine on Peopleware	**Constantine 95**
201 Principles of Software Development	**Davis 95**
Controlling Software Projects	**DeMarco 82**
Peopleware: Productive Projects and Teams	**DeMarco+L 87**
Managing the Software Process	**Humphrey 89**
Managing Technical People	**Humphrey 97**
Debugging the Development Process	**Maguire 94**
Dynamics of Software Development	**McCarthy 95**
Rapid Development	**McConnell 96**
Software Project Survival Guide	**McConnell 98**
How to Run Successful Projects	**O'Connell 94**
Pitfalls of Object-Oriented Development	**Webster 95**

Client/Server Systems

Distributed Object-Oriented Data-Systems Design	**Andleigh+G 92**
CORBA: A Guide to the Common Object Request Broker Architecture	**Ben-Natan 95**
Client/Server Architecture	**Berson 92**
Firewalls and Internet Security	**Cheswick+B 94**
Distributed Systems: Concepts and Design	**Coulouris+DK 94**
CORBA Design Patterns.	**Mowbray+M 97**
Inside CORBA: Distributed Object Standards and Applications	**Mowbray+R 97**
Distributed Systems	**Mullender 93**
Essential Client/Server Survival Guide	**Orfali+HE 94**
Introduction to Client/Server Systems: A Practical Guide for Systems Professionals	**Renaud 93**
Enterprise Computing with Objects: From Client/Server Environments to the Internet. .	**Shan+E 98**

UI: Human Factors

Proceeding of CHI, 1985-98	**ACM-CHI**
Readings in Human-Computer Interaction: A Multidisciplinary Approach	**Baecker+B 87**
Reading in Human-Computer Interaction: Toward the Year 2000	**Baecker+GBG 95**
User Interface Design	**Cox+W 93**
Computers as Theatre	**Laurel 91**
The Elements of User Interface Design	**Mandel 97**
Designing Visual Interfaces: Communication Oriented Techniques	**Mullet+S 95**
Usability Engineering	**Nielsen 93**
The Design of Everyday Things	**Norman 88**
Turn Signals Are the Facial Expressions of Automobiles	**Norman 92**
User Centered System Design	**Norman+D 86**

Designing the User Interface: Strategies for Effective Human-Computer Interaction **Shneiderman 98**

TOG on Interface **Tognazzini 92**

UI: Specific Guidelines

Macintosh Human Interface Guidelines **Apple 92**

Inside Taligent Technology **Cotter+P 95**

User-Interface Screen Design **Galitz 93**

PenPoint User Interface Design Reference **GO 91**

Object-Oriented Interface Design: IBM Common User Access Guidelines **IBM 89**

NeXTSTEP User Interface Guidelines **NeXT 90**

UI: Programming Concepts

Object Oriented Application Frameworks **Lewis 96**

The Smalltalk Developer's Guide to VisualWorks **Howard 95**

Hyptertext and WWW Design

Looking Good Online **Bain+G 96**

Making Hypermedia Work: A User's Guide to HyTime. **DeRose+D 94**

Literary Machines **Nelson 81**

Multimedia and Hypermedia **Nielsen 90**

Multimedia and Hypertext: The Internet and Beyond **Nielsen 95**

Deconstructing Web Graphics **Weinman 96**

Document Modeling

Developing SGML DTDs: From Text to Model to Markup. **Maler+A 96**

The SGML Handbook. **Goldfarb 90**

LaTeX: A Document Preparation System **Lamport 86**

The TeXbook **Knuth 84**

5.2 By Title

Beyond	
Object Database Standard: ODMG 2.0.	**Cattell+D 97**
Object Oriented Software Construction, 2nd Edition.	**Meyer 97**
Object Solutions: Managing the Object-Oriented Project	**Booch 96**
Object-Oriented Analysis and Design with Applications.	**Booch 94**
Object-Oriented Modeling and Design.	**Rumbaugh+BPEL 91**
Object-Oriented Programming: The CLOS Perspective.	**Paepcke 93**
Object-Oriented Software Engineering: A Use Case Driven Approach.	**Jacobson+CJO 92**
Pattern Languages of Program Design	**Coplien+S 95**
Pattern Languages of Program Design 2	**Vlissides+CK 96**
Pattern Languages of Program Design 3	**Martin+RB 98**
Peopleware: Productive Projects and Teams	**DeMarco+L 87**
Pitfalls of Object-Oriented Development	**Webster 95**
Rapid Development	**McConnell 96**
Relational Database Writings, 1985-1989	**Date 90**
Relational Database Writings, 1989-1991	**Date 92**
Relational Database Writings, 1991-1994	**Date 95b**
Relational Database: Selected Writing	**Date 86**
Relational Model for Database Management, Version 2.	**Codd 90**
SGML Handbook.	**Goldfarb 90**
Smalltalk Best Practice Patterns, Volume 1: Coding	**Beck 96**
Smalltalk-80: The Language and its Implementation.	**Goldberg+R 83**
Software Architecture	**Shaw+G 96**
Software Architecture and Design: Principles, Models, and Method.	**Witt+BM 94**
Software Architecture Bibliography	**SEI**
Software Project Survival Guide	**McConnell 98**
Software Reuse: Architecture, Process and Organization for Business Success	**Jacobson+GJ 97**
Structure and Interpretation of Computer Programs.	**Abelson+S 96**
System Architecting: Creating and Building Complex Systems	**Rechtin 91**
TeXbook	**Knuth 84**
The Object Advantage: Business Process Reengineering with Object Technology	**Jacobson+EJ 95**
UML Distilled: Applying the Standard Object Modeling Language	**Fowler 97**
UML: Unified Modeling Language, Version 1.1	**Rational 98**
Understanding UML: The Developer's Guide	**Harmon+W 98**
Using CRC Cards: An Informal Approach to Object-Oriented Development	**Wilkinson 95**

5.3 References

Abelson+S 96 Harold Abelson and Gerald Jay Sussman, with Julie Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, Cambridge, MA, 1996.

ACM-CHI 91 Association for Computing Machinery, Inc. *Proceeding of CHI, 1991.* Addison-Wesley, Reading, MA, 1991. ISBN: 0-201-51278-5.

Andleigh+G 92 Prabhat Andleigh and Michael Gretzinger. *Distributed Object-Oriented Data-Systems Design.* Prentice Hall, Englewood Cliffs, NJ, 1992.

Apple 92 Apple Computer, Inc. *Macintosh Human Interface Guidelines.* Addison-Wesley, Reading, MA, 1992. ISBN: 0-201-62216-5.

Baecker+B 87 Ronald M. Baecker and William A.S. Buxton. *Readings in Human-Computer Interaction: A Multidisciplinary Approach.* Morgan Kauffman, Los Altos, California, 1987.

Baecker+GBG 95 Ronald M. Baecker, Jonathan Grudin, William A.S. Buxton, Saul Greenberg. *Reading in Human-Computer Interaction: Toward the Year 2000.* Morgan Kauffman, Los Altos, California, 1995.

Bain+G 96 Steve Bain with Daniel Gray. *Looking Good Online.* Ventana, Research Triangle Park, NC, 1996. ISBN: 1-56604-469-3.

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

Bellin+S 97 David Bellin and Susan Suchman Simone. *The CRC Card Book.* Addison-Wesley, Reading, MA, 1997.

Ben-Natan 95 Ron Ben-Natan. *CORBA: A Guide to the Common Object Request Broker Architecture.* McGraw-Hill, New York, NY, 1995.

Bergin+G 96 Thomas J. Bergin, Jr. and Richard G. Gibson, Jr., editors. *History of Programming Languages – II.* Addison-Wesley, Reading, MA, 1996.

Berson 92 Alex Berson. *Client/Server Architecture.* McGraw Hill, New York, NY, 1992.

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

Booch 96 Grady Booch. *Object Solutions: Managing the Object-Oriented Project*. Addison-Wesley, Menlo Park, CA, 1996.

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

Buschmann+MRSS 96 Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *A System of Patterns: Pattern-Oriented Software Architecture.* Wiley, Chichester, England, 1996

Castano+FMS 95 Silvana Castano, Mariagrazia Fugini, Giancarlo Martella, and Pierangela Samarati. *Database Security.* Addison-Wesley, Wokingham, England, 1995.

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

Cattell+B 97 R.G.G. Cattell and Douglas K. Barry, Editors. *The Object Database Standard: ODMG 2.0.* Morgan Kaufmann, San Francisco, CA 1997.

Cheswick+B 94 William Cheswick and Steven Bellovin. *Firewalls and Internet Security.* Addison-Wesley, Reading, MA, 1994.

ChiMu 97a ChiMu Corporation. *Learning FORM*.

ChiMu 97e ChiMu Corporation. *ChiMu OO/Java Development: Guidelines and Resources*.

Coad+M 96 Peter Coad and Mark Mayfield. *Java Design: Building Better Apps & Applets.* Yourdon Press, Upper Saddle River, NJ, 1996.

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

Constantine 95 Larry Constantine. *Constantine on Peopleware.* Prentice Hall, Englewood Cliffs, NJ, 1995.

Cook 92 William R. Cook. "Interfaces and Specifications for the Smalltalk-80 Collection Classes" *OOPSLA 92 Proceedings* Association for Computer Machinery, New York, NY, 1992

Cook+D 94 Steve Cook and John Daniels. *Designing Object Systems: Object-Oriented Modeling with Syntropy*. Prentice Hall, New York, NY, 1994.

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.

Cotter+P 95 Sean Cotter with Mike Potel. *Inside Taligent Technology*. Addison-Wesley, Reading, MA, 1995. ISBN: 0-201-40970-4.

Coulouris+DK 94 George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems: Concepts and Design.* Addison-Wesley, Reading, MA, 1994.

Cox+W 93 Kevin Cox and David Walker. *User Interface Design.* Prentice Hall, New York, NY, 1993. ISBN: 0-13-952888-1.

Date 86 C.J. Date. *Relational Database: Selected Writing*. Addison-Wesley, Reading, MA, 1986.

Date 90 C.J. Date. *Relational Database Writings, 1985-1989.* Addison-Wesley, Reading, MA, 1990.

Date 92 C.J. Date. *Relational Database Writings, 1989-1991*. Addison-Wesley, Reading, MA, 1992.

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.

Date+D 97 C.J. Date with Hugh Darwin. *A Guide to the SQL Standard – Fourth Edition.* Addison-Wesley, Reading, MA, 1997.

Davis 95 Alan Davis. *201 Principles of Software Development.* McGraw Hill, New York, NY, 1995.

DeMarco 82 Tom DeMarco. *Controlling Software Projects.* Prentice Hall, Englewood Cliffs, NJ, 1982.

DeMarco 95 Tom DeMarco. *Why Does Software Cost So Much?* Dorset House, New York, NY, 1995.

DeMarco+L 87 Tom DeMarco and Tim Lister. *Peopleware: Productive Projects and Teams.* Dorset House, New York, NY, 1987.

DeRose+D 94 Steven J. Derose and David G. Durand. *Making Hypermedia Work: A User's Guide to HyTime.* Kluwer, Boston, MA, 1994.

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

Fowler 97 Martin Fowler. *Analysis Patterns: Reusable Object Models.* Addison-Wesley, Menlo Park, CA, 1997.

Fowler 97 Martin Fowler. *UML Distilled: Applying the Standard Object Modeling Language.* Addison-Wesley, Reading, MA, 1997.

Fussell 96 Mark L. Fussell. "A Good Architecture for Object-Oriented Information Systems". http://www.chimu.com/publications/oopsla96tutorial23/

Fussell 97a Mark L. Fussell. *SmallJava: Using Language Transformation to Show Language Differences.* http://www.chimu.com/publications/smallJava/

Fussell 97b Mark L. Fussell. *Java Development Standards.* http://www.chimu.com/publications/javaStandards/

Galitz 93 Wilber O. Galitz. *User-Interface Screen Design.* QED, Wellesley, MA, 1993. ISBN: 0-89435-406-X.

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

GemStone 95 GemStone Systems, Incorporated. *GemStone Programmers Guide*. GemStone, 1995.

GO 91 GO Corporation. *PenPoint User Interface Design Reference.* Addison-Wesley, Reading, MA, 1991. ISBN: 0-201-60858-8.

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

Goldfarb 90 Charles F. Goldfarb. *The SGML Handbook.* Oxford University Press, New York, NY, 1990.

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

Harmon+W 98 Paul Harmon and Mark Watson. *Understanding UML: The Developer's Guide.* Morgan Kaufmann, San Francisco, CA, 1998.

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

Humphrey 89 Watts Humphrey. *Managing the Software Process.* Addison-Wesley, Reading, MA, 1989.

Humphrey 97 Watts Humphrey. *Managing Technical People.* Addison-Wesley, Reading, MA, 1997.

IBM 89 IBM Corporation. *Object-Oriented Interface Design: IBM Common User*

Access Guidelines. Que, Carmel, IN, 1989. ISBN: 1-56529-170-0.

Jacobson+CJO 92 Ivar Jacobson with Magnus Christerson, Patrick Johnsson, and Gunnar Övergaard. *Object-Oriented Software Engineering: A Use Case Driven Approach.* Addison-Wesley, Wokingham, England, 1992.

Jacobson+EJ 95 Ivar Jacobson, Maria Ericsson, and Agneta Jacobson. *The Object Advantage: Business Process Reengineering with Object Technology.* Addison-Wesley, Wokingham, England, 1995.

Jacobson+GJ 97 Ivar Jacobson, Martin Gris, and Patrik Jonsson. *Software Reuse: Architecture, Process and Organization for Business Success.* Addison-Wesley, Harlow, England, 1997.

Kay 96 Alan Kay. "The Early History of Smalltalk" in [Bergin+G 96].

Kiczales+RB 91 Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, Cambridge, MA, 1991.

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

Kim 95 Won Kim, editor. *Modern Database Systems: The Object Model, Interoperability, and Beyond.* Addison-Wesley, Reading, MA, 1995.

Knuth 84 Donald E. Knuth. *The TeXbook.* Addison-Wesley, Reading, MA, 1984.

Lamport 86 Leslie Lamport. *LaTeX: A Document Preparation System.* Addison-Wesley, Reading, MA, 1986.

Larman 97 Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design.* Prentice Hall, Englewood Cliffs, NJ, 1997.

Larman 98 Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design.* Prentice Hall, Upper Saddle River, NJ, 1998.

Laurel 91 Brenda Laurel. *Computers as Theatre.* Addison-Wesley, Reading, MA, 1991. ISBN: 0-201-51048-0.

Lea-1 Doug Lea. "Java Coding standards". http://gee.cs.oswego.edu/dl/html/javaCodingStd.html

Lewis 96 Ted Lewis and Glenn Andert, Paul Calder, Erich Gamma, Wolgang Pree, Larry Rosentstein, Kurt Schmucker, André Weinang, and John Vlissides. *Object Oriented Application Frameworks.* Manning, Greenwich, England, 1996.

Maguire 94 Steve Maguire. *Debugging the Development Process.* Microsoft Press, Redmond, WA, 1994.

Maler+A 96 Eve Maler and Jeanne El Andaloussi. *Developing SGML DTDs: From Text to Model to Markup.* Prentice Hall, Upper Saddle River, NJ, 1996.

Mandel 97 Theo Mandel. *The Elements of User Interface Design.* Wiley, New York, NY, 1997. ISBN: 0-471-16267-1.

Martin+RB 98 Robert Martin, Dirk Riehle, and Frank Buschmann eds. *Pattern Languages of Program Design 3.* Addison-Wesley, Reading, MA, 1998.

McCarthy 95 Jim McCarthy. *Dynamics of Software Development.* Microsoft Press, Redmond, WA, 1995.

McConnell 93 Scott McConnell. *Code Complete: A Practical Handbook of Software*

Construction. Microsoft Press, Redmond, WA, 1993.

McConnell 96 Scott McConnell. *Rapid Development*. Microsoft Press, Redmond, WA, 1996.

McConnell 98 Steve McConnell. *Software Project Survival Guide.* Microsoft Press, Redmond, WA, 1998.

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.

Mowbray+M 97 Thomas J. Mowbray and Raphael C. Malveau. *CORBA Design Patterns.* Wiley, New York, NY, 1997.

Mowbray+R 97 Thomas J. Mowbray and William A. Ruh. *Inside CORBA: Distributed Object Standards and Applications.* Addison-Wesley, Reading, MA, 1997.

Mullender 93 Sape Mullender, ed. *Distributed Systems.* Addison-Wesley, Reading, MA, 1993.

Mullet+S 95 Kevin Mullet and Darrell Sano. *Designing Visual Interfaces: Communication Oriented Techniques.* Prentice Hall, Englewood Cliffs, NJ, 1995. ISBN: 0-13-303389-9.

Nelson 81 Theodor H. Nelson. *Literary Machines.* (Self published: ISBN 0-89347-055-04)

NeXT 90 NeXT Computer, Inc. *NeXTSTEP User Interface Guidelines.* Addison-Wesley, Reading, MA, 1990.

Nielsen 90 Jakob Nielsen. *Multimedia and Hypermedia.* Academic Press, Boston, MA, 1990. ISBN: 0-12-518410-7.

Nielsen 93 Jakob Nielsen. *Usability Engineering.* Academic Press, San Diego, CA, 1993. ISBN: 0-12-518405-0.

Nielsen 95 Jakob Nielsen. *Multimedia and Hypertext: The Internet and Beyond.* AP Professional, Boston, MA, 1995. ISBN: 0-12-518408-5.

Norman 88 Donald A. Norman. *The Design of Everyday Things.* Doubleday, New York, NY, 1988. ISBN: 0-385-26774-6.

Norman 92 Donald A. Norman. *Turn Signals Are the Facial Expressions of Automobiles.* Addison-Wesley, Reading, MA, 1992. ISBN: 0-201-58124-8.

Norman+D 86 Donald A. Norman and Stephen W. Draper. *User Centered System Design.* Lawrence Erlbaum Associates, Hillsdale, NJ, 1986. ISBN: 0-89859-872-9.

O'Connell 94 Fergus O'Connell. *How to Run Successful Projects.* Prentice Hall, New York, NY, 1994.

Orfali+HE 94 Robert Orfali, Dan Harkey, and Jeri Edwards. *Essential Client/Server Survival Guide.* Van Nostrand Reinhold, New York, NY, 1994.

Paepcke 93 Andreas Paepcke, Editor. *Object-Oriented Programming: The CLOS Perspective.* MIT Press, Cambridge, MA, 1993.

Rational 98 Rational Corporation. "UML: Unified Modeling Language, Version 1.1". http://www.rational.com/uml/

Rechtin 91 Eberhardt Rechtin. *System Architecting: Creating and Building Complex Systems.* Prentice Hall, Englewood Cliffs, NJ, 1991

Renaud 93 Paul Renaud. *Introduction to Client/Server Systems: A Practical Guide for Systems Professionals.* Wiley, New York, NY, 1993.

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.

SEI Software Engineering Institute. "Software Architecture Bibliography". http://www.sei.cmu.edu/technology/architecture/bibliography.html

Shan+E 98 Yen-Ping Shan and Ralph H. Earle. *Enterprise Computing with Objects: From Client/Server Environments to the Internet.* Addison-Wesley, Reading, MA, 1998.

Shaw+G 96 Mary Shaw and David Garlan. *Software Architecture*. Prentice-Hall, Englewood Cliffs, NJ, 1996.

Shneiderman 98 Ben Shneiderman. *Designing the User Interface: Strategies for Effective Human-Computer Interaction.* Addison-Wesley, Reading, MA, 1998. ISBN: 0-201-69497-2.

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

Squeak "Squeak: An open, Highly-portable Smalltalk-80 Implementation" http://squeak.cs.uiuc.edu/

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

Tognazzini 92 Bruce Tognazzini. *TOG on Interface.* Addison-Wesley, Reading, MA, 1992. ISBN: 0-201-60842-1

UIUC UIUC Smalltalk/Patterns Group. "Patterns Papers and Bibliography". http://st-www.cs.uiuc.edu/users/patterns/patterns.html

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

Webster 95 Bruce Webster. *Pitfalls of Object-Oriented Development.* M&T Books, New York, NY, 1995

Weinman 96 Lynda Wienman. *Deconstructing Web Graphics.* New Riders, Indianapolis, IN, 1996. ISBN: 1-56205-641-7.

Wilkinson 95 Nancy Wilkinson. *Using CRC Cards: An Informal Approach to Object-Oriented Development.* SIGS, New York, NY, 1995.

Winston+H 81 Patrick Henry Winston and Berthold Klaus Paul Horn. *LISP*. Addison-Wesley, Reading, MA, 1981.

Wirfs-Brock+WW 90 Rebecca Wirfs-Brock, Brian Wilkerson, and Laura Wiener. *Designing Object-Oriented Software.* Prentice Hall, Englewood Cliffs, NJ, 1990.

Witt+BM 94 Bernard Witt, Terry Baker, and Everett Merrit. *Software Architecture and Design: Principles, Models, and Methods*. Van Nostrand Reinhold, New York, NY, 1994.

Woolf Bobby Woolf. "Partitioning Smalltalk Code into ENVY/Developer Components". http://c2.com/ppr/envy/. 1995.

6 Notation

Within this document and other ChiMu documents, some additions and modifications to UML notation are used. This chapter describes those modifications.

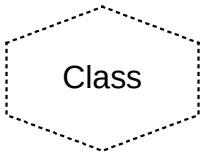
Design notation enhances communication among people. It is an important tool for both the design process itself and for communicating the finished design to clients and maintainers. Design notation needs to be standardized because it is shared among many people who must all interpret it consistently.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

ChiMu uses a design notation based on UML (see [Rational-1 http://www.rational.com/uml/]), but it has some variations that are improvements, additions, or legacies from other notations. For example, we 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 we merged into our common notation.

The following describes our notation's differences from UML. If something is not mentioned it should be assumed that the UML notation is the correct notation. Also, at no point is UML notation interpreted differently: UML notation is interpreted just as in UML. The notation below augments and can override UML but does not conflict with it. All of the items below could be described as stereotypes within UML.

6.1 Objects and Classes

The most obvious difference between UML and our notation is that we normally 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 us.



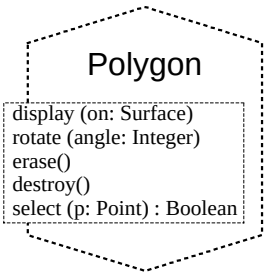
Rumbaugh did have a legitimate argument that Booch clouds were difficult to draw and even to electronically use (because they are hard to connect to). Booch and Rumbaugh came up with a very good (synergistic) solution of using hexagons in early versions of the UML, but later versions of the UML abandoned them. This was unfortunate and we decided to reverse that mistake.

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.

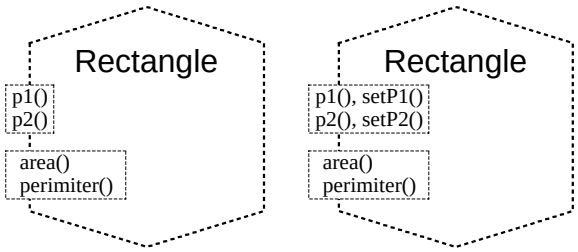
If placed within one compartment, different protocols (groups of methods) are organized like lists are in UML.

We use standard UML method annotations.



Attributes

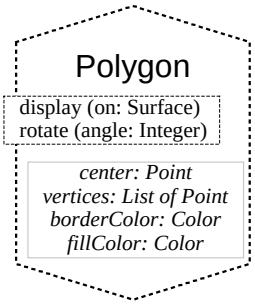
An attribute is a public property of an object that shows the state of the object. We do not make any distinction between attributes and other methods other than possibly showing them as a separate protocol from other methods. If attributes are independently changable, they will have a corresponding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

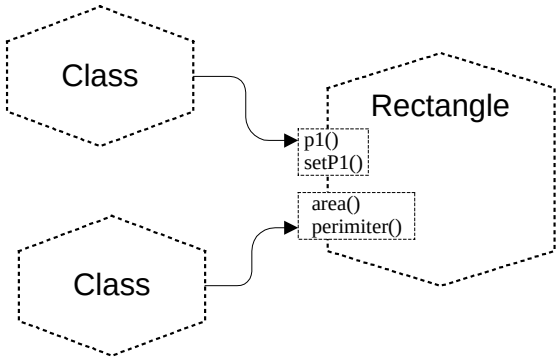
Instance Variables

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.



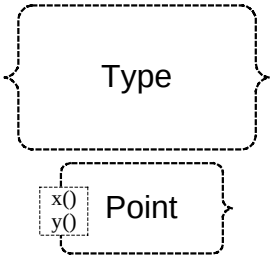
Shorthand for hand drawings

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.

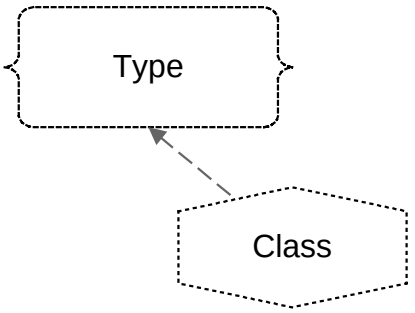
6.2 Types

We 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].

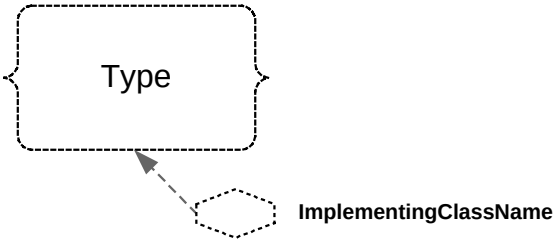
Method (and Attribute) compartments can be specified for a Type, but instance variables can not be.



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.



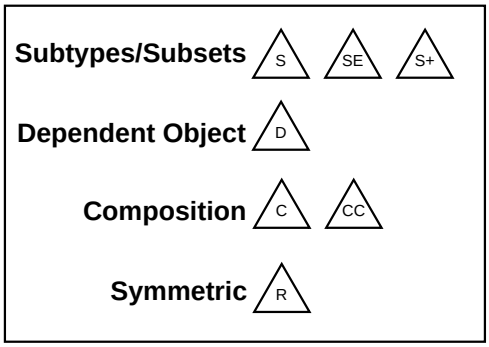
Shorthand for hand drawings

It is acceptable to draw types as straight rectangles instead of the curly rectangles. It is also acceptable to use a Class notation to indicate a Type when doing Conceptual or Design model diagrams. This is the result of an inherent property of UML that during Conceptual and Design phases all models are external specification models.

6.3 Relationships

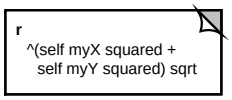
Generally we use UML notation when possible. *Information Modeling* [Kilov+R 94] has a richer set of relationships than UML and a cleaner decomposition of them, so in certain diagrams we will use them instead of the less precise UML notations.

Kilov and Ross relationship types



6.4 Code Blocks

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

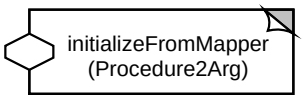


6.5 Other Notations

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

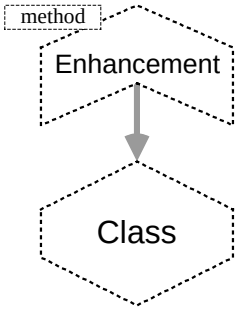
Functor

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.



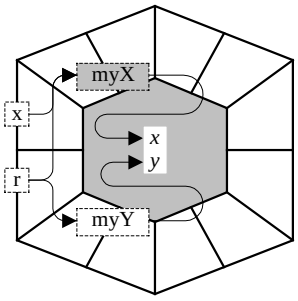
Enhancement

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.



ChiMu Corporation

1220 N. Fair Oaks Ave, #1314
Sunnyvale, CA 94089

Phone: 408 734-9068
Email: info@chimu.com
www.chimu.com