Principles of Object-Oriented Programming

Collection Editors:
Stephen Wong
Dung Nguyen

Principles of Object-Oriented Programming

Collection Editors:
Stephen Wong
Dung Nguyen

Authors:
Dung Nguyen
Alex Tribble
Stephen Wong

Online:
< http://legacy.cnx.org/content/col10213/1.37/ >

OpenStax-CNX

This selection and arrangement of content as a collection is copyrighted by Stephen Wong, Dung Nguyen. It is licensed under the Creative Commons Attribution License 3.0 (http://creativecommons.org/licenses/by/3.0/).
Collection structure revised: May 10, 2013
PDF generated: January 14, 2015
For copyright and attribution information for the modules contained in this collection, see p. 182.

Table of Contents

Chapter 1

Introduction

1.1 Abstraction[1]

1.1.1 Introduction

Abstraction is the process of hiding the details and exposing only the essential features of a particular concept or object. Computer scientists use abstraction to understand and solve problems and communicate their solutions with the computer in some particular computer language. We illustrate this process by way of trying to solve the following problem using a computer language called Java.

Problem: Given a rectangle 4.5 ft wide and 7.2 ft high, compute its area.

We know the area of a rectangle is its width times its height. So all we have to do to solve the above problem is to multiply 4.5 by 7.2 and get the the answer. The question is how to express the above solution in Java, so that the computer can perform the computation.

1.1.2 Data Abstraction

The product of 4.5 by 7.2 is expressed in Java as: `4.5 * 7.2`. In this expression, the symbol `*` represents the multiplication operation. 4.5 and 7.2 are called number **literals**. Using DrJava, we can type in the expresssion `4.5 * 7.2` directly in the interactions window and see the answer.

Now suppose we change the problem to compute the area of a rectangle of width 3.6 and height 9.3. Has the original problem really change at all? To put it in another way, has the **essence** of the original problem changed? After all, the formula for computing the answer is still the same. All we have to do is to enter 3.6 * 9.3. What is it that has **not** change (the **invariant**)? And what is it that has changed (the **variant**)?

1.1.2.1 Type Abstraction

The problem has not changed in that it still deals with the same geometric shape, a rectangle, described in terms of the same dimensions, its width and height. What vary are simply the values of the width and the height. The formula to compute the area of a rectangle given its width and height does not change:

```
width * height
```

It does not care what the actual specific values of width and height are. What it cares about is that the values of width and height must be such that the multiplication operation makes sense. How do we express the above invariants in Java?

We just want to think of the width and height of a given rectangle as elements of the set of real numbers. In computing, we group values with common characteristics into a set and called it a **type**. In Java, the

[1] This content is available online at <http://legacy.cnx.org/content/m11785/1.21/>.

type **double** is the set of real numbers that are implemented inside the computer in some specific way. The details of this internal representation is immaterial for our purpose and thus can be ignored. In addition to the type double, Java provides many more pre-built types such as **int** to represent the set of integers and **char** to represent the set of characters. We will examine and use them as their need arises in future examples. As to our problem, we only need to restrict ourselves to the type double.

We can define the width and the height of a rectangle as double in Java as follows.

```
double width;
double height;
```

The above two statements are called **variable** definitions where width and height are said to be variable names. In Java, a variable represents a memory location inside the computer. We define a variable by first declare its type, then follow the type by the name of the variable, and terminate the definition with a semi-colon. This a Java syntax rule. Violating a syntax rule constitutes an error. When we define a variable in this manner, its associated memory content is initialized to a default value specified by the Java language. For variables of type double, the default value is 0.

> NOTE: Use the interactions paneof DrJava to evaluate width and height and verify that their values are set to 0.

Once we have defined the width and height variables, we can solve our problem by writing the expression that computes the area of the associated rectangle in terms of width and height as follows.

```
width * height
```

Observe that the two variable definitions together with the expression to compute the area presented in the above directly translate the description of the problem -two real numbers representing the width and the height of a rectangle- and the high-level thinking of what the solution of the problem should be -area is the width times the height. We have just expressed the invariants of the problem and its solution. Now, how do we vary width and height in Java? We use what is called the **assignment** operation. To assign the value 4.5 to the variable width and the value 7.2 to the variable height, we write the following Java assignment statements.

```
width = 4.5;
height = 7.2;
```

The syntax rule for the assignment statement in Java is: first write the name of the variable, then follow it by the equal sign, then follow the equal sign by a Java expression, and terminate it with a semi-colon. The semantic (i.e. meaning) of such an assignment is: evaluate the expression on the right hand side of the equal sign and assign the resulting value into the memory location represented by the variable name on the left hand side of the equal side. It is an error if the type of the expression on the right hand side is not **a subset** of the type of the variable on the left hand side.

Now if we evaluate `width * height` again (using the Interactions Window of DrJava), we should get the desired answer. Life is good so far, though there is a little bit of inconvenience here: we have to type the expression width * height each time we are asked to compute the area of a rectangle with a given width and a given height. This may be OK for such a simple formula, but what if the formula is something much more complex, like computing the length of the diagonal of a rectangle? Re-typing the formula each time is quite an error-prone process. Is there a way to have the computer memorize the formula and perform the computation behind the scene so that we do not have to memorize it and rewrite it ourselves? The answer is yes, and it takes a little bit more work to achieve this goal in Java.

What we would like to do is to build the equivalent of a black box that takes in as inputs two real numbers (recall type **double**) with a button. When we put in two numbers and depress the button, "magically" the black box will compute the product of the two input numbers and spit out the result, which we will interpret

as the area of a rectangle whose width and height are given by the two input numbers. This black box is in essence a specialized calculator that can only compute one thing: the area of a rectangle given a width and a height. To build this box in Java, we use a construct called a class, which looks like the following.

```
class AreaCalc {
    double rectangle(double width, double height) {
        return width * height;
    }
}
```

What this Java code means is something like: `AreaCalc` is a **blue print** of a specialized computing machine that is capable of accepting two input `doubles` , one labeled width and the other labeled height, computing their product and returning the result. This computation is given a name: `rectangle`. In Java parlance, it is called a **method** for the class `AreaCalc`.

Here is an example of how we use `AreaCalc` to compute area of a rectanglee of width 4.5 and height 7.2. In the Interactions pane of DrJava, enter the following lines of code.

```
AreaCalc calc = new AreaCalc();
calc.rectangle(4.5, 7.2)
```

The first line of code defines `calc` as a variable of type `AreaCalc` and assign to it an **instance** of the class `AreaCalc`. **new** is a keyword in Java. It is an example of what is called a class operator. It operates on a class and creates an instance (also called **object**) of the given class. The second line of code is a call to the object `calc` to perform the `rectangle` task where `width` is assigned the value 4.5 and `height` is assigned the value 7.2. To get the area of a 5.6 by 8.4 rectangle, we simply use the same calculator `calc` again:

```
calc.rectangle(5.6, 8.4);
```

So instead of solving just one proble -given a rectangle 4.5 ft wide and 7.2 ft high, compute its area- we havebuilt a "machine" that can compute the area of **any** given rectangle. But what about computing the area of a right triangle with height 5 and base 4? We cannot simply use this calculator. We need another specialized calculator, the kind that can compute the area of a circle.

There are at least two different designs for such a calculator.

- create a new class called AreaCalc2 with one method called **rightTriangle** with two input parametersof type **double**. This corresponds to designing a different area calculator with one button labeled **rightTriangle** with two input slots.
- add to AreaCalc a method called rightTriangle with two input parameters of type **double**. This corresponds to designing an area calculator with two buttons: one labeled **rectangle** with two input slots and the other labeled **rightTriangle**, also with two input slots.

In either design, it is the responsibility of the calculator user to pick the appropriate calculator or press the appropriate button on the calculator to correctly obtain the area of the given geometric shape. Since the two computations require exactly the same number of input parameters of exactly the same type, the calculator user must be careful not get mixed up. This may not be too much of an inconvenience if there are only two kinds of shape to choose from: rectangle and right triangle. But what if the user has to choose from hundreds of different shapes? or better yet an **open-ende** number of shapes? How can we, as programmers, buid a calculator that can handle an **infinite** number of shapes? The answer lies in **abstraction**. To motivate how conceptualize the problem, let us digress and contemplate the behavior of a child!

1.1.2.2 Modeling a Person

For the first few years of his life, Peter did not have a clue what birthdays were, let alone his own birth date. He was incapable of responding to your inquiry on his birthday. It was his parents who planned for

his elaborate birthday parties months in advance. We can think of Peter then as a rather "dumb" person with very little intelligence and capability. Now Peter is a college student. There is a piece of memory in his brain that stores his birth date: it's September 12, 1985! Peter is now a rather smart person. He can figure out how many more months till his next birthday and e-mail his wish list two months before his birth day. How do we model a "smart" person like Peter? Modeling such a person entails modeling

- a birth date and
- the computation of the number of months till the next birth day given the current month.

A birth date consists of a month, a day and a year. Each of these data can be represented by an integer, which in Java is called a number of type **int**. As in the computation of the area of a rectangle, the computation of the number of months till the next birth day given the current month can be represented as a method of some class. What we will do in this case that is different from the area calculator is we will lump both the data (i.e. the birth date) and the computation involving the birth date into one class. The grouping of data and computations on the data into one class is called encapsulation. Below is the Java code modeling an intelligent person who knows how to calculate the number of months before his/her next birth day. The line numbers shown are there for easy referencing and are not part of the code.

```
1  public class Person {
2     /**
3     * All data fields are private in order to prevent code outside of this
4     * class to access them.
5     */
6     private int _bDay;   // birth day
7     private int _bMonth; // birth month; for example, 3 means March.
8     private int _bYear;  // birth year

9      /**
10    * Constructor: a special code used to initialize the fields of the class.
11    * The only way to instantiate a Person object is to call new on the constructor.
12    * For example: new Person(28, 2, 1945) will create a Person object with
13    * birth date February 28, 1945.
14    */
15    public Person(int day, int month, int year) {
16       _bDay = day;
17       _bMonth = month;
18       _bYear = year;
19    }

20    /**
21    * Uses "modulo" arithmetic to compute the number of months till the next
22    * birth day given the current month.
23    * @param currentMonth an int representing the current month.
24    */
25    public int nMonthTillBD(int currentMonth) {
26        return (_bMonth - currentMonth + 12) % 12;
27    }
28 }
```

(Download the above code[2]) We now explain what the above Java code means.

[2] http://legacy.cnx.org/content/m11785/latest/Person.java

- line 1 defines a class called Person. The opening curly brace at the end of the line and the matching closing brace on line 28 delimit the contents of class Person. The key word **public** is called an **access specifier** and means all Java code in the system can reference this class.
- lines 2-5 are comments. Everything between /* and */ are ingored by the compiler.
- lines 6-8 define three integer variables. These variables are called **fields** of the class. The key word private is another access specifier that prevents access by code outside of the class. Only code inside of the class can access them. Each field is followed by a comment delimited by // and the end-of-line. So there two ways to comment code in Java: start with /* and end with */ or start with // and end with the end-of-line.
- lines 9-14 are comments.
- lines 15-19 constitute what is called a **constructor**. It is used to initialize the fields of the class to some particular values. The name of the constructor should spell exactly like the class name. Here it is **public**, menaing it can be called by code outside of the class Person via the operator new. For example, `new Person(28, 2, 1945)` will create an instance of a Person with _bDay = 28, _bMonth = 2 an d_bYear = 1945.
- lines 20-24are comments.
- line 23 is a special format for documenting the parameters of a metod. This format is called the javadoc format. We will learn more about javadoc in another module.
- lines 25-27 constitute the definition of a method in class Person.
- line 26 is the formula for computing the number of months before the next birthday using the remainder operator %. x % y gives the remainder of the integer division between the dividend x and the divisor y.

1.2 Objects and Classes[3]

1.2.1 Objects

In the "real" world, objects are the entities of which the world is comprised. Everything that happens in the world is related to the interactions between the objects in the world. Just as atoms, which are objects, combine to form molecules and larger objects, the interacting entities in the world can be thought of as interactions between and among both singular ("atomic") as well as compound ("composed") objects. The real world consists of many, many objects interacting in many ways. While each object may not be overly complex, their myriad of interactions creates the overall complexity of the natural world. It is this complexity that we wish to capture in our software systems.

In an object-oriented software system,**objects** are entities used to represent or model a particular piece of the system.

> *Objects are the primary units used to create abstract models.*

There are a number of schools of object-oriented programming, which differ slightly on how they view objects. Here, we will take a "behaviorist" (our term) stance:

> ***An object is characterized solely by it behaviors.***

Essentially this defines an object by the way it interacts with its world. An object that does not interact with anything else effectively does not exist. Access to internally stored data is necessarily through some sort of defined behavior of the object. It is impossible for an outside entity to truly "know" whether or not a particular piece of data is being stored inside of another object.

> SEEING VS. BEING: A beautiful example of a model that exhibits a particular behavior but without exactly replicating the mechanics we expect to produce that behavior is the "Dragon"

[3]This content is available online at <http://legacy.cnx.org/content/m11708/1.7/>.

optical illusion. A printout to create this simple folded paper display can be found at the web site of the Grand Illusions Toy Shop in England[4] .

This does not mean however, that an object may not contain data (information) in fields. The essence of the object is in how the object behaves in relationship to or as a result of its data, should it contain any.

The existence of data in an object is an implementation technique used to generate the required behavior of that object.

1.2.2 Classes

Many objects differ from each other only in the **value** of the data that they hold. For example, both a red crayon and a blue crayon are crayons; they differ only in the value of the color attribute, one has a red color and the other a blue color. Our object-oriented system needs a way to capture the abstraction of a crayon, independent of the value of its color. That is, we want to express that a set of objects are abstractly **equivalent**, differing only in the values of their attributes and perhaps, thus differing in the behaviors that result from those values.

Many objects are similar in many overall, generalized ways, differing only in smaller, more specific details. In biology and other fields, scientists organize objects into taxonomies, which are classification hierarchies used to express these similarities. For instance, a butterfly and a lobster are quite different, yet they share the common characteristics of all Arthropods, such as a jointed exoskeleton. The notion of an Arthropod enables us to understand and deal with butterflies and lobsters in an abstract, unified manner. So once again we are trying to express abstract equivalence.

Object-oriented systems use **classes** to express the above notions of abstract equivalence.

A class is an abstract description of a set of objects.

A class thus contain the descriptions of all the behaviors of the objects that it represents. In computer science parlance, we call the individual behaviors of a class its **methods**. In addition, a class may, but not always, contain descriptions of the internal data held by the objects, called its **fields**, as well as implementation details about its methods and fields.

Turning the description around, we can say that a class is a template or recipe for the creation of a particular type of object. That is, one can use a class to create ("**instantiate**") objects of the type described by the class. Be careful not to make the very beginner's common mistake of equating classes and objects. A class is a specification of an set of objects, it is not the actual object.

In technical terms, a class defines a new **type** in the system. Types are identifies used to differentiate different kinds of data. For instance, integers, characters, strings and arrays are all types of data.

1.2.2.1 Implementation in Java

Classes are the fundamtental building blocks of Java programs. Defining a class is basically a matter of specifying a name for the class, its relationship to other classes and what sort of behaviors the class exhibits as well as what data it may contain.

SuppSuppose we wanted to define a class to describe a household pet. We could do it as such:

```
 class Pet {
}
```

The word `class` above is called a **keyword** and can only be used to state that the following code is the definition of a class. The `class` keyword is immediately followed by the desired name of the class, `Pet`here. The curly braces, `{...}`, indicate the extent of the definition of the class. That is, any definitions of the class's behaviors and data will be found between the curly braces. Curly braces must therefore always appear as a matched set.

[4] http://www.grand-illusions.com/opticalillusions/dragon_illusion/

In general, in Java, curly braces mark the extent of a definition.

JAVA STYLE NOTE: The accepted typographic standard in the Java community is that the opening curly brace is at the end of a line and the ending curly brace is at the beginning of its own line. Any code between the curly braces is indented.

Well, our class for pets is simple, but not very interesting because it doesn't do anything. We could say that our pet has a name, but personally, I have found that the behavior of most pets is not affected by the particular name we give them. That is, we can give our pet a name but that doesn't change the way they behave. Let's try something else.

Pets eat a certain amount of food per day. Pets have a certain weight. Let's create a **model**, that states that the number of hours that a pet sleeps per day is related to the amount of food they eat and their weight. Thus we could say that a pet has a behavior of sleeping a certain number of hours depending on its weight and the amount of food eaten.

```
    /**
 * A class that models a household pet
 */
class Pet{

  /**
   * The weight of the pet in lbs
   */
  double weight;

  /**
   * The number of hours the pet will sleep after eating
   * the given lbs of food.
   * @param lbsOfFood The number of lbs of food eaten.
   * @return The number of hours slept after eating the food.
   */
  double hoursSlept(double lbsOfFood) {
    return 24.0*lbsOfFood/weight;
  }
}
```

But what about the pet's owner (a person, supposedly)? A person is not such a simple thing as a weight. Assigning a number to every possible person in the world is possible but not necessarily the most practical technique. There are many ways to represent people: the Registrar likes to think of you as a number with credit hours and unpaid bills, The CS department sees you as a 8-character login and unfinished graduation requirements, your doctor sees you as a collection of pulsating blood vessels, cells and bones, and your parents see you as this sweet little thing just out of diapers. A person is still a person, but the way that we choose to represent them may differ from situation to situation.

But here's the crux of the issue: does your pet care how you are internally represented? Or does your pet just want something that is a person to be their owner?

```
    /**
 * A class that models a household pet
 */
class Pet{

  /**
```

```
   * The weight of the pet in lbs
   */
  double weight;

  /**
   * The pet's owner
   */
  Person owner;

  /**
   * The number of hours the pet will sleep after eating
   * the given lbs of food.
   * @param lbsOfFood The number of lbs of food eaten.
   * @return The number of hours slept after eating the food.
   */
  double hoursSlept(double lbsOfFood) {
    return 24.0*lbsOfFood/weight;
  }

  /**
   * Determines whether or not this pet is happy to see a
   * particular person.
   * @param p The person who the pet sees.
   * @return true if the Person is their owner, false otherwise.
   */
  boolean isHappyToSee(Person p) {
    return p == owner;
  }
}
```

Here we've added a field of type `Person`, which is a class used to represent people. It doesn't matter how `Person` is implemented, just that it is a representation of a person. We could use the class definition of `Person` that was created in the module on Abstraction (Section 1.1), or perhaps one that is fixed up a little bit, or perhaps a completely new one. The `Pet` doesn't care.

Thus we see that objects can contain objects. What does this say about the possible complexity of a system of objects?

1.2.3 Download code

To download a zipped file containing the code above, click one of the following links:

- DrJava Elementary Language Level code: DrJava_Code.zip[5]
- Standard Java code: Std_Java_Code.zip[6]

Both of the above codebases include a DrJava project file (.pjt) that can be opened from DrJava to easily manage all the code and test files.

[5] See the file at <http://legacy.cnx.org/content/m11708/latest/DrJava_Code.zip>
[6] See the file at <http://legacy.cnx.org/content/m11708/latest/Std_Java_Code.zip>

1.3 Object Relationships[7]

An object-oriented system can be characterized as a system of cooperating objects. Some objects interact only with certain other objects or perhaps only with a certain set of objects. Sometimes objects are treated as equivalent even though there may be specific differences between them, for instance a situation may call for a "fruit" whereupon an "apple" will do just as well as an "orange". That is, apples and oranges are treated as abstractly equivalent. Conversely, the system designer may want to express the commonality between apples and oranges. An OO system has two distinct mechanisms to express these relationship notions: "is-a" which is technically referred to as "inheritance" and "has-a" which is technically referred to as "composition".

1.3.1 "Is-a" or "Inheritance"

"Is-a" or "inheritance" (sometimes also called "generalization") relationships capture a hierarchal relationship between classes of objects. For instance, a "fruit" is a generalization of "apple", "orange", "mango" and many others. We say that fruit is an abstraction of apple, orange, etc. Conversely, we can say that since apples are fruit (i.e. an apple "is-a" fruit), that they **inherit** all the properties common to all fruit, such as being a fleshy container for the seed of a plant.

UML Class Diagram Showing Inheritance

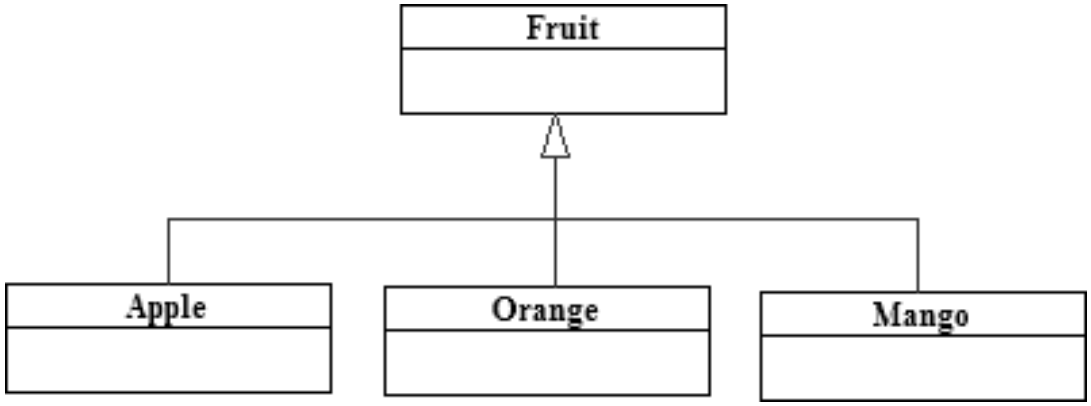


Figure 1.1: The above diagram shows the "is-a" relationship between Apple, Orange and Mango subclasses and the more abstract Fruit superclass.

NOTE: Classes are represented by boxes with the class name separated at the top by a horizontal line.

NOTE: **Inheritance ("is-a") lines** are represented by solid lines with solid arrowheads. The arrow points from the subclass to the superclass (think "a subclass object is-a superclass object")

In Java, inheritance relationships are declared using the `extends` keyword when declaring a class. A **subclass** "extends" a **superclass**, which means that the subclass is a concrete example of the more abstract superclass. For instance, the class `Apple` would extend the class `Fruit`.

[7] This content is available online at <http://legacy.cnx.org/content/m11709/1.5/>.

```
public class Apple extends Fruit {
...
}
```

In Java, a subclass is allowed to extend only a single superclass (no "multiple inheritance"). This restricts the interpretation of a hierarchal taxomony. A subclass is an embodiment of its superclass. It is useful to think of the subclass as not inheriting its superclass's behaviors but rather possessing these behaviors simply because it **is** the superclass (this is **polymorphism**). `Extend` really models what an object intrinsically is– its true "being" as it were. This is particularly useful when the superclass has particular concrete behaviors that all the subclasses should exhibit.

However, "is-a" can really be more of an "acts-like-a" relationship. This stems from the perspective that all objects are defined soley by their behaviors. We can never truly know what an object truly is, only how it **acts**. If two objects behave identically (at some abstract level) then we say that they are abstractly equivalent. What we need is a way to express the pure behavioral aspects of an object. Java has a the keyword **implements** which is used to show generalization of a pure behavioral abstraction called an **interface**. An interface has a similar syntax to a class, but only specifies behaviors in terms of the "signatures" (the input and output types) of the methods. For example we could have

```
public interface ISteerable {
public abstract void turnLeft();
public abstract void turnRight();
}

public class Truck implements ISteerable {
public void turnLeft() {
// turn the tires to the left
}
public void turnRight() {
// turn the tires to the right
}
}

public class Boat implements ISteerable {
public void turnLeft() {
// turn the rudder to the left
}
public void turnRight() {
// turn the rudder to the right
}
}
```

> NOTE: A `public` class, method or field can be seen and used by anyone. Contrasts with private (seen and used only by that class) and package (seen and used only by classes in the same package). We'll talk more about these later. An `abstract` class or method is a purely abstract definition in that it specifies the existence of a behavior without specifying exactly what that behavior is. A `void` return type declares a non-existent return value, that is, no return value at all.

Above, Trucks and Boats are not taxonomically related, but yet they both embody the behaviors of steerability, such as the ability to turn left or right. A person can pilot either a boat or a truck based soley on the fact that they both support turning left or right and not based on what sort of entity they are fundamentally. Note that as per the above definition, a class can implement multiple interfaces at the same time.

UML Class Diagram Showing Implementation

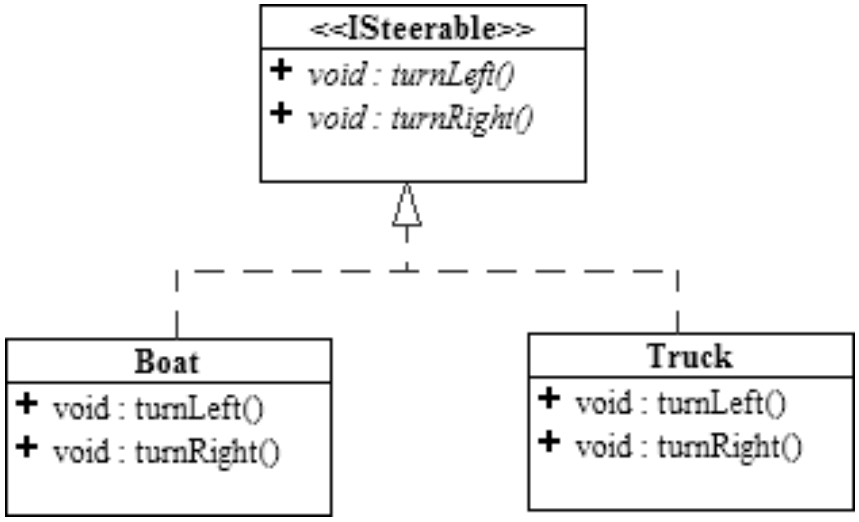


Figure 1.2: The above diagram shows the "acts-like-a" relationships between ISteerable, Boat, and Truck classes

NOTE: **Implementation ("acts-like-a") lines** are represented by **dotted** lines with solid arrowheads. The arrow points from the subclass to the interface (think "a subclass object acts-like-a interface")

1.3.2 "Has-a" or "Composition"

"Has-a" or "composition" (sometimes referred to as an "associative") relationships capture the notion that one object has a distinct and persistant communication relationship with another object. for instance, we can say a car "has-a" motor. The car and the motor are not related in a hierarchal manner, but instead we need to be able to express that this pair of objects has a particular working relationship. The car gives gas to the motor and the the motor will propel the car along. Compositional relationships can be one-way, where one object can, in a persistant manner, talk to (i.e. call methods of) a second object but the second object cannot, in a persistent manner, talk back to the first object. Compositional relationships can also be two-way, where both objects can talk to each other in a persistent manner.

UML Class Diagram Showing Composition

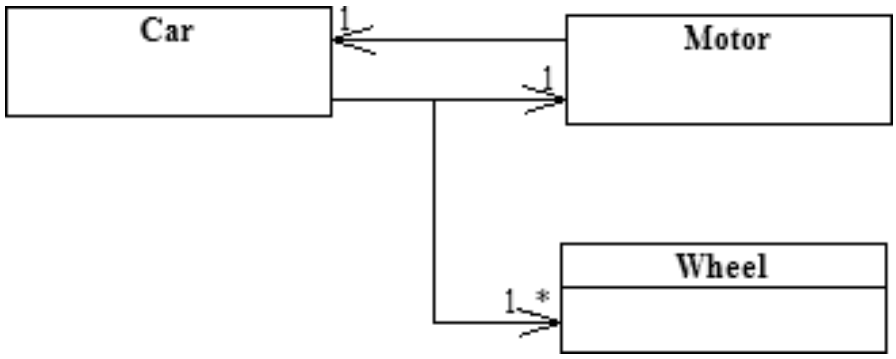


Figure 1.3: The above diagram shows the "has-a" relationships between the Car, Motor and Wheel classes

NOTE: **Composition ("has-a") lines** are represented by solid lines with **open** arrowheads. The arrow points from the owner ("composite") to the ownee ("composee"). Think "a composite has a composee". The number at the arrowhead tells how many composees there are, e.g. 1, 2, etc. "*" means an limited number, so "0...* means zero or more and "1..*" means at least one.

The emphasis made above with regards to persistent communication is deliberate and important. It is indeed possible for an object to communicate with another object in a non-persistent manner. Such non-persistent communication is generally not thought of as a compositional relationship, but rather as a dependency relationship where the action of one object depends on that of another. An object can tell a second object that it (the second object) needs to talk to a specific, perhaps third object. The second object does not know with whom it will be communicating until the first object tells it. The second object may not "remember" the object it is supposed to communicate with for any length of time after the communication was accomplished and the second object merely waits for the first object to tell it with whom to communicate next. This non-persistent communication is normally established by simply passing the third target object as an input parameter to the method call made on the second object by the first. Note that the third object could actually be the first object, creating a non-persistent two-way communication from an initially one-way communication.

1.4 UML Diagrams[8]

Unified Modeling Language ("UML") is the industry standard "language" for describing, visualizing, and documenting object-oriented (OO) systems. UML is a collection of a variety of diagrams for differing purposes. Each type of diagram models a particular aspect of OO design in an easy to understand, visual manner. The UML standard specifies exactly how the diagrams are to be drawn and what each component in the diagram means. UML is not dependent on any particular programming language, instead it focuses one the fundamental concepts and ideas that model a system. Using UML enables anyone familiar with its specifications to instantly read and understand diagrams drawn by other people. There are UML diagram for modeling static class relationships, dynamic temporal interactions between objects, the usages of objects, the particulars of an implementation, and the state transitions of systems

In general, a UML diagram consists of the following features:

[8]This content is available online at <http://legacy.cnx.org/content/m11658/1.3/>.

- **Entities**: These may be classes, objects, users or systems behaviors.
- **Relationship Lines** that model the relationships between entities in the system.
  - **Generalization** – a solid line with an arrow that points to a higher abstraction of the present item.
  - **Association** – a solid line that represents that one entity uses another entity as part of its behavior.
  - **Dependency** – a dotted line with an arrowhead that shows one entity depends on the behavior of another entity.

1.4.1 Class Diagrams

UML class diagrams model static class relationships that represent the fundamental architecture of the system. Note that these diagrams describe the relationships between **classes**, not those between specific **objects** instantiated from those classes. Thus the diagram applies to **all the objects** in the system.

A class diagram consists of the following features:

- **Classes**: These titled boxes represent the classes in the system and contain information about the name of the class, fields, methods and access specifiers. Abstract roles of the class in the system can also be indicated.
- **Interfaces**: These titled boxes represent interfaces in the system and contain information about the name of the interface and its methods.
- **Relationship Lines** that model the relationships between classes and interfaces in the system.
  - **Generalization**
    - **Inheritance**: a solid line with a solid arrowhead that points from a sub-class to a superclass or from a sub-interface to its super-interface.
    - **Implementation**: a dotted line with a solid arrowhead that points from a class to the interface that it implement
  - **Association** – a solid line with an open arrowhead that represents a "has a" relationship. The arrow points from the containing to the contained class. Associations can be one of the following two types or not specified.
    - **Composition**: Represented by an association line with a solid diamond at the tail end. A composition models the notion of one object "owning" another and thus being responsible for the creation and destruction of another object.
    - **Aggregation**: Represented by an association line with a hollow diamond at the tail end. An aggregation models the notion that one object uses another object without "owning" it and thus is **not** responsible for its creation or destruction.
  - **Dependency** – a dotted line with an open arrowhead that shows one entity depends on the behavior of another entity. Typical usages are to represent that one class instantiates another or that it uses the other as an input parameter.
- **Notes** that are used to provide further details or explanations of particular parts of the diagram. Notes are boxes with a little "dog-ear" on one corner.

Here is an example of a UML class diagram that holds most of the more common features:

UML Class Diagram

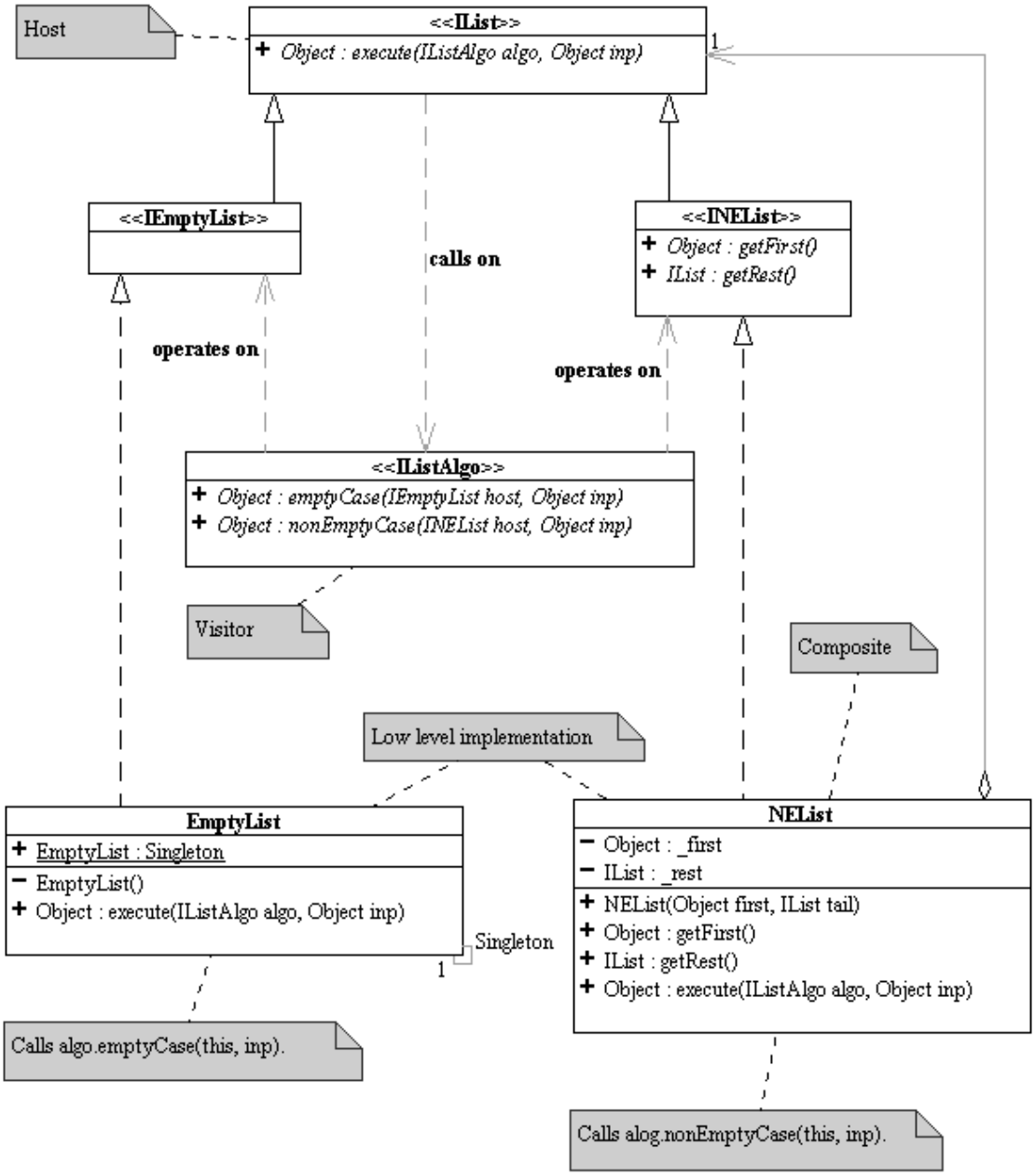


Figure 1.4: The above diagram contains classes, interfaces, inheritance and implementation lines, aggregation lines, dependency lines and notes.

Chapter 2

Polymorphism in Action

2.1 Union Design Pattern: Inheritance and Polymorphism[1]

Inheritance and polymorphism (discussed below) are two sides of the same coin and represent very foundational concepts in object-oriented programming. The union design pattern is an expression of these relationships and enables us to talk about them in a more tangible manner.

2.1.1 Union Design Pattern

Consider the following "is-a" or inheritance relationships between "concrete" entities Coca Cola, sulfuric acid, milk and the "abstract" liquid (named `"ALiquid"` by convention since it is an abstract entity):

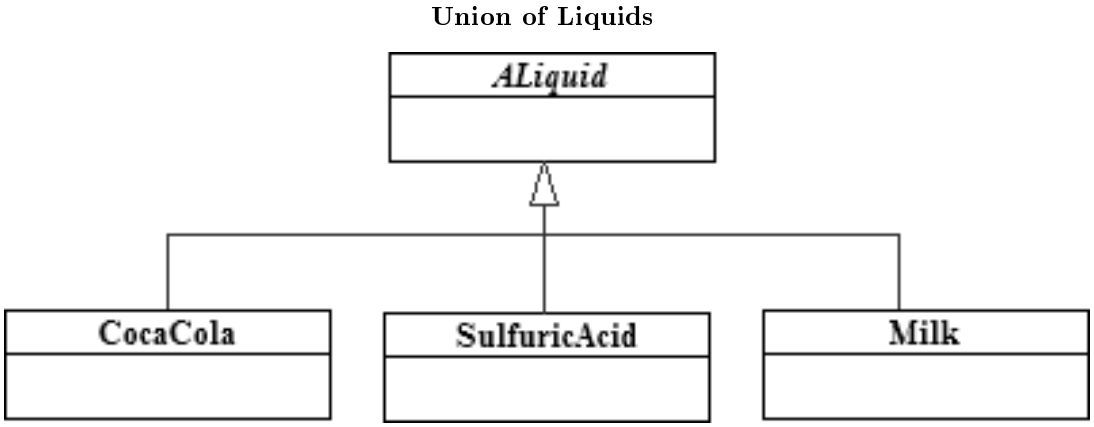


Figure 2.1: `ALiquid` is the union of `CocaCola`, `SulfuricAcid` and `Milk`

The UML diagram shows us that Coca Cola, sulfuric acid and milk are all liquids. Conversely, the diagram tells us that a liquid could be either Coca Cola, sulfuric acid or milk. Note of course, that liquids are not constrained to these 3 entities but that doesn't affect the discussion here–in fact, this will be an important feature later on.

Another way that we can express the notions depicted by the diagram is to say that the abstract `ALiquid` superclass represents the union of Coca Cola, sulfuric acid and milk. That is,

[1] This content is available online at <http://legacy.cnx.org/content/m11796/1.11/>.

a superclass represents the union of allof its subclasses.

or in other words

a superclass represents all that is abstractly equivalent about its subclasses.

For instance, the notion of an abstract liquid embodies all the behaviors and attributes such as having no definite shape, randomized atomic positions, freezing and boiling points that are common to Coca Cola, sulphuric acid and milk. Note the fine distinction between having a value and having the same value.

NOTE: In general, an interface can be substituted for the abstract superclass discussed here with no loss of generality.

The above diagram illustrating the relationship betwen a superclass and its subclasses is called the **Union Design Pattern**. The union pattern shows us **inheritance** in how the Coca Cola, sulfuric acid and milk will all inherit the abstract behaviors of liquids, such as the lack of a definite shape and freezing/boiling points. Conversely, it also shows that if a situation utilizes a liquid, either Coca Cola, milk or sulphuric acid can be used as they are all abstractly equivalent as liquids. Note that this does not imply that all three will act identically! For instance, the human throat can swallow any liquid because it is made to work with fluids that can flow. However, the reaction of the throat to sulphuric acid is markedly different than it reaction to milk! This ability to substitute any subclass for its superclass and get different behavior is called **polymorphism**.

2.1.1.1 Abstraction vs. Commonality

A subtle but extremely important point is that

Commonality does not imply abstract equivalence.

Just because a feature is common to every item in a set, does not necessarily mean that it represents some sort of abstract feature of those elements. For instance, cats, dogs, humans, and rats are all mammals where a mammal is defined as an animal that produces milk to feed its young. One could thus make a class model where a superclass `Mammal` has subclasses `Cat`, `Dog`, `Human` and `Rat`. One common feature is behavior is that cats, dogs, humans and rats all give live birth of their young. So it is tempting to say that the `Mammal` class should also embody the "live birth" behavior. However, as one wanders the world discovering new mammals, in the backwaters of Australia one finds the duck-billed platypus[2] which produces milk and is therefore clearly a mammal. However, the duck-billed platypus also lays eggs. Thus the "live birth" behavior does not belong in the `Mammal` superclass as it was only a coincidence that it was common to our initial set of subclasses. More importantly, being able to give live birth was never part of the abstract definition of a mammal and thus should never have been included in the `Mammal` superclass in the first place.

Cats, monkeys and whales, while diverse creatures, are all mammals. Hence to model such a system in the computer, it makes sense to make `Cat`, `Monkey` and `Whale` all subclasses of an abstract `Mammal` superclass. Each species has many behaviors (methods) but I will only concentrate on 3 in particular:

1. `boolean givesMilk()` : returns true if the animal can give milk to feed its young, false otherwise
2. `String makeSound()` : returns a String represenation of a common sound the animal makes.
3. `boolean givesLiveBirth()`: returns true if the animal bears live young.

In the table below are the methods and what happens when each species executes that method:

[2] http://www.creationscience.com/onlinebook/LifeSciences13.html

Mammal	Method		
	`boolean givesMilk()`	`String makeSound()`	`boolean givesLiveBirth()`
Cat	true	"Meow"	true
Monkey	true	"Screech"	true
Whale	true	"[whale song]"	true

Table 2.1

We could start out with the following class implemenation (Mammal0.java[3]):

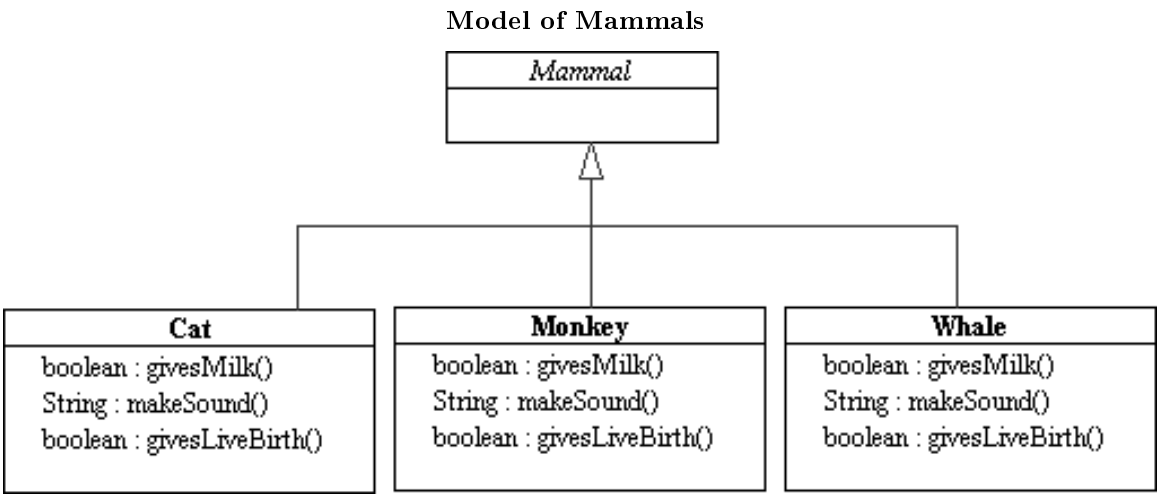


Figure 2.2: No common methods defined in the superclass.

NOTE: Italics signify abstract methods or classes

NOTE: `return_value : method_name(parameter_type_#1 parameter_name_#1, parameter_type_#2 parameter_name_#2, etc)`

Let's start our analysis:

- A mammal is defined by the fact that it gives milk to feed its young. It is thus not surprising that all the `givesMilk()` methods in the subclasses return true. The `givesMilk()` method is a prime candidate for "hoisting" up into the `Mammal` superclass ("hoisting" = moving the method upwards from the subclass to the superclass).
- `makeSound()` returns a different value for each species, but intrisically, we expect any animal, which includes mammals, to be able to make some sort of sound. Thus `Mammals` should have a `makeSound()` method, but since, at the `Mammals` level, we don't know exactly how that sound will be made, the method at that level must be abstract. The `makeSound()` method at the concrete `Cat`, `Monkey` and `Whale` level however, would be concrete because each animal makes its own unique sound.

[3] http://legacy.cnx.org/content/m11796/latest/Mammal0.java

- `givesLiveBirth()` returns exactly the same value for all of our rather diverse selection of animals here. It seems like a fine candidate for hoisting. Or is it....? Let's go ahead an hoist it anyway.

This is what we have so far (Mammal1.java[4]):

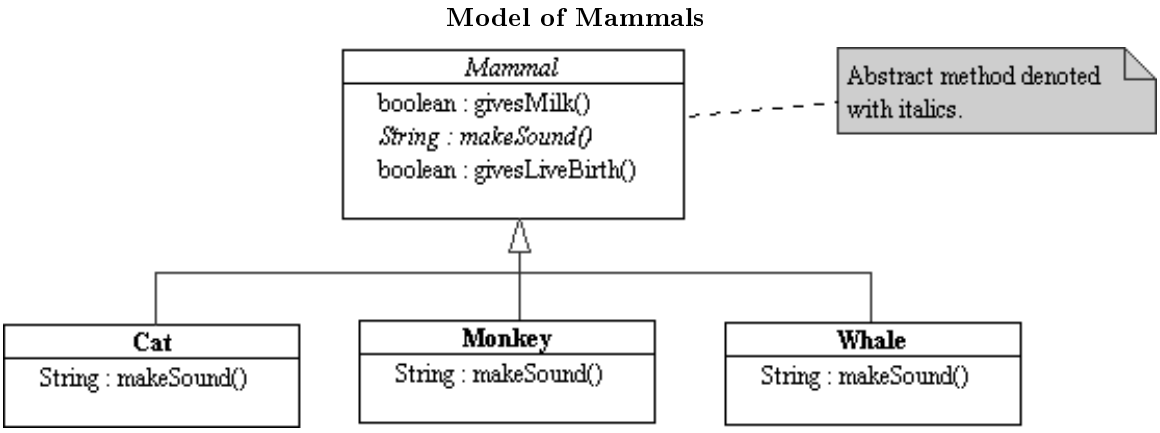


Figure 2.3: Abstract and common methods hoisted to the superclass.

Before we go charging ahead, let's stop for a moment and review what we've done: Cats, monkeys, and whales do represent a wide spectrum of mammals, but remember, the abstract `Mammal` class is a representation of **ALL** mammals, not just the ones we have so far. The correlation of like behavior with all our represented animals does not imply its inclusion in their abstract representation!

For instance, one day, in our wanderings through Australia, we encounter a Duckbilled Platypus[5] . Let's see how it behaves with respect to our 3 methods:

Mammal	**Method**		
	`boolean givesMilk()`	`String makeSound()`	`boolean givesLiveBirth()`
Duckbilled Platypus	true	"growl"	false

Table 2.2

Duckbilled platypus lay eggs!!

Giving live birth is not part of the definition of a mammal. On the other hand, the question of whether or not the animal gives live birth can always be asked of any animal, including all mammals. The result may be true or false however, so the method must be abstract at the `Mammal` level.

Our class structure should look like this (Mammal2.java[6]):

[4] http://legacy.cnx.org/content/m11796/latest/Mammal0.java
[5] http://en.wikipedia.org/wiki/Duckbilled_platypus
[6] http://legacy.cnx.org/content/m11796/latest/Mammal0.java

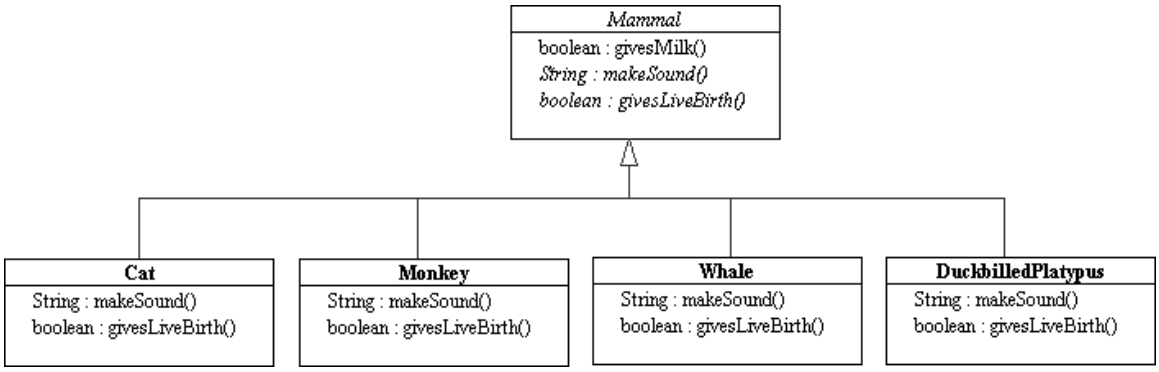


Figure 2.4: Properly abstracted model.

Hoisting does not guarantee proper abstraction. Hoisting should be driven by a need for abstraction, not by coincidence.

Another key notion that the union pattern emphasizes is **levels of abstraction**. What we see is that the concept of a liquid is more abstract than milk. In turn, the general concept of milk is more abstract than "2% milk" or "skim milk" which would be subclasses of milk. In general we can see that a superclass is a distinctly higher level of abstraction than any of its subclasses. One of the key tools we use to help us design and build high quality object-oriented systems is careful attention to the abstraction level at any given moment.

Good OOP code always maintains a consistent level of abstraction.

Abstraction levels are links in a chain. A chain is only as strong as its weakest link. A program is only as abstract as its lowest abstraction level.

Levels of abstraction illustrate another important aspect of an OO program. Since a superclass represents the union of the subclasses or conversely, that the superclass can be represented by any of its subclasses, we see that the superclass is an embodiment of all the **invariant** aspects of the subclasses. That is, the superclass's definition is all that is abstractly equivalent about the subclasses–all that does not change from subclass to subclass. Note that this does **not** imply that the values of common fields are necesarily the same, just that, perhaps, that the field exists. Not does it imply that what is common to all the subclasses is necessarily what is abstractly equivalent about them (see the note above). The differences between the subclasses is what creates the variations in how the program behaves when any given subclass is used in place of the superclass. We call this the **variant** aspects of the system.

The total behavior of a program is the combination of its variant and invariant behaviors.

2.1.2 Inheritance and Polymorphism

Inheritance and polymorphism are really just two ways of looking at the same class relationship.

Inheritance is looking at the class hierarchy from the bottom up. A subclass inherits behaviors and attributes from its superclass. A subclass automatically possesses certain behaviors and/or attributes simply

because it is classified as being a subclass of an entity that possesses those behaviors and/or attributes. That is, a cherry can be said to automatically contain a seed because it is a subclass of Fruit and all fruit contain seeds.

Inheritance is useful from a **code reuse** perspective. Any (non-private) code in the superclass does not have to be replicated in any of the subclasses because they will automatically inherit those behaviors and attributes. However, one must be very careful when transferring common code from the subclasses to the superclass (a process called "**hoisting**"), as the proper abstraction represented by the superclass may be broken (see note above).

Polymorphism, on the other hand, is looking at the class hierarchy from the top down. Any subclass can be used anywhere the superclass is needed because the subclasses are all abstractly equivalent to the superclass. Different behaviors may arise because the subclasses may all have different implementations of the abstract behaviors defined in the superclass. For instance, all liquids have a boiling temperature. They may have different values for that boiling temperature which leads to different behaviors at any given temperature.

Polymorphism is arguably the more useful perspective in an object-oriented programming paradigm. Polymorphism describes how an entity of a lower abstraction level can be substituted for an entity of a higher abstraction level and in the process, change the overall behavior of the original system. This will be the cornerstone that enables us to build OO systems that are flexible, extensible, robust and correct.

2.1.3 Exploring Polymorphism

Let's explore some different ways in which polymorphism presents itself. Consider the following example of the union design pattern:

```
/**
* An interface that represents an operation on two doubles
*/
public interface IBinaryOp {
double apply( double x, double y);  // all interface methods are public and abstract by default
}

/**
* An IBinaryOp that adds two numbers
*/
public class AddOp implements IBinaryOp {
public double apply( double x, double y) {
return x+y;
}
}

/**
* An IBinaryOp that multiplies two numbers
*/
public class MultOp implements IBinaryOp {
public double apply( double x, double y) {
return x*y;
}

public String getDescription() {
return "MultOp is a multiplying function.";
}
```

```
}
```

Exercise 2.1.1 *(Solution on p. 35.)*
Is the following legal code? `IBinaryOp bop = new IBinaryOp();`

Exercise 2.1.2 *(Solution on p. 35.)*
Is the following legal code? `IBinaryOp bop = new AddOp();`

Exercise 2.1.3 *(Solution on p. 35.)*
Given the above declaration and assignment of `bop`, is the following assignment then possible? `bop = new MultOp();`

Exercise 2.1.4 *(Solution on p. 35.)*
Suppose we have `bop = new AddOp();` , what is the result of `bop.apply(5,3)` ?

Exercise 2.1.5 *(Solution on p. 35.)*
Suppose we now say `bop = new MultOp()`, what is the result of `bop.apply(5,3)` now?

Exercise 2.1.6 *(Solution on p. 35.)*
Suppose we have some variable, called `myOp` of type `IBinaryOp` what is the result of `myOp.apply(5,3)`?

Exercise 2.1.7 *(Solution on p. 35.)*
Suppose we have `bop = new MultOp()`, is it legal to call `bop.getDescription()` ?

Exercise 2.1.8 *(Solution on p. 35.)*
Is the following legal code? `AddOp aop = new AddOp()`

Exercise 2.1.9 *(Solution on p. 35.)*
Given the declaration in the previous exercise, is the following legal? `aop = new MultOp()`

Exercise 2.1.10 *(Solution on p. 35.)*
Suppose we have definitions of `aop` and `bop` from above. Is the following legal? That is, can we compile and run the folowing statement without error? `bop = aop;`

Exercise 2.1.11 *(Solution on p. 35.)*
Is the converse legal as well? That is, using the above definitions, can we compile and run the following statement? `aop = bop;`

2.2 Ballworld, inheritance-based[7]

In this module we will explore many OOP concepts by examining the program "Ballworld". Download the code for Ballworld here[8] (**Link Temporarily Disabled.** Please contact the authors for the code.).

[7] This content is available online at <http://legacy.cnx.org/content/m11806/1.8/>.
[8] See the file at <http://legacy.cnx.org/content/m11806/latest/>

UML class diagram of Ballworld

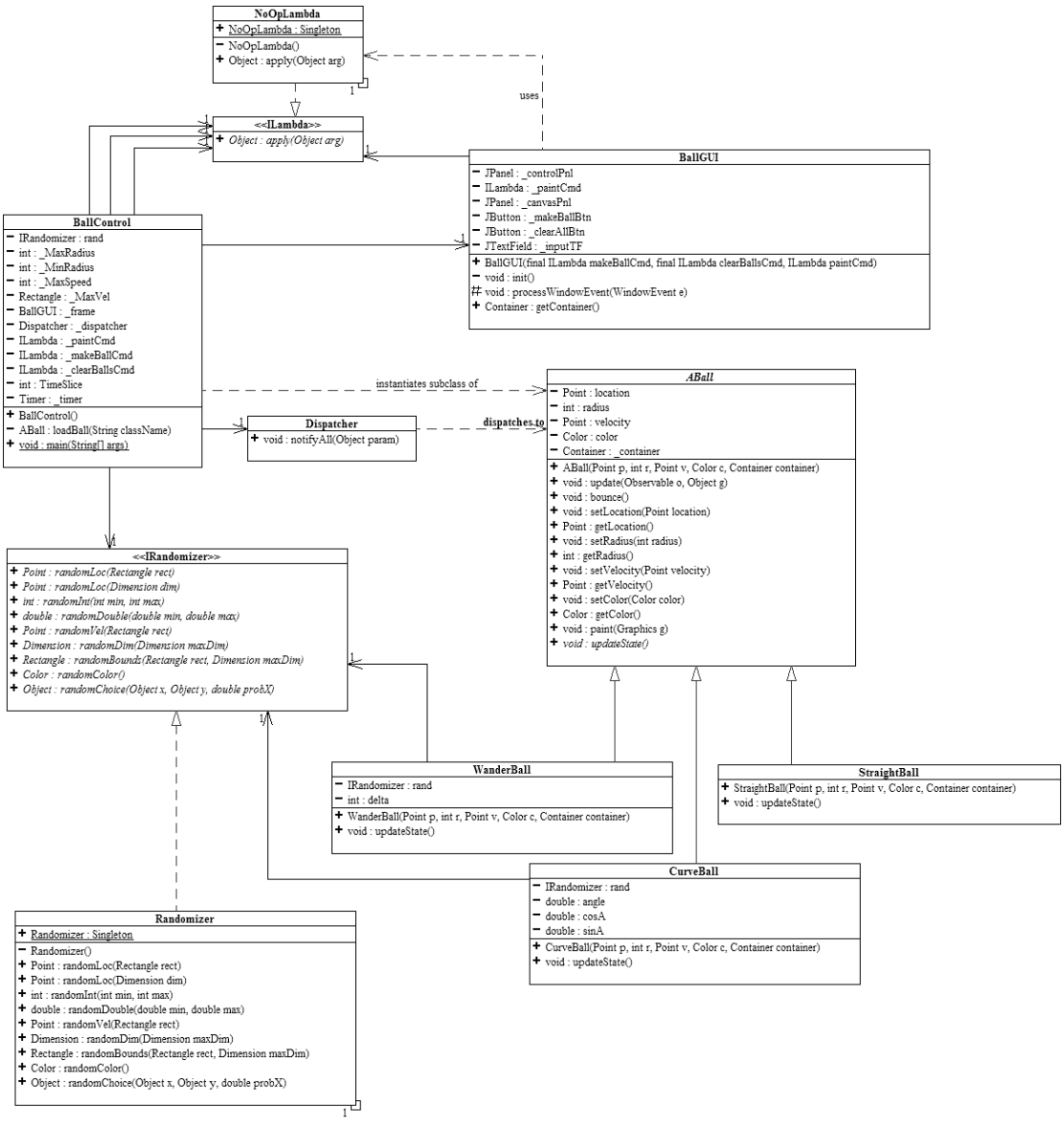


Figure 2.5: Note the union design pattern w.r.t. `ABall` and its subclasses.

To run Ballworld, load the files into DrJava and right-click the BallControl file. Select "*Run Document's Main Method*" to run the program. From the command line, go to the uppermost Ballworld directory and compile all the files in both the directories ("*javac ballworld/*.java*" and "*javac command/*.java*") and then run the program as such: "*java ballworld.BallControl*". The *Make Ball* button creates an instance of whatever `ABall` class is typed into the text field on the left. The "*Clear All*" button clears any balls off of the screen.

In a nutshell, the way Ballworld works is akin to a flip-book animation: The `BallControl` class contains

a `Timer` that "ticks" every 50 milliseconds. Every time the timer ticks, the panel in the `BallGUI` upon which the balls are to be drawn is requested to repaint itself. When the panel repaints, it also tells the `Dispatcher` to notify every `ABall` in the system to update itself. When an `ABall` updates, it updates any internal values it has (its "state") such as its color, radius, and/or velocity. It then moves its position by an amount corresponding to its velocity and draws ("paints") itself onto the panel. Since this is happening 20 times a second, the balls appear to be moving and/or changing and thus the animation is acheived. The `ILambda` interface3 enables the `BallGUI` to communicate in a generic, decoupled manner to the `BallControl` and the `Randomizer` class is a utility class that provides methods to produce various random values needed by the system. Much of the code in Ballworld is significantly more sophisticated than what has been covered in the course so far–it will be covered soon, don't worry!

2.2.1 Abstract Classes

First, we will focus on the union design pattern between `ABall`, `WanderBall`, `CurveBall` and `StraightBall`. In a union design pattern, we see that the superclass represents an abstraction of the union of its subclasses. For instance, a fruit is an abstraction of specific concrete classes such as apple and pear. A fruit embodies the common characteristics of its subclasses even if it doesn't completely describe the exact nature of those characteristics. In a fruit, there is a seed. However, the notion of "fruit" doesn't specify exactly the number, size, color or shape of its seed(s). It only specifies that it does indeed have a seed. Likewise, a fruit has the behavior of ripening. Apples, oranges, and bananas all ripen differently and at different rates. The abstract fruit notion does not specify the specific nature of the ripening behavior, just simply that it does have that behavior. In such, we see that we can never have a fruit that is not a specific class of fruit, such as an orange or grape.

Corresponding to the above notions, abstract classes in Java cannot be instantiated. Abstract classes are denoted by the `abstract` keyword in the class definition:

```
public abstract class AFruit {...}
```

By convention, the classnames of abstract classes always begin with "A".

In Ballworld, `ABall` is an abstract class representing a circular ball that has a number of properties: a color, a position, a velocity, etc. The abstract ball also has some defining behaviors, such as that all balls should paint a filled, colored circle when requested to display themselves on a graphics context (a panel). Likewise all balls know how to bounce off the walls of the container. These concrete behaviors are called "default behaviors" because all subclasses of `ABall`, such as `StraightBall` and `CurveBall`, automatically get these behaviors by default. One of the most common and useful reasons for using an abstract class is to be able to define the default behaviors for all the subclasses.

2.2.1.1 Abstract Methods

But what about the abstract behaviors that abstract classes exhibit? For instance the abstract "ripening" behavior of a fruit? At the abstraction level of a fruit, the exact implentation of ripening cannot be specified because it varies from one concrete subclass to another. In Java, this is represented by using the keyword abstract as part of the signature of a method which has no code body:

```
public abstract class AFruit {
// rest of the code

public abstract void ripen();
}
```

There is no code body because it cannot be specified at this abstraction level. All that the above code says is that the method does exist in all `AFruit`. The specific implmentation of method is left to the subclasses, where the method is declared identically except for the lack of the `abstract` keyword:

```
public class Mango extends AFruit {
// rest of code

public void ripen() {
// code to ripen a mango goes here
}
}

public class Tomato extends AFruit {
// rest of code

public void ripend() {
// code to ripen a tomato goes here
}
}
```

The technical term for this process is called **overriding**. We say that the concrete methods in the subclasses **override** the abstract method in the superclass.

Note that if a class has an abstract method, the class itself must be declared `abstract`. This simply because the lack of code in the abstract method means that the class connot be instantiated, or perhaps more importantly, it says that in order for a class to represent abstract behavior, the class itsefl must represent an abstract notion.

Overriding is not limited to abstract methods. One can override any concrete method not declared with the `final` keyword. We will tend to avoid this technique however, as the changing of behavior as one changes abstraction levels leads to very unclear symantics of what the classes are actually doing.

In Ballworld we see the abstract method `updateState`. Abstract methods and classes are denoted in UML diagrams by italic lettering. This method is called by the `update` method as part of the invariant process of updating the condition of the ball every 50 milliseconds. The update method does 4 things:

1. Update the state of the ball by calling the abstract `updateState` method.
2. Move (translate) the position of the ball by adding the velocity to it.
3. Check if the ball needs to bound off a wall.
4. Paint the ball up on the screen.

This technique of calling an abstract method from inside of a concrete method is called the **template method design pattern**–which we will get to later in the course.

`ABall.updateState()` is abstract because at the abstraction level of `ABall`, one only knows that the ball will definitely do something with regards to modifying (perhaps) its internal field values(its "state"). Each subclass will do it differently however. The `StraightBall` will do nothing in its `updateState` method because a ball with a constant (unchanging) velocity will travel in a straight line. Remember, **doing nothing is doing something**! The `CurveBall`'s `updateState` method uses sines and cosines to turn the velocity by a fixed (though randomly chosen) angle at every update event. You can imagine that other possible subclassses could do things such as randomly change the velocity or change the radius of the ball or change the color of the ball.

There is no code in the entire Ballworld system that explicitly references any of the concrete `ABall` subclasses. All of the code runs at the level of abstraction of an abstract ball. The differences in behavior of the various balls made on the screen using the different concrete subclasses is strictly due to **polymorphism**. New types of balls can be added to the system without recompiling **any** of the existing code. In fact, new types of balls can be added without even stopping the Ballworld program!

2.2.2 Abstract classes vs. Interfaces

Subclasses have a different relationship between interfaces and abstract superclasses. A subclass that implements an interface is saying simply that it "acts like" that specified by the interface. The class makes no statements however about fundamentally what it actually is. An actor implements a fiercesome alien from a distant plantet in one movie and a fickle feline in another. But an actor is actually neither. Just because the actor protrayed a interplanetary alien, doesn't mean that the actor fundamentally possessed all the abilities of such an alien. All it says is that in so far the context in which the actor was utilized as the alien, the actor did implement all the necessary behaviors of the alien.

A subclass is fundamentally an example of its superclass. A subclass automatically contains all the behaviors of its superclass because it fundamentally **is** the superclass. The subclass doesn't have to implement the behaviors of its superclass, it already has them. An actor is a human and by that right, automatically possesses all that which makes a human: one head, two arms, 10 toes, etc. Note that this is true even if the abstract class has 100% abstract methods–it still enforces a strict taxonomical hierarchy.

> `implements` *is about* ***behaving***, `extends` *is about* ***being***.

2.2.3 Variant vs. Invariant Behaviors

A crucial observation is that the the Ballworld code that manages the GUI (`BallGUI`) and the ball management (`BallControl`, `Dispatcher`, etc.) only deal with the abstract ball, `ABall`. That is, they represent **invariant** behavior at the abstract ball level. The display, creation and managment of the balls is independent of the particular kinds of concrete balls that is being handled. The main Ballworld framework can thus handle **any** type of `ABall`, past, present and future.

`StraightBall`, `CurveBall` and `WanderBall` are thus concrete variants of `ABall`. They represent the **variant** behaviors of the system. Other than in their constructors (which will prove to be a significant point when this inheritance-based model is compared to a more advanced composition-based model), these concrete subclasses only code the abstract variant behavior associated with a ball, namely the `updateState` method. Inheritance gives any instantiation of these classes both the invariant behaviors from the `ABall` superclass plus the variant behaviors from the subclass.

The Ballworld code demonstrates the importance of the concept of **separation of variant and invariant behaviors**.

> *Clearly and cleanly separating the variant and invariant behaviors in a program is crucial for achieving flexible, extensible, robust and correct program execution.*

Where and how to separate the variant and invariant behaviors is arguably the most important design consideration made in writing god software.

2.2.4 Java Syntax Issues

2.2.4.1 Packages

Packages are way that Java organizes related classes together. Packages are simply directories that contain the related code files. Each class file in a package directory should have the line `package XXX;` at its top, where the `XXX` matches with the directory. name. If neither `public` nor `private` (nor `protected` – i.e. a blank specifier) is used to specify the visibility level of a class or method, then that method can be seen by other members of the package but not by those outside of the package. To use the public classes in a package, the `import myPackage.*;` syntax is used. This tells the Java compiler to allow all the public classes in the `myPackage` directory. Packages can be nested, though each level must be imported separately.

2.2.4.2 Static fields and methods

Static fields and methods, denoted by the `static` keyword in their declarations, are fields and methods that can be accessed at a class level, not just an object level. In general these are values or behaviors that one wishes for all instances of a class to have access to. These values and behaviors are necessarily independent of the state of any particular instance. Static fields and methods are often referred to as "**class variables**" and "**class methods**".

An examples of a class variables are `Math.PI` and `Color.BLUE` or `Color.RED`. These are universal values associated with math and color respectively and thus do not need an object instance to be viable. By convention, all static field names are in all capitol letters. A static field is referenced simply by writing the class name followied by a period and then by the field name. No instantiations are necessary.

The `Randomizer` class contains numerous static methods. This is because each of the methods to produce various random values is independent of each other and that the process in each method does not affect nor is affected by the state of the rest of the class. Essentially, this entails that the class contain no non-static fields. A class as such is referred to as being "**stateless**". Just like a static field, a static method is invoked in the same manner as the static fields: `ClassName.staticMethodName(...)` Classes with static methods are usually utility classes that are used to hold a set of related functional processes, e.g. `Randomizer` holds a collection of random value generators. Likewise, `Math` holds a combination of static values, such as `PI` and static methods such as `sin`() and `cos`().

There is one very special static method with the following **exact** signature:

```
public static void main(String[] args)
```

This method, found in `BallControl`, is the method that Java uses to start programs up. Since OO programs are systems of interacting objects, this static "main" method is used to create the first object(s) and get the program up and running. So when Java starts a program, it looks for this and only this method.

2.2.4.3 Calling methods of the superclass

When concrete methods or the constructor of a superclass are overriden, sometimes it is necessary or desirable to call the original superclass behavior from the subclass. A common reason for this is that the subclass's behavior is simple an addition to the superclass behavior. One does not want to replicate the superclass code, so a call to the superclass's original methods is required at some point in the subclasses overriding method. To accomplish this, Java uses the `super` keyword. `super` refers to the superclass instance, just as `this` refers to the class instance itself (the subclass here). Note that technically, `super` and `this` are the same object – think of it as the difference between the *id* and the *ego*. (Does that mean that a coding mistake wiith respect to `super` and `this` is a Freudian slip?)

Suppose the superclass has a method called `myMethod()` which the subclass overides. For the subclass to call the superclass's `myMethod`, it simply needs to say `super.myMethod()`. Contrast this to the subclass calling its own `myMethod`: `this.myMethod()` (note: Java syntax rules allow for the `this` to be omitted).

To make a call to the superclass's constructor the subclass simply says `super(...)`, supplying wahtever parameters the superclass constructor requires. This is a very common scenario as the the subclass almost always needs to superclass to initialize itself before it can perform any additional initializations. Thus the `super(...)` call must be the first line in the subclass's constructor. If the no-parameter constructor of the superclass is required, the call to `super` can be omitted as it will be automatically performed by the Java run-time engine. This of course presumes that the superclass's no-parameter constructor exists, which it does **not** if a parameterized constructor has been declared without explicitly declaring the no-parameter constructor.

2.3 Ballworld, composition-based[9]

In this module we will explore what is gained by modifying the inheritance-based Ballworld system (Section 2.2) into a composition-based system.

In the inheritance-based Ballworld system, we were able to generate quite a bit of flexibility and extensibility. For instance, we could develop new kinds of balls and add them into the system without recompiling the rest of the code. This was accomplished by having the invariant ball creation and management code deal only with the abstract ball while the variant behaviors were encapsulated into the concrete subclasses.

2.3.1 The Problem with Inheritance

Inheritance seems to work quite well, but suppose we want to do more than just put different kinds of balls on the screen? What if we wanted to be able to change how a ball behaves, **after** it has been created? What if we want create balls that do a multiple of different behaviors, such as change color and radius? While working solutions using an inheritance-based system do exist, they are cumbersome, inefficient and most importantly, inconsistent with any sort of clear abstract model of what balls should be like.

The problem lies in the very nature of inheritance. When we attempted to separate the variant from the invariant behaviors, we overlooked a crucial aspect of inheritance. In our model, the superclass represented the invariant behaviors of a ball while the subclasses represented the variant behaviors. The separation seemed clear enough in the UML diagram, except that when one has an actual object **instance**, **both** the superclass and subclass behaviors are bound into a **single** entity. A ball **object** cannot change its variant `updateState` behavior because it is inextricably bound with to the invariant behaviors. A ball object cannot be composed of multiple `updateState` behaviors because that code cannot be isolated from the rest of the ball's code. If you want a curving behavior, you have to get it packaged in a whole ball object–you can't get just the behavior.

A clear sympton of this problem is the common code to call the superclass constructor found in all the subclasses' constructors. This tells us that the superclass is really right there in the subclass with everything else. The fact that the code is repeated from class to class says that it is invariant code in the middle of what we want to be variant code.

> *The inheritance-based model of Ballworld does not separate the variant and the invariant at the proper place. There is invariant code mixed together with the variant code.*

That's why they can't be separated and the invariant behaviors are dragged along with the variant behaviors. This is what makes dynamically changing behaviors and multiply composed behaviors so difficult in this system.

2.3.2 Pizzas and Shapes

To understand what we can do to remedy the problems with our inheritance-based model, let's digress for a bit and consider a simple model of pizzas. Here, we have a pizza which has a price and **has a** shape. A shape, be it a circle, square, rectangle of triangle, is capable of determining its own area. A pizza, when requested to calculate its price per square inch, simply takes its price and divides it by the area of its shape. To obtain that area, the `Pizza` **delegates** to the `IShape`, since it is the shape that knows how to calculate its area, not the pizza.

[9] This content is available online at <http://legacy.cnx.org/content/m11816/1.6/>.

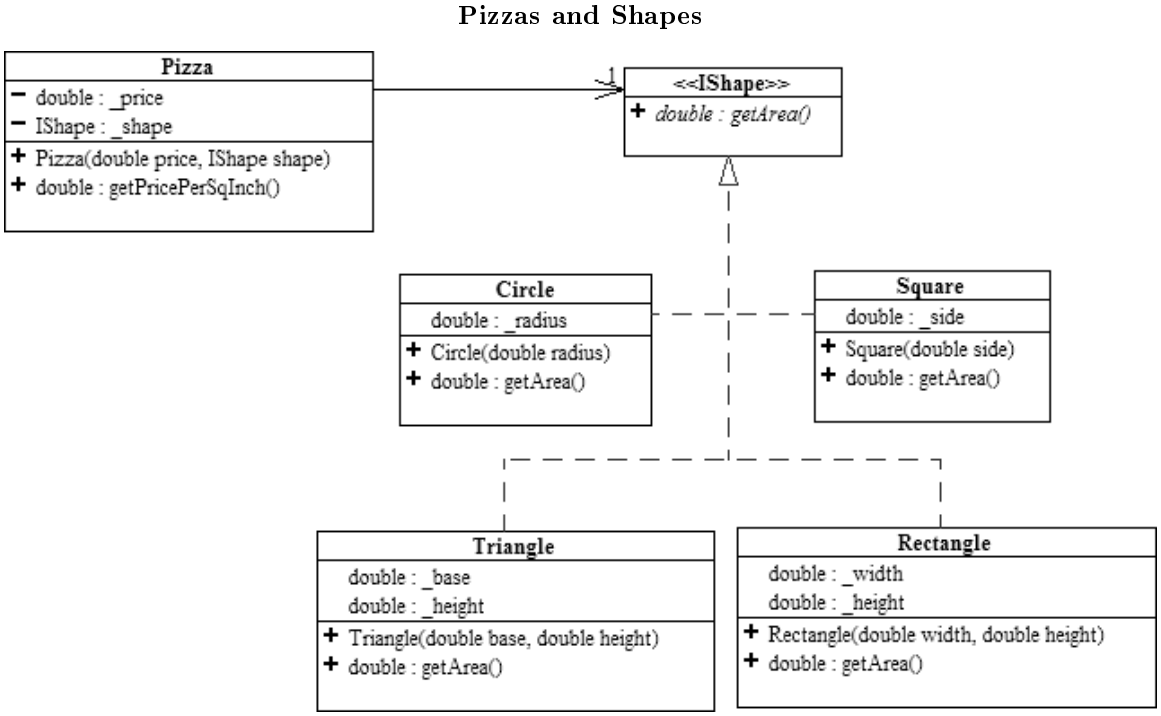


Figure 2.6: A pizza **has-a** shape, which is able to calculate its area.

Delegation is the handing of a calculation off to another object for it process. Here, the pizza is only interested in the result of the area calculation, not how it is performed.

To the pizza, the shape represents an abstract algorithm to calculate the area.

The `Pizza` and the `IShape` classes represent the invariant processes involved with calculating the price per square inch ration, while the concrete `Circle`, `Square`, `Triangle` and `Rectangle` classes represent the variant area calculations for different shapes. What wee see from this example is that

objects can be used to represent pure behavior, not just tangible entities.

Interfaces are particularly useful here as they are expressly designed to represent pure, abstract behavior.

2.3.3 From Inheritance to Composition

Coming back to Ballworld, we see that the `updateState` method in `ABall` is an abstract algorithm to update the state of the ball. So, just as in the pizza example, we can represent this algorithm, **and just this algorithm**, as an object. We can say that a ball **has an** algorithm to update its state. Another wa of saying this is to say that the ball has a **strategy** to update its state. We can represent this by using composition. Instead of having an abstract method to update the state, we model a ball as having a reference to an `IUpdateStrategy` object. The code for update thus becomes

```
public void update(Observable o, Object g)
{
  _strategy.updateState(this);   // update this ball's state using the strategy
```

```
    location.translate (velocity.x, velocity.y);  // move the ball
    bounce();  // bounce the ball off the wall if necessary
    paint((Graphics) g); // paint the ball onto the container
  }
```

The ball hands a reference to itself, `this`, to the strategy so that the strategy knows which ball to update. The variant updating behaviors are now represented by concrete implementations of the `IUpdateStrategy` interface.

Composition-based Ballworld

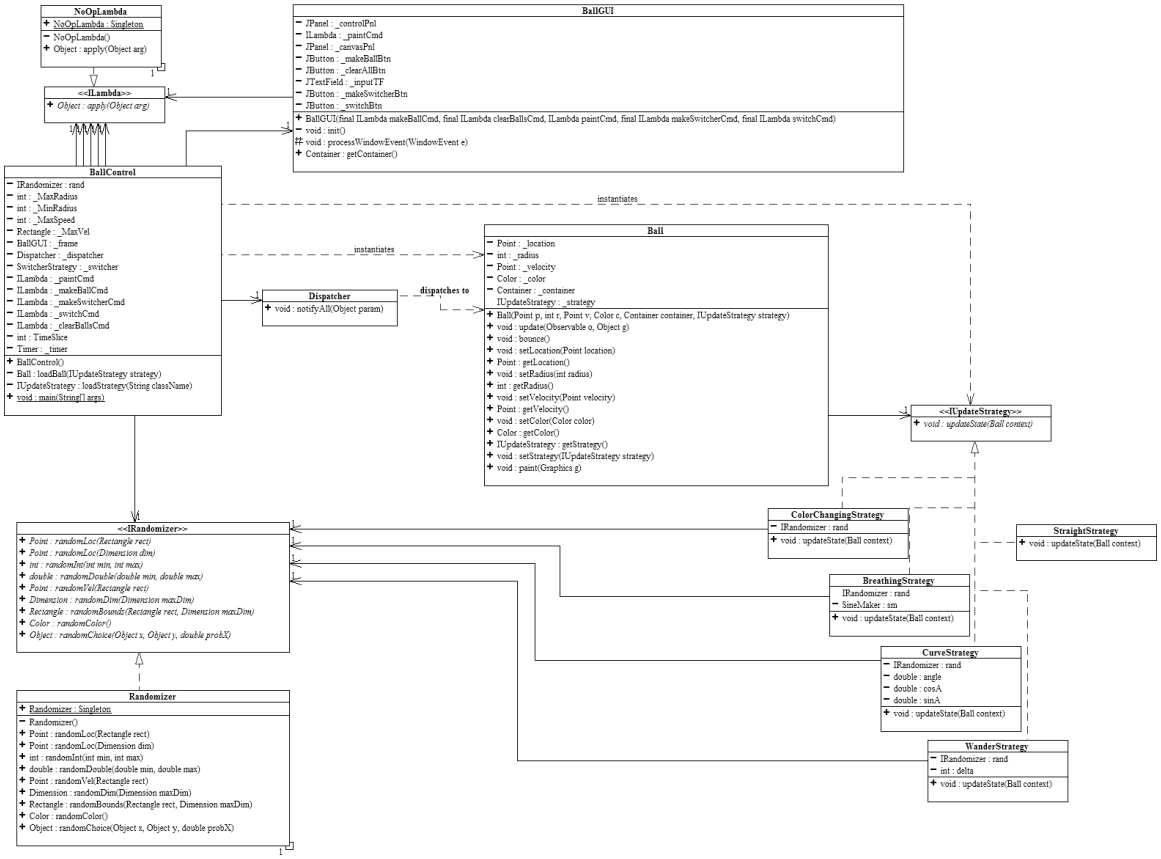


Figure 2.7: `Ball` is now a concrete class and its subclasses have been eliminated in favor of being composed with an abstract strategy.

(Note that the `Randomizer` class has been redesigned to eliminate its static methods. One new method has been added as well.)

There are a number of very important points to notice about this new formulation:

- The modified `ABall` class now contains 100% concrete code and thus should not be abstract anymore.
  - `ABall` has been renamed to simply `Ball`.
  - Accessor methods for the strategy (`getStrategy` and `setStrategy`) have been added.

- **The `Ball` class is still 100% invariant code.**

- The `CurveBall`, `StraightBall`, etc. subclasses are no longer needed as their variant behaviors have been moved to the `IUpdateStrategy` subclasses.
  - Effectively what has happened is that the `updateState` method has been moved from the `ABall` subclasses and embodied into their own classes.
  - The `IUpdateStrategy` subclasses do not inherit anything from `Ball`, hence they do not contain any invariant code.
  - **The strategies are thus 100% variant code.**
- The reference to the ball during the updating process has gone from a peristent communication link (implicitly, `this`), to a transient communication link (`host`).
- This composition-based model divides the code exactly between the variant and invariant behaviors– this is the key to the power and flexibility it generates.

This new composition-based model of Ballworld is an example of the Strategy Design Pattern[10] . The strategy design pattern allows us to isolate variant behaviors on a much finer level than simple inheritance models.

2.3.3.1 Composing Behaviors

So far, all of our redesigning has resulted in a system that behaves exactly as it did when we started. But what one finds very often in developing systems is that in order to make two steps forward, one must first make one step backwards in order to fundmentally change the direction in which they are going. So, even though it looks like our system has not progressed because it still does exactly the same thing, we are indeed in a very different position, architecturally. By freeing the variant behaviors from the invariant ones, we have generated a tremendous amount of flexibility.

2.3.3.1.1 Balls that change their strategies

Let's consider a the notion of a ball that changes its behavior. Since we have modeled a ball as **having** a strategy, we can simply say that in some manner, it is the ball's **strategy** that changes. We could say that the ball changes its strategy, but since the ball doesn't know which strategy it has to begin with, it really doesn't know one strategy from another. One could argue that it therefore can't know when or if it should ever change its strategy. Therefore, the ball cannot be coded to change its own strategy! So, whose baliwick is the changing of the strategy?

Since the changing of a strategy is a strategy for updating the ball, it is the strategy that determines the change. The strategy changes the strategy! Let's consider the following strategy:

```
package ballworld;
import java.awt.*;

public class Change1Strategy implements IUpdateStrategy {

   private int i = 500;  // initial value for i

   public void updateState(Ball context) {
     if(i==0) context.setStrategy(new CurveStrategy());   // change strategy if i reaches zero
     else i--;  // not yet zero, so decrement i
   }
}
```

[10] http://www.exciton.cs.rice.edu/JavaResources/DesignPatterns/StrategyPattern.htm

This strategy acts just like a `StraightStrategy` for 500 updates and then it tells the ball (its `context`) to switch to a `CurveStrategy`. Once the `CurveStrategy` is installed, the ball becomes curving, without the need for any sort of conditionals to decide what it should do. The context ball fundamentally and permanently **becomes** curving.

Exercise 2.3.1 *(Solution on p. 35.)*

What would happen if you had two strategies like the one above, but instead of replacing themselves with `CurveStrategy`'s , they instead instantiated each other?

A key notion above is that a strategy can contain another strategy. In the above example, the `Change1Strategy` could have easily pre-instantiated the `CurveStrategy` and saved it in a field for use when it was needed. But the does it matter exactly which concrete strategy is being used? If not, why not work at a higher abstraction level and let one strategy hold a reference to an **abstract** strategy? For instance, consider the following code:

```
package ballworld;
import java.awt.*;

public class SwitcherStrategy implements IUpdateStrategy {

   private IUpdateStrategy _strategy = new StraightStrategy();

   public void updateState(Ball context) {
     _strategy.updateState(context);
   }

   public void setStrategy(IUpdateStrategy newStrategy) {
     _strategy = newStrategy;
   }
}
```

This strategy doesn't look like it does much, but looks are deceiving. All the `SwitcherStrategy` does is to delegate the `updateState` method to the `_strategy` that it holds. This does not seem much in of itself, but consider the fact that the `SwitcherStrategy` also has a settor method for `_strategy`. This means that the strategy held can be changed at run time! More importantly, suppose a ball is instantiated with a `SwitcherStrategy`. The behavior of the ball would be that of whatever strategy is being held by the `SwitcherStrategy` since the switcher just delegates to the held strategy. If one were to have a reference to that `SwitcherStrategy` instance from somewhere else, one could then change the internal strategy. The ball is none the wiser because all it has is a reference to the `SwitcherStrategy` instance, which hasn't changed at all! However, since the held strategy is now different, the ball's **behavior** has completely changed! This is an example of the Decorator Design Pattern[11] , where the `SwitcherStrategy` class is formally called the **decorator** and the held strategy is formally called the **decoree**. In theoretical terms, the decorator is what is known as an **indirection layer**, which is like a buffer between two enities that enables them to depend on each other but yet still be free to move and change with respect to each other. A very useful analogy for indirection layers is like the thin layer of oil that will enable two sheets of metal to slide easily past each other.

2.3.3.1.2 Balls with multiple, composite behaviors

Now that we can dynamically change a ball's behavior, let's tackle another problem:

[11] http://www.exciton.cs.rice.edu/JavaResources/DesignPatterns/DecoratorPattern.htm

Exercise 2.3.2 *(Solution on p. 35.)*
How can we have balls with multiple behaviors but yet not duplicate code for each one of those behaviors?

Let's start with a very straightforward solution:

```
package ballworld;
import java.awt.*;

public class DoubleStrategy implements IUpdateStrategy {

   private IUpdateStrategy _s1 = new CurveStrategy();
     private IUpdateStrategy _s2 = new BreathingStrategy();

   public void updateState(Ball context) {
     _s1.updateState(context);
     _s2.updateState(context);
   }
   }
```

Ta da! No problem. The `DoubleStrategy` simply holds two strategies and delegates to each of them in turn when asked to `updateState`. So why stop here?

```
package ballworld;
import java.awt.*;

public class TripleStrategy implements IUpdateStrategy {

   private IUpdateStrategy _s1 = new CurveStrategy();
     private IUpdateStrategy _s2 = new BreathingStrategy();
     private IUpdateStrategy _s3 = new BreathingStrategy();

   public void updateState(Ball context) {
     _s1.updateState(context);
     _s2.updateState(context);
     _s3.updateState(context);
   }
   }
```

We're on a roll now! We could go on and on, making as complex a strategy as we'd like, making a new class for each combination we want. But somewhere around the 439'th combination, we get mightly tired of writing classes. Isn't there an easier way?

Abstraction is the key here. We want to write code that represents that abstraction of multiple, composite strategies. Does what we were doing above depend on the particular concrete strategies that we were using? No? Then we should eliminate the concrete classes, raise the abstraction level and use the abstract superclass (interface) instead. For a combination of two behaviors, we end up with the following:

```
package ballworld;
import java.awt.*;

public class MultiStrategy implements IUpdateStrategy {
```

```
   private IUpdateStrategy _s1;
   private IUpdateStrategy _s2;

   public MultiStrategy(IUpdateStrategy s1, IUpdateStrategy s2) {
     _s1 = s1;
     _s2 = s2;
   }

   public void updateState(Ball context) {
     _s1.updateState(context);
     _s2.updateState(context);
   }
}
```

Notice how we have added a constructor that enables us to initialize the two abstract strategy fields. All we have to do is to construct a `MultiStrategy` object with the two desired strategies, and we're good to go!

Exercise 2.3.3 *(Solution on p. 35.)*

So if we want three behaviors, all we have to do is to make the same sort of thing but with 3 abstract strategy fields, right?

Thus, with just a `Multistrategy` we are capable of composing arbitrarily complex behaviors!

2.3.3.1.3 Composite Patterns

So what have we wrought here? Let's take a look at the UML diagram of our to most abstract strategies.

SwitcherStrategy and MultiStrategy

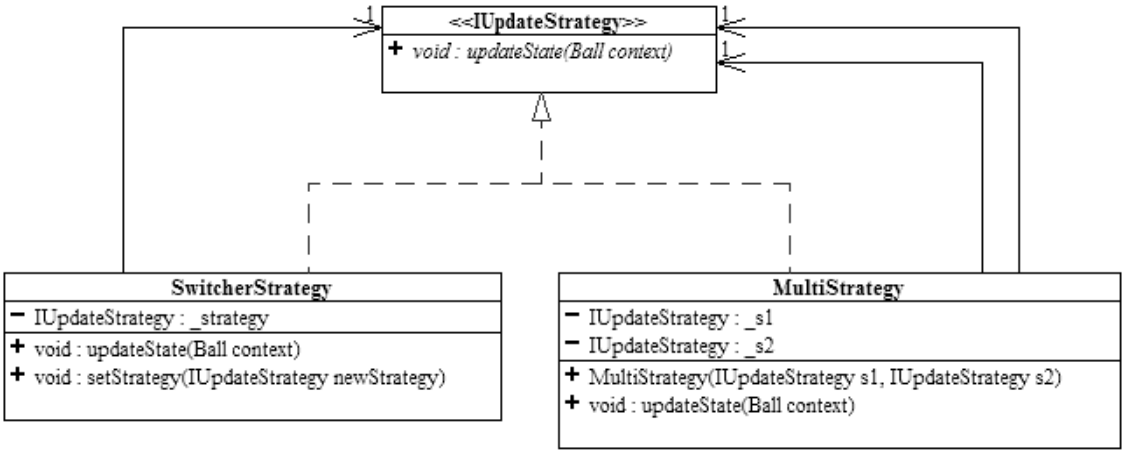


Figure 2.8: Note that the subclasses hold references to their own superclasses!

The key to the power that lies in the `SwitcherStrategy` and the `MultiStrategy` lies in the fact that they hold references to their own superclass, `IUpdateStrategy`. This is what enables them to be create any behavior they want, including combinations of behaviors and dynamic modifications of behaviors. This

self-referential class structure is known as the Composite Design Pattern[12] (The Decorator pattern can be considered to be specialized form of the Composite pattern). The massive power, flexibility and extensiblility that this pattern generates warrants further, more formal study, which is where we're heading next. Stay tuned!

[12] http://www.exciton.cs.rice.edu/JavaResources/DesignPatterns/composite.htm

Solutions to Exercises in Chapter 2

Solution to Exercise 2.1.1 (p. 21)
No, it won't compile. `IBinaryOp` is an interface and does not specify any actual executable code, so it cannot be instantiated.
Solution to Exercise 2.1.2 (p. 21)
Yes! `AddOp` is an concrete class and can be instantiated. `AddOp` is an `IBinaryOp` (technically, `AddOp` **implements** the `IBinaryOp`interface), so `bop` can reference it.
Solution to Exercise 2.1.3 (p. 21)
Yes, for the same reasons as the previous exercise! `MultOp` is an concrete class and can be instantiated. `MultOp` is an `IBinaryOp`, so `bop` can reference it.
Solution to Exercise 2.1.4 (p. 21)
The result is 8 because `bop` refers to an `AddOp` instance, whose `apply` method adds its two input values.
Solution to Exercise 2.1.5 (p. 21)
The result is 15 because `bop` now refers to an `MultOp` instance, whose `apply` method multiplies its two input values.
Solution to Exercise 2.1.6 (p. 21)
It is impossible to tell because it depends on the exact type of the object instance to which `myOp` refers.
Solution to Exercise 2.1.7 (p. 21)
No, because `bop` is a variable of type `IBinaryOp` , which is not defined as having a `getDescription` method. This is true even if bop references an object of type MultOp. This is because the **static typing** of `bop` tells the compiler that it references an `IBinaryOp`, not the particular concrete type of the object it currently references. If we had `MultOp mop = new MultOp()`, then `mop.getDescription()` is perfectly legal.
Solution to Exercise 2.1.8 (p. 21)
Yes, because `aop` is a variable of type `AddOp`, and thus can reference an instance of the same type.
Solution to Exercise 2.1.9 (p. 21)
No, because `aop` is a variable of type `AddOp` , and `MultIOp` is not an `AddOp`, so `aop` cannot reference an instance of `MultOp`.
Solution to Exercise 2.1.10 (p. 21)
Yes! `bop` is a variable of type `IBinaryOp`, and `aop` is defined as referencing an `AddOp` object, which is an `IBinaryOp` .
Solution to Exercise 2.1.11 (p. 21)
Not as written, because `bop` is a variable of type `IBinaryOp` (i.e. staticly typed as such), which and does not necessarily reference an object of type `AddOp`. to which `aop` must reference. That is, a variable of the type of the superclas can always reference an object of the type of any subclass, but a variable of the type of a particular subclass cannot necessarily reference something of the type of its superclass. Another way of saying this is that a superset contains its subsets but not the other way around. The above assignment will cause a compile time error because the compiler cannot know if the assignment is possible. An explicit cast is required to supress the compile time error (`aop = (AddOp) bop`), but may trigger a run-time error if indeed the instance referred to by `bop` is not of type `AddOp`.
Solution to Exercise 2.3.1 (p. 31)
Try it!
Solution to Exercise 2.3.2 (p. 31)
Use the same techniques as before: strategies that hold strategies.
Solution to Exercise 2.3.3 (p. 33)
But isn't a `MultiStrategy` an `IUpdateStrategy` iteself? That is, since a `MultiStrategy` holds `IUpdateStrategy`'s, couldn't a `Multistrategy` hold a `Multistrategy` , which is holding a `Multistrategy` (or two) which is hold a `Multistrategy` , which is holding.....?

Chapter 3

Immutable List Structure

3.1 List Structure and the Composite Design Pattern[1]

3.1.1 Going Shopping

Before I go to the groceries store, I make a list of what I want to buy. Note how I build my shopping list: I start with a blank sheet of paper then I add one item at a time.

When I get to the store, I start buying things by going down my list. For each item I buy, I mark it off the list.

After I am done shopping, I go to the cashier and check out my items.

The cashier scans my items one item at a time. Each time, the cash register prints one line showing the item just scanned together with its price. Again, note how the cash register builds the list: it start with a blank sheet of paper and then add one item at a time. After all items have been scanned, the cashier press a key and "poof", the cash register prints a subtotal, then a tax amount for all the taxable items, then a total amount, and finally a total number of items bought.

At different store, the cash register not only prints out all of the above, but also a total amount of "savings" due to the fact that I have a "member-plus" card. Some other stores don't care to print the total number of items bought at all. Whatever the store, wherever I go, I see "lists" and "list processing" all over.

The check out cash register uses a program to enter the items and print the receipt. At the heart of the program is a container structure to hold data (data structure) and a few algorithms to manipulate the structure and the data it holds. The simplest way to organize data is to structure them in a linear fashion; that is, intuitively, if we can get hold of one data element, then there is exactly one way to get to the next element, if any. We call this linear organization of data the list structure. In order to write program to process lists, it is necessary to define what lists are and express their definitions in terms of code.

3.1.2 What is a list?

Analogous to the notion of a shape, a list is an abstract notion. Recall how I built my list of groceries items? I started with a blank list: an empty list! The empty set!

An empty list is a list that has no element.

It is obvious that there are non-empty lists. But what do we mean by a non-empty list? How can we articulate such an obvious notion? Consider for example the following list consisting of three elements.

- milk
- bread

[1] This content is available online at <http://legacy.cnx.org/content/m15111/1.1/>.

- butter

In the above, we organize the items in a linear fashion with milk being the first element, bread being the next element following milk and butter being the next element following bread. Is there any item that follows butter?

Is

- bread
- butter

a list?

Is

- butter

a list?

Is there a list that follows butter in the above?

A non-empty list is a list that has an element called first and a list called rest.

Note that in the above formulation, the rest of a list is itself a list! The definition of a list is an example of what we call a **recursive** definition: the list contains a substructure that is **isomorphic** to itself.

3.1.3 List Design and the Composite Design Pattern

The UML diagram below captures the recursive data definition of the list data structure.

UML diagram of a list

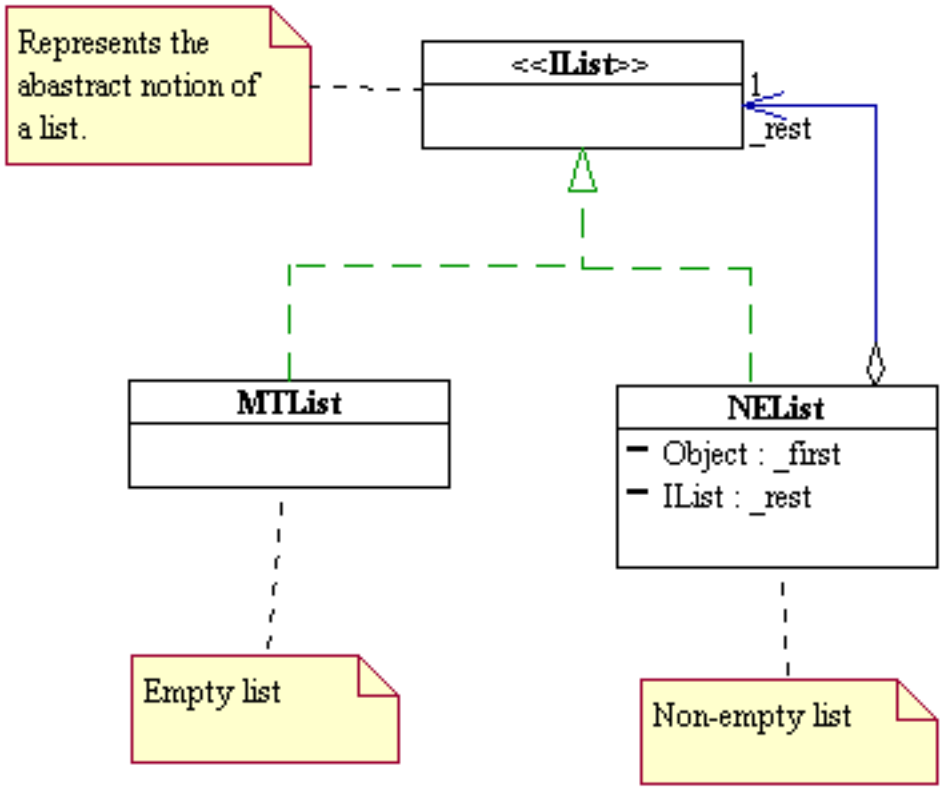


Figure 3.1: A list can be represented using the composite design pattern

This definition translates into Java code as follows.

```
/** * Represents the abstract list structure. */ public interface IList { }
```

```
/**
* Represents empty lists.
*/
public class MTList implements IList {
}
```

```
/**
* Represents non-empty lists.
*/
public class NEList implements IList {
    private Object _first;
    private IList _rest;
}
```

Table 3.1

The above is an example of what is called the **composite design pattern**. The composite pattern is a **structural** pattern that prescribes how to build a container object that is composed of other objects whose structures are **isomorphic** to that of the container itself. In this pattern, the container is called a composite. In the above, IList is called the abstract component, MTList is called the basic component and NEList is called the composite. The composite design pattern embodies the concept of **recursion**, one of the most powerful thinking tool in computing. (There is a subject in theoretical computer science and mathematics called "recursive function theory," which studies the meaning of what computing means and in effect defines in the most abstract form what a computer is and what it can and cannot do.)

3.1.4 List Creation

Now that we have defined what a list is, we ask ourselves how we can process it? What can we do with a list? The above code makes it clear that there is not a whole lot we can do with a list besides instantiating a bunch of MTList objects via the call new MTList() (why?). Now that we are using the full Java language, we need to write a constructor for NEList in order to instantiate non-empty list objects with appropriate first and rest. The Java code for NEList now looks as follows (note how the comments are written).

```
/**
* Represents non-empty lists.
*/
public class NEList implements IList {
   private Object _first;
   private IList _rest;

   /**
   * Initializes this NEList to a given first and a given rest.
   * @param f the first element of this NEList.
   * @param r the rest of this NEList.
   */
   public NEList(Object f, IList r) {
      _first = f;
      _rest = r;
   }
}
```

The list structure as coded in the above is completely **encapsulated**, that is, all internal components (if any) of a list are private and cannot be accessed by any external code. Using the appropriate constructors, we can make a bunch of lists to store data but we cannot retrieve data nor do anything with these lists. In Object-Oriented Programming (OOP) parlance, the list is said to have no behavior at all. As such they are of no use to us.

3.1.5 List Processing

In order to perform any meaningful list processing at all, we need to program more "intelligence" into the list structure by adding appropriate methods to the list to provide the desired behaviors. So instead of asking what we can do with a list, the right question to ask in OOP is "what can a list do for us?" Let us start by presenting a few simple tasks that we want a list to perform and try to figure out how an "intelligent" list would carry out such tasks via some role acting.

3.1.5.1 In class role-acting exercises:

- Compute the length of a list.
- Compute the sum of a list that holds integers.

3.2 List Structure and the Interpreter Design Pattern[2]

3.2.1 List Processing

In the previous lecture, we define what a list is and implement it using the composite design pattern. This list structure is fully encapsulated and does not expose any of its internal components. In order to manipulate such a list without having to make public its internals, we need to add methods to the structure. This lecture discusses the structure of the algorithms on the list.

3.2.1.1 What can a list do?

3.2.1.1.1 Length of a list

Suppose we are given a list L and asked to find out how many elements it has. What should we do? The temptation here is to start thinking about "traversing" the list and keep a count as we go along, and when we encounter the "end" of the list, the count should be the number of elements in the list. But how do we know that that's the right answer? In order to determine whether or not the result obtained by counting as one traverses the list from beginning to end is correct, we have to define what it means to be the number of elements in the list. The number of elements in a list is an abstract notion, isn't it? In order to define such a quantity, we need to go back to the definition of what a list is.

- A **list** is an **abstract** notion of a container structure.
- An empty list is a **list** that has no element
- A non-empty list is a **list** that has an element called first and a **list** called rest.

To define the notion of the number of elements in a list, we need to define what we mean by the number of elements in an empty list and what we mean by the number of elements in a non-empty list.

- The number of elements in a list is an abstract notion because the list is an abstract notion.
- The number of elements of an empty list is 0.
- The number of elements in a non-empty list that contains first and rest is 1 plus the number of elements in rest.

The definition of the number of elements in a list is thus recursive. The recursive characteristic of this definition arises naturally from the recursive characteristic of the list structure. What ever approach we use to compute the number of elements in a list, in order to prove correctness, we must show that the result satisfies the above definition.

Here is the code for the above computation.

[2]This content is available online at <http://legacy.cnx.org/content/m15110/1.3/>.

Top-level abstract list definition

```
package listFW;
/**
 * Represents the abstract list structure.
 */
public interface IList {
    /**
     * Abstract notion of the number of elements in this IList.
     */
    public int getLength();
}
```

Table 3.2

Concrete list implementations

```
package listFW;
/**
 * Represents empty lists.
 */
public class MTList implements IList {
    /**
     * The number of elements in an
     * empty list is zero.
     */
    public int getLength() {
        return 0;
    }
}
```

```
package listFW;
/**
 * Represents non-empty lists.
 */
public class NEList implements IList {
   private Object _first;
   private IList _rest;
   // Constructor ommitted.

   /**
    * The number of elements in a non-empty
    * list is the number of elements of its
    * rest plus 1.
    */
   public int getLength() {
       return 1 + _rest.getLength();
   }
}
```

Table 3.3

The above coding pattern is an example of what is called the **interpreter design pattern**: we are interpreting the abstract behavior of a class (or interface) in each of the concrete subclasses (or implementations). The composite pattern is a pattern to express the structure of a system, while the interpreter pattern is used to express the behaviors (i.e. methods) of the system. The interpreter pattern is usually applied to coding methods in a composite structure. In a later lecture, we shall see another way of expressing the behavior of a composite structure without having to add new methods and interpret them.

3.2.1.2 Code Template

Whenever we want the IList to perform a task, we add a method to IList and write appropriate concrete implementations in MTList and NEList. The following table illustrates the code template for writing the

concrete code in MTList and NEList.

Interpreter Pattern coding template for lists

interface IList	
public abstract returnType methodName(param list); // returnType may be 'void'	
MTList // no data	NEList Object _first; IList _rest;

public returnType methodName(param list) {
/*
This is in general the base case of a recursive call. Write the (non-recursive) code to solve the problem.
*/
}
	public returnType methodName(param list) {
	/*
	This is in general a recursive method. The code here can refer to _first and _rest, and all the methods in NEList
	When referencing _rest, one usually makes the recursive call:
	_rest.methodName(appropriate parameters).
	*/
	}

Table 3.4

3.2.1.2.1 In Class Exercises (Role-Acting)

- Find a number in a list and return "Found it!" if the number is in the list otherwise return "Not found!"
- Append a list B to a given list A and return a new list consisting of A and B concatenated together.

3.3 Recursion[3]

3.3.1 Recursive Data Structures

A recursive data structure is an object or class that contains an abstraction of itself.

In mathematical terms, we say that the object is "isomorphic" to itself. The basic embodiment of a recursive data structure is the Composite Design pattern[4] . Recursive data structures enable us to represent repetitive abstract patterns. In such, they enable us to generate or represent complexity from simplicity.

Characteristics of a recursive data structure:

- **Abstract representation** : Since the actual total structure of the data is not known until run-time, the data must be represented by an abstraction, such as an abstract class or interface.

[3]This content is available online at <http://legacy.cnx.org/content/m17306/1.4/>.
[4]http://www.exciton.cs.rice.edu/JavaResources/DesignPatterns/composite.htm

- **Base case(s)** : These represent the "end" of the pattern. They are the termination point(s) of the data structure.
- **Inductive case(s)** : These represent the on-going, "interior" portion of the repetitive pattern. They embody the ability to represent the data structure as a a simple connection between abstractly equivalent entities.

Recursive data structures are arguably the most important data structure in computer science as they are able to represent arbitrarily complex data. Indeed, if one looks across all the sciences, one sees that one of the fundamental modeling tools used is to attempt to

3.3.2 Recursive Algorithms

In order to process a recursive data structure, it makes sense that any such algorithm should reflect the recursive nature of the data structure:

> *A recursive algorithm is a process that accomplishes its task, in part, by calling an abstraction of itself*

Recursion is thus a special case of delegation.

In light of the above definition, it is not surprising that recursive algorithms and recursive data structures share common characteristics:

Characteristics of a recursive algorithm:

- **Abstract representation** : Since the actual total process needed to process the recursive datastructure of the data is not known until run-time, the algorithm must be represented by an abstraction, such as an abstract method (this is not the only way).
- **Base case(s)** : These represent the "end" of the algorithm. They are the termination point(s) of the algorithm.
- **Inductive case(s)** : These represent the on-going, "interior" portion of the algorithm. They embody the ability to process the recursive data structure by calling the same abstract process on the composed elements of the structure.

The similarity between recursive algorithms and recursive data structures is because in an OO system, **the structure drives the algorithm**. That is, it is the form of the data structure that determines the form if the algorithm. In an OO system, objects are asked to perform algorithms as they pertain to that object– that is, an algorithm on an object is a method of that object. **The data has the behavior. The data is intelligent.** This is in contrast to procedural or functional programming, where data is handed to the behavior. That is, stand-alone functions are used to process non-intelligent data. (Caveat: With all that said, in more advanced designs, we will show the algorithm can be decoupled from its data structure and thus be removed as a method of the data. This will not change the above principles however.)

The basic notions of creating a recursive algorithm on a composite design pattern structure are

- The abstract superclass or interface of the data structure has the invariant abstract behavior of being able to perform the algorithm (and thus the computations associated with it).
- Each concrete subclass has its own implementation of that abstract behavior, which is just the variant part of the algorithm that pertains to that particular subclass.

This is the Interpreter Design pattern.[5] Notice that no checks of the type of data being processed (e.g. base case or inductive case) are necessary. Each data object knows intrinsically what it is and thus what it should do. This is called "polymorphic dispatching" when an abstract method is called on an abstract data object, resulting in a particular concrete behavior corresponding to the concrete object used. In other words, we

[5] "Interpreter Design Pettern" <http://legacy.cnx.org/content/m16877/latest/>

call a method on a list, but get the behavior of an empty list if that what the list is, or we get the behavior of a non-empty list if that is what the list is.

In order to **prove** that a recursive algorithm will eventually complete, one must show that every time the recursive call is made, the "problem" is getting "smaller". The "problem" is usually the set of possible objects that the recursive call could be called upon. For instance, when recursively processing a list, every call to the rest of the list is calling on a list that is getting progessively shorter. At times, one cannot prove that the problem is definitely getting smaller. This does not mean that the algorithm will never end, it just means that there is a non-zero probability that it will go on forever.

One of the key aspects of a recursive algorithm is that in the inductive case, the inductive method makes the recursive call to another object's method. But in doing so, it has to wait for the called method to return with the needed result. This method that is waiting for the recursive call to return is called a "**pending operation**". For instance, at the time the empty list (base case) is reached during a recursive algorithm on a list, every non-empty node in that list has a pending operation.

Example 3.1: Animated Recursion Demo
Below is an example of generally what is happening in four linked objects during the call to the recursive method of the first object:

This media object is a Flash object. Please view or download it at
<recursion_demo.swf>

3.3.3 Tail Recursion

Consider the problem of finding the last element in a list. Again we need to interpret what it means to be the last element of (a) the empty list and (b) a non-empty list.

- Last element of the empty list: the empty list has no element; so there is no such thing as the last element for the empty list. How do we represent the non-existence of an object? For now, we use a key word in Java called `null`. `null` is a special value in Java that can be assigned to any variable of `Object` type to signify that the variable is not referencing any `Object` at all.
- Last element of a non-empty list with first and rest: it all depends on rest! rest has the capability to tell whether or not first is the last element of the current list. When rest is empty, then first is the last element. When rest is not empty, the first is certainly not the last element. rest has its own fist in this case, and it's up to the rest of rest to determine whether or not this new first is the last element! It's recursion again, isn't it?

To recapitulate, here is how a list can find its own last element.

- empty list case: return null or throw an exception to signify there is no such element.
- non-empty list case: pass first to rest and ask rest for help to find the last element.

How does rest use the first element of the enclosing list to help find the last element of the enclosing list?

- empty list case: the first element of the enclosing list is the last element.
- non-empty list case: recur! Pass its own first to its rest to help find the last element.

Here is the code.

Top-level abstract list's abstract method definition

```
/**
 * Represents the abstract list structure.
 */
public interface IList {
    /**
     * Returns the last element in this IList.
     */
    Object getLast();


    /**
     * Given the first of the preceding list, returns the last element of the preceding list.
     * @param acc the first of the preceding list.
     */
    Object getLastHelp(Object acc);

}
```

Table 3.5

Concrete list's concrete method implementations

```
/**
 * Represents empty lists.
 */
public class MTList implements IList {
    // Singleton Pattern
    public static final MTList Singleton
                          = new MTList();
    private MTList() {
    }

    /**
     * Returns null to signify there is
     * no last element in the empty list.
     */
    public Object getLast() {
        return null;
    }

    /**
     * Returns acc, because being the
     * first element of the preceding
     * list, it is the last element.
     */
    public Object getLastHelp(
                             Object acc) {
        return acc;
    }
}
```

```
/**
 * Represents non-empty lists.
 */
public class NEList implements IList {

   private Object _first;
   private IList _rest;

   public NEList(Object f, IList r) {
       _first = f;
       _rest = r;
   }

    /**
     * Passes first to rest and asks for
     * help to find the last element.
     */
    public Object getLast() {
        return _rest.getLastHelp(_first);
    }

    /**
     * Passes first to rest and asks for
     * help to find the last element.
     */
    public Object getLastHelp(Object acc) {
        return _rest.getLastHelp(_first);
    }
}
```

Table 3.6

The above algorithm to compute the last element of a list is another example of forward accumulation. Note that in the above, `getLast` is not recursive while `getLastHelp` is recursive. Also note that for the `NEList`, the last computation in `getLastHelp` is a recursive call to `getLastHelp` on `_rest`. There is no other computation after the recursive call returns. This kind of recursion is called **tail recursion**. Tail recursion is important for program performance. A smart compiler can recognize tail recursion and generate code that speeds up the computation by bypassing unnecessary setup code each time a recursive call is made.

.

3.4 Visitor Design Pattern[6]

3.4.1 1. Decoupling Algorithms from Data Structures

Recall the current formulation of the immutable list structure using the composite pattern.

[6]This content is available online at <http://legacy.cnx.org/content/m16707/1.3/>.

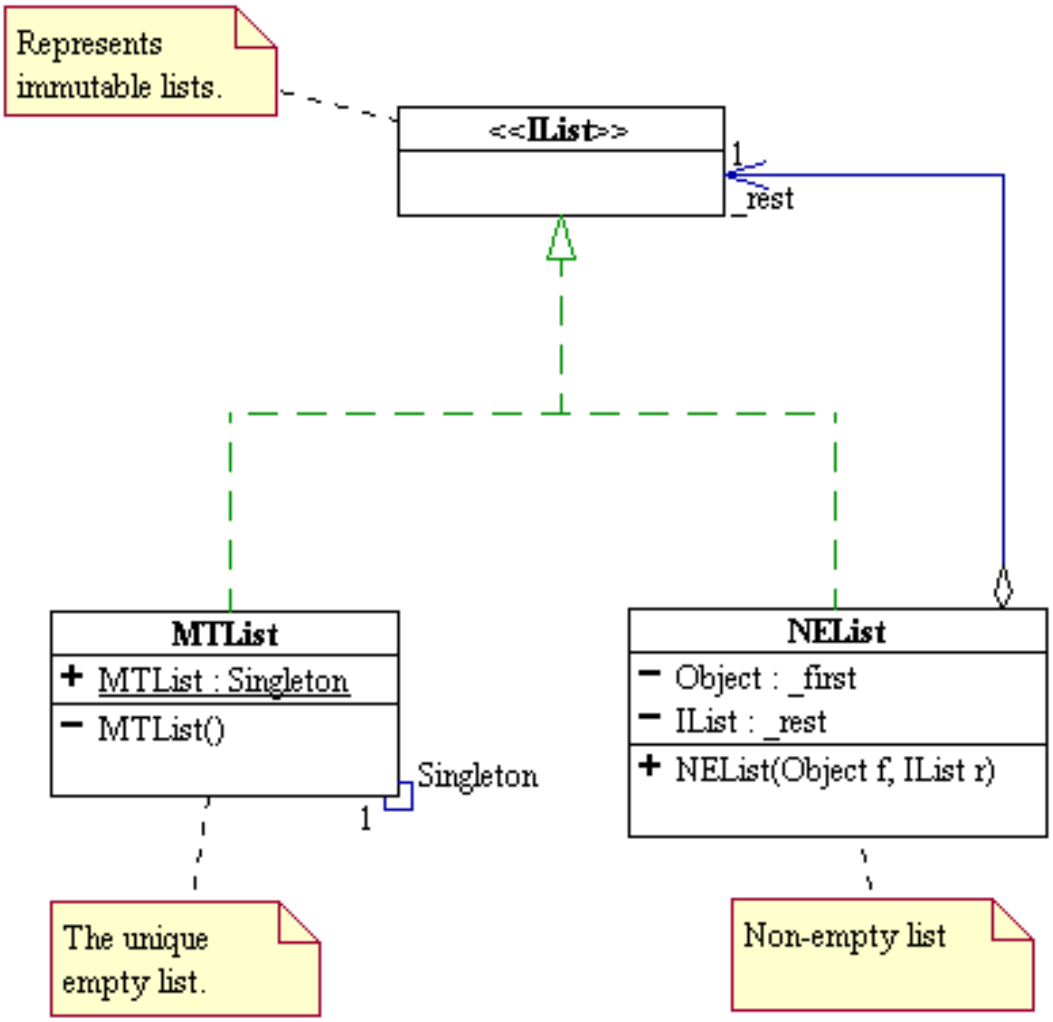


Figure 3.2

Each time we want to compute something new, we apply the interpreter pattern add appropriate methods to IList and implement those methods in MTList and NEList. This process of extending the capability of the list structure is error-prone at best and cannot be carried out if one does not own the source code for this structure. Any method added to the system can access the private fields of MTList and NEList and modify them at will. In particular, the code can change _fist and _rest of NEList breaking the invariant immutable property the system is supposed to represent. The system so designed is inherently fragile, cumbersome, rigid, and limited. We end up with a forever changing IList that includes a multitude of unrelated methods.

These design flaws come of the lack of delineation between the intrinsic and primitive behavior of the

structure itself and the more complex behavior needed for a specific application. The failure to decouple primitive and non-primitive operations also causes reusability and extensibility problems. The weakness in bundling a data structure with a predefined set of operations is that it presents a static non-extensible interface to the client that cannot handle unforeseen future requirements. Reusability and extensibility are more than just aesthetic issues; in the real world, they are driven by powerful practical and economic considerations. Computer science students should be conditioned to design code with the knowledge that it will be modified many times. In particular is the need for the ability to add features after the software has been delivered. Therefore one must seek to decouple the data structures from the algorithms (or operations) that manipulate it. Before we present an object-oriented approach to address this issue, let's first eat!

3.4.2 2. To Cook or Not To Cook

Mary is a vegetarian. She only cooks and eats vegetarian food. John is carnivorous. He cooks and eats meat! If Mary wants to eat broccoli and cheese, she can learn how to cook broccoli and cheese. If she wants corn of the cob, she can learn how to cook corn on the cob. The same goes for John. If he wants to eat greasy hamburger, he can learn how to cook greasy hamburger. If he wants to eat fatty hotdog, he can learn how to cook fatty hotdog. Every time John and Mary want to eat something new, they can learn how to cook it. This requires that John and Mary to each have a very big head in order to learn all the recipes.

But wait, there are people out there called chefs! These are very special kinds of chefs catering only to vegetarians and carnivores. These chefs only know how to cook two dishes: one vegetarian dish and one meat dish. All John and Mary have to do is to know how to ask such a chef to cook their favorite dish. Mary will only order the vegetarian dish, while John will only order the meat dish!

How do we model the vegetarian, the carnivore, the chef, the two kinds of dishes the chef cooks, and how the customer orders the appropriate kind of dish from the chef?

3.4.2.1 The Food

To simplify the problem, let's treat food as String. (In a more sophisticated setting, we may want to model food as some interface with veggie and meat as sub-interface.)

3.4.2.2 The Food Consumers

Vegetarians and carnivores are basically the same animals. They have the basic ingredients such as salt and pepper to cook food. They differ in the kind of raw materials they stock to cook their foods and in the way they order food from a chef. Vegetarians and Carnivores can provide the materials to cook but do not know how to cook! In order to get any cooked meal, they have to ask a chef to cook for them. We model them as two concrete subclasses of an *abstract class* called `AEater`. `AEater` has two concrete methods, `getSalt` and `getPepper`, and an *abstract* method called `order`, as shown in the table below.

Top-level abstract definition

```
public abstract class AEater {
    public String getSalt() {
        return "salt";
    }
    public String getPepper() {
        return "pepper";
    }
    /**
     * Orders n portions of appropriate food from restaurant r.
     */
    public abstract String order(IChef r, Integer n);
    // NO CODE BODY!
}
```

Table 3.7

Concrete implementations

```
public class Vegetarian extends AEater{
    public String getBroccoli() {
        return "broccoli";
    }
    public String getCorn() {
        return "corn";
    }
    public String order(IChef c, Object n) {
        // code to be discussed later;
    }
}
```

```
public class Carnivore extends AEater{
    public String getMeat() {
        return "steak";
    }
    public String getChicken() {
        return "cornish hen";
    }
    public String getDog() {
        return "polish sausage";
    }
    public String order(IChef c, Object n) {
    // code to be discussed later;
    }
}
```

Table 3.8

3.4.2.3 The Chef

The chef is represented as an interface IChef with two methods, one to cook a vegetarian dish and one to cook a meat dish, as shown in the table below.

Top-level abstract definition

```
interface IChef  {
    String cookVeggie(Vegetarian h, Integer n);
    String cookMeat(Carnivore h, Integer n);
}
```

Table 3.9

Concrete implementations

```
public class ChefWong implements IChef {

    public static final ChefWong Singleton
        = new ChefWong();

    private ChefWong() {}

    public String cookVeggie(
        Vegetarian h, Integer n) {

        return  n + " portion(s) of " +
                h.getCarrot() + ", " +
                h.getSalt();
    }

    public String cookMeat(
        Carnivore h, Integer n) {

        return  n + " portion(s) of " +
                h.getMeat() + ", " +
                h.getPepper();
    }
}
```

```
public class ChefZung implements IChef {

    public static final ChefZung Singleton
        = new ChefZung();

    private ChefZung() {}

    public String cookVeggie(
        Vegetarian h, Integer n) {

        return  n + " portion(s) of " +
                h.getCorn() + ", " +
                h.getSalt();
    }

    public String cookMeat(
        Carnivore h, Integer n) {

        return  n + " portion(s) of " +
                h.getChicken() + ", " +
                h.getPepper() +
                ", " + h.getSalt();
    }
}
```

Table 3.10

3.4.2.4 Ordering Food From The Chef

To order food from an IChef , a Vegetarian object simply calls cookVeggie, passing itself as one of the parameters, while a Carnivore object would call cookMeat, passing itself as one of the parameters as well. The Vegetarian and Carnivore objects only deal with the IChef object at the highest level of abstraction and do not care what the concrete IChef is. The polymorphism machinery guarantees that the correct method in the concrete IChef will be called and the appropriate kind of food will be returned to the AEater caller The table below shows the code for Vegetarian and Carnivore, and sample client code using these classes.

Concrete implementations

```
public class Vegetarian extends AEater {
    // other methods elided
    public String order(IChef c, int n) {
        return c.cookVeggie(this, n);
    }
}
```

```
public class Carnivore extends AEater {
    // other methods elided
    public String order(IChef c, int n) {
        return c.cookMeat(this, n);
    }
}
```

Table 3.11

Client code

```
public void party(AEater e, IChef c, int n) {
    System.out.println(e.order(c, n));
}

    // blah blah blah...
    AEater John = new Carnivore();
    AEater Mary = new Vegetarian();
    party(Mary, ChefWong.Singleton, 2);
    party(John,ChefZung.Singleton, 1);
```

Table 3.12

The above design is an example of what is called the visitor pattern.

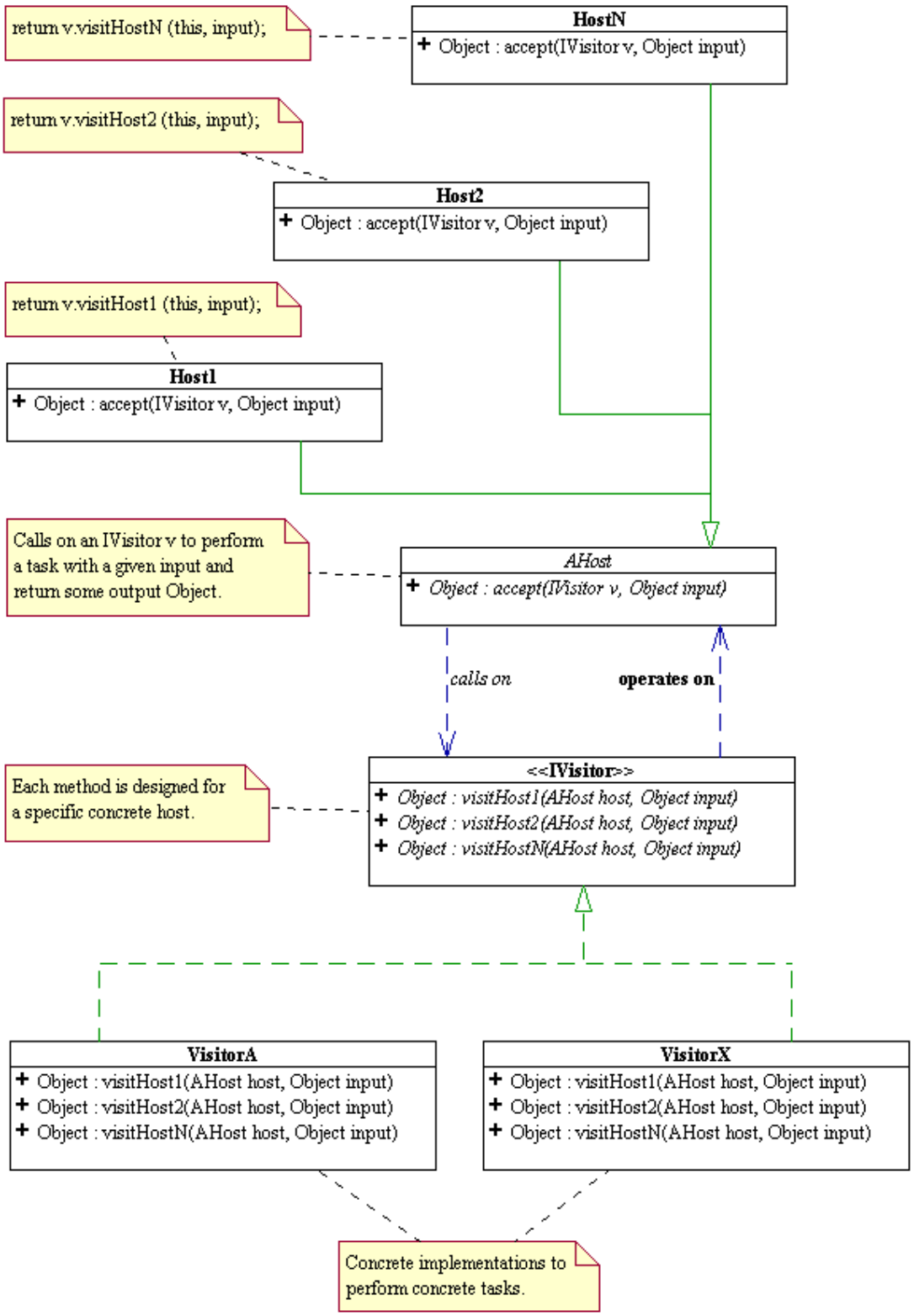
- The abstract class AEater and its concrete subclasses are called the hosts. The method public String order(IChef c, Object n) is called the hook method. Each concrete subclasses of AEater knows exactly to call the appropriate method on the IChef parameter, but does know and need not how the IChef concretely performs its task. This allows an open-ended number of ways to cook the appropriate kinds of food.
- The chef interface IChef and all of its concrete implementations are called visitors. When an IChef performs cookMeat/cookVeggie, it knows that its host is a Carnivore/Vegetarian and can only call the methods of Carnivore/Vegetarian to cook a dish. Java static type checking will flag an error should there be a call on the host to getCarot in the method cookMeat. This is makes the interaction between hosts (Vegetarian and Carnivore) and visitors (IChef and all of its concrete implementations) much more robust.

3.4.3 3. The Visitor Pattern

The visitor pattern[7] is a pattern for communication and collaboration between two union patterns: a "host" union and a "visitor" union. An abstract visitor is usually defined as an interface in Java. It has a separate method for each of the concrete variant of the host union. The abstract host has a method (called the "hook") to "accept" a visitor and leaves it up to each of its concrete variants to call the appropriate

[7] http://cnx.org/content/m26430/latest/

visitor method. This "decoupling" of the host's structural behaviors from the extrinsic algorithms on the host permits the addition of infinitely many external algorithms without changing any of the host union code. This extensibility only works if the taxonomy of the host union is stable and does not change. If we have to modify the host union, then we will have to modify ALL visitors as well!



Available for free at Connexions <http://legacy.cnx.org/content/col10213/1.37>

Figure 3.3

NOTE: All the "state-less" visitors, that is visitors that contain no non-static fields should be singletons. Visitors that contain non-static fields should not be singletons.

3.4.4 4. Fundamental Object-Oriented Design Methodology (FOODM)

1. **Identify and separate the variant and the invariant behaviors.**
2. **Encapsulate the invariant behaviors into a system of classes.**
3. **Add "hooks" to this class to define communication protocols with other classes.**
4. **Encapsulate the variant behaviors into a union of classes that comply with the above protocols.**

The result is a flexible system of co-operating objects that is not only reusable and extensible, but also easy to understand and maintain.

Let us illustrate the above process by applying it to the design of the immutable list structure and its algorithms.

1. Here, the invariant is the intrinsic and primitive behavior of the list structure, `IList`, and the variants are the multitude of extrinsic and non-primitive algorithms that manipulate it, `IListAlgo`.
2. The recursive list structure is implemented using the composite design pattern and encapsulated with a minimal and complete set of primitive structural operations: `getFirst()` and `getRest()`.
3. The hook method `Object execute(IListAlgo ago, Object inp)` defines the protocols for operating on the list structure. The hook works as if a `IList` announces to the outside world the following protocol: *If you want me to execute your algorithm, encapsulate it into an object of type IListAlgo, hand it to me together with its inp object as parameters for my execute(). I will send your algorithm object the appropriate message for it to perform its task, and return you the result.*
   - `emptyCase(...)` should be the part of the algorithm that deals with the case where I am empty.
   - `nonEmptyCase(...)` should be the part of the algorithm that deals with the case where I am not empty."
4. `IListAlgo` and all of its concrete implementations forms a union of algorithms classes that can be sent to the list structure for execution.

Below is the UML class diagram of the resulting list design. Click here to see the full documentation.[8] Click here to see the code[9] .

[8] http://www.clear.rice.edu/comp201/08-spring/lectures/visitor/listFW/doc/
[9] http://www.clear.rice.edu/comp201/08-spring/lectures/lec17/listFW.zip

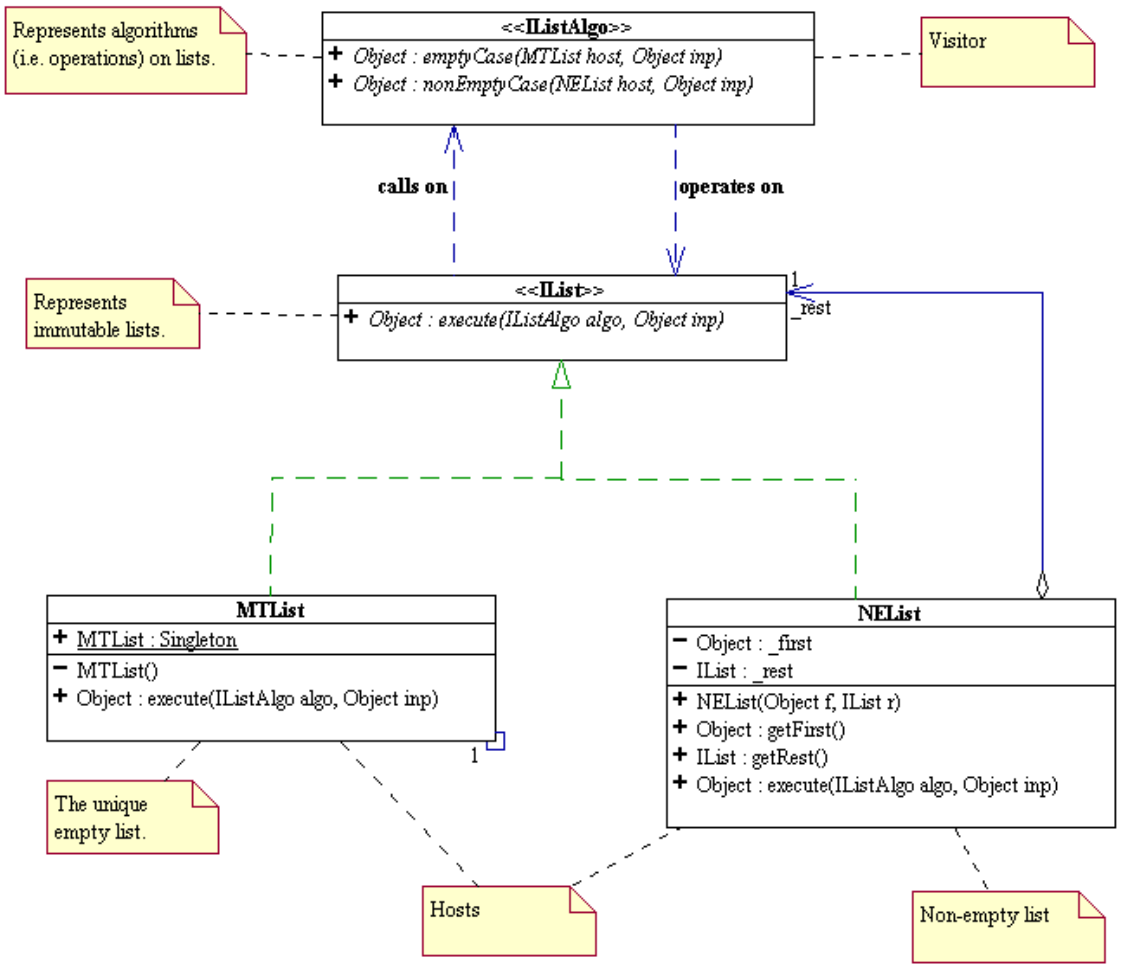


Figure 3.4

The above design is nothing but a special case of the **Visitor Pattern**. The interface `IList` is called the **host** and its method `execute()` is called a "**hook**" to the `IListAlgo`**visitors**. Via polymorphism, `IList` knows exactly what method to call on the specific `IListAlgo` visitor. This design turns the list structure into a (miniature) framework where control is inverted: one hands an algorithm to the structure to be executed instead of handing a structure to an algorithm to perform a computation. Since an `IListAlgo` only interacts with the list on which it operates via the list's public interface, the list structure is capable of carrying out any conforming algorithm, past, present, or future. This is how reusability and extensibility is achieved.

3.4.5 5. List Visitor Examples

Singleton constructor for list algorithm

```
/**
 * Computes the length of the IList host.
 */
public class GetLength implements IListAlgo {
    /**
     * Singleton Pattern.
     */
    public static final GetLength Singleton = new GetLength();
    private GetLength() {  }
```

Table 3.13

Empty and non-empty cases of algorithm

```
/**
 * Returns Integer(0).
 * @param nu not used
 * @return Integer(0)
 */
public Object emptyCase(
    MTList host, Object... nu) {

    return 0;
}
```

```
/**
 * Return the length of the host's rest
 * plus 1.
 * @param nu not used.
 * @return Integer > 0.
 */
public Object nonEmptyCase(
    NEList host, Object... nu) {

    Object restLen
        = host.getRest().execute(this);
    return 1 + (Integer)restLen);
}
```

Table 3.14

Singleton constructor for list algorithm

```
package listFW;
/**
 * Computes a String reprsentation of IList showing a left parenthesis followed
 * by elements of the IList separated by commas, ending with with a right parenthesis.
 */
public class ToStringAlgo implements IListAlgo {
    public static final ToStringAlgo Singleton = new ToStringAlgo();

    private ToStringAlgo() {  }
```

Table 3.15

Empty and non-empty cases of algorithm

```
/**
 * Returns "()".
 */
public Object emptyCase(
    MTList host, Object... inp) {

    return "()";
}
```

```
/**
 * Passes "(" + first to the rest of IList
 * and asks for help to complete the
 * computation.
 */
public Object nonEmptyCase(
    NEList host, Object... ~inp) {

    return host.getRest().execute(
        ToStringHelper.Singleton,
        "(" + host.getFirst());
}
} //closing parenthesis for the class
```

Table 3.16

Singleton constructor for helper visitor

```
/**
 * Helps ToStringAlgo compute the String representation of the rest of the list.
 */
class ToStringHelper implements IListAlgo {
    public static final ToStringHelper Singleton = new ToStringHelper();
    private ToStringHelper() {   }
```

Table 3.17

Empty and non-empty cases of algorithm

```
/**
 * Returns the accumulated String + ")".
 * At end of list: done!
 */
public Object emptyCase(
    MTList host, Object... acc) {

    return~ acc[0] + ")";
}
```

```
/**
 * Continues accumulating the String
 * representation by appending ", " + first
 * to acc and recurse!
 */
public Object nonEmptyCase(
    NEList host, Object... acc) {

    return host.getRest().execute(
        this,
        acc[0] + ", " + host.getFirst());
    }
}
```

Table 3.18

We now can use to ToStringAlgo to implement the toString() method of an IList.

```
package listFW;

public class MTList implements IList {

    /**
     * Singleton Pattern
     */
    public final static MTList Singleton
        = new MTList();

        private MTList() { }

        /**
         * Calls the empty case of the
         * algorithm algo, passing to it
         * itself as the host parameter
         * and the given input inp as the
         * input parameter.
         * This method is marked as final
         * to prevent all subclasses from
         * overriding it. Finalizing a
         * method also allows the compiler
         * to generate more efficient
         * calling code.
         */
         public final Object execute(
             IListAlgo algo,
             Object... inp) {

            return algo.emptyCase(
                this, inp);
         }
         public String toString() {
            return (String)ToStringAlgo
                .Singleton.emptyCase(this);
         }
    }
```

```
package listFW;

public class NEList implements IList {
    /**
     * The first data element.
     */
    private Object _first;
    /**
     * The rest or "tail" of this NEList.
     * Data Invariant: _rest != null;
     */
    private IList _rest;
    /**
     * Initializes this NEList to a given
     * first and a given rest.
     * @param f This NEList's first element
     * @param r !=null; rest of this NEList.
     */
    public NEList(Object f, IList r) {
        _first = f;
        _rest = r;
    }
    /**
     * Returns the first data element of
     * this NEList.
     * Marked final for the above reasons.
     */
    public final Object getFirst() {
        return _first;
    }
    /**
     * Returns the rest of this NEList which
     * is an IList.
     * Marked final for the above reasons.
     */
    public final IList getRest() {
       return _rest;
    }
    /**
     * Calls the nonEmptyCase method of the
     * IListAlgo parameter, passing itself
     * as the host parameter and the given
     * input as the input parameter.
     * Marked final for the above reasons.
     */
    public final Object execute(
        IListAlgo algo, Object... inp) {
        return algo.nonEmptyCase(this, inp);
    }
    public String toString() {
        return (String)ToStringAlgo
            .Singleton.nonEmptyCase(this);
    }
}
```

Table 3.19

Download the above code here[10] .

3.5 Abstract Factory Design Pattern[11]

3.5.1 1. Information Hiding

Information hiding is a tried-and-true design principle that advocates hiding all implementation details of software components from the user in order to facilitate code maintenance. It was first formulated by David L. Parnas (in 1971-1972) as follows.

- One must provide the intended user with all the information needed to use the module correctly **and nothing more**.
  - translation to OOP: the user should not know anything about how an interface or abstract class is implemented. For example, the user need not and should not know how `IList` is implemented in order to use it. The user should only program to the abstract specification.
- One must provide the implementor with all the information needed to complete the module and nothing more.
  - translation to OOP: the implementor of a class or an interface should not know anything about how it will be used. For example, the implementor need not and should not know how, when, or where `IList` will be used. The implementor should write the implementation code based solely on the abstract specification.

By adhering to the above, code written by both users and implementors will have a high degree of flexibility, extensibility, interoperability and interchangeability.

The list framework that we have developed so far has failed to hide `MTList` and `NEList`, which are concrete implementations of `IList`, the abstract specification of the list structure. In many of the list algorithms that we have developed so far, we need to call on `MTList.Singleton` or the constructor of `NEList` to instantiate concrete `IList` objects. The following is another such examples.

[10] http://www.owlnet.rice.edu/~comp201/08-spring/lectures/lec17/listFW.zip
[11] This content is available online at <http://legacy.cnx.org/content/m16796/1.3/>.

InsertInOrder.java

```
import listFW.*;

/**
 * Inserts an Integer into an ordered host list, assuming the host list contains
 * only Integer objects.
 */
public class InsertInOrder implements IListAlgo {

    public static final InsertInOrder Singleton = new InsertInOrder();
    private InsertInOrder() {
    }

    /**
     * This is easy, don't you think?
     * @param inp inp[0] is an Integer to be inserted in order into host.
     */
    public Object emptyCase(MTList host, Object... inp) {
        return new NEList(inp[0], host);
    }

    /**
     * Delegate (to recur)!
     * @param inp inp[0] is an Integer to be inserted in order into host.
     */
    public Object nonEmptyCase(NEList host, Object... inp) {
        int n = (Integer)inp[0];
        int f = (Integer)host.getFirst();
        return n < f ?
                new NEList(inp[0], host):
                new NEList(host.getFirst(), (IList)host.getRest().execute(this, inp[0]));
    }
}
```

Table 3.20

The above algorithm to insert in order an integer into an ordered list of integers can only be used for a very specific implementation of `IList`, namely the one that has `MTList` and `NEList` as concrete subclasses. How can we write list algorithms that can be used for ANY implementation of the abstract specification of the list structure represented by the abstract class `IList`?

We can achieve our goal by

1. abstracting the behavior of `MTList` and `NEList` into interfaces with pure abstract structural behaviors.
2. applying the Abstract Factory Design Pattern[12] to hide the concrete implementation from the user.

[12] http://cnx.org/content/m17203/latest/

3.5.2 2. Abstract List Behaviors

The concrete empty list and non-empty list implemented as `MTList` and `NEList` are now expressed as interfaces as follow.

Top-level abstract list definition

```
package listFW;

/**
 * Represents the abstract behavior of the immutable list structure.
 * A list intrinsically "knows" how to execute an algorithm on itself.
 */
interface IList {
    Object execute(IListAlgo algo, Object... inp);
}
```

Table 3.21

Abstract empty and non-empty list definitions

```
package listFW;

/**
 * Represents the immutable empty list.
 * The empty list has no well-defined
 * structural behavior: it has no first
 * and no rest.
 */
interface IMTList extends IList {
}
```

```
package listFW;

/**
 * Represents the immutable non-empty list.
 * An immutable non-empty list has a data
 * object called first, and an isomorphic
 * subcomponent called rest.  Its structural
 * behavior provides access to its internal
 * data (first) and substructure (rest).
 */
interface INEList extends IList {
    /**
     * "Gettor" method for the list's first.
     * @return this INElist's first element.
     */
    Object getFirst();

    /**
     * "Gettor" method for the list's rest.
     * @return this INElist's rest.
     */
    IList getRest();
}
```

Table 3.22

3.5.3 3. Abstract List Factory

Before we describe in general what the **Abstract Factory Pattern** is, let's examine what we have to do in the case of `IList`.

- Define an abstract factory interface, `IListFactory`, to manufacture empty and non-empty `IList` objects. Put `IList`, `IMTList`, `INEList`, `IListVistor`, and `IListFactory` in the same package. `IListFactory` is specified as followed.

`IListFactory.java`

```
package listFW;

/**
 * Abstract factory to manufacture IMTList and INEList.
 */
interface IListFactory {
    /**
     * Creates an empty list.
     * @return an IMTList object.
     */
    IMTList makeEmptyList();

    /**
     * Creates a non-empty list containing a given first and a given rest.
     * @param first a data object.
     * @param rest != null, the rest of the non-empty list to be manufactured.
     * @return an INEList object containing first and rest
     * @exception IllegalArgumentException if rest is null.
     */
    INEList makeNEList(Object first, IList rest);
}
```

Table 3.23

`IList`, `IListAlgo`, and `IListFactory` prescribe a **minimal** and **complete** abstract specification of what we call a list **software component**. We claim without proof that we can do everything we ever want to do with the list structure using this specification.

- All algorithms (i.e. visitors) that call for the creation of concrete `IList` objects will need to have an abstract factory as a parameter and use it to manufacture `IList` objects. We usually pass the factory as an argument to the constructor of the visitor. The visitor is thus not a singleton.

InsertInOrderWithFactory.java

```
import listFW.*;

/**
 * Inserts an Integer into an ordered host list, assuming the host list contains
 * only Integer objects.  Has no knowledge of how IList is implemented.  Must
 * make use of a list factory (IListFactory) to create IList objects instead of
 * calling the constructors of concrete subclasses directly.
 */
public class InsertInOrderWithFactory implements IListAlgo {

    private IListFactory _listFact;

    public InsertInOrderWithFactory(IListFactory lf) {
        _listFact = lf;
    }

    /**
     * Simply makes a new non-empty list with the given inp parameter as first.
     * @param host an empty IList.
     * @param inp inp[0] is an Integer to be inserted in order into host.
     */
    public Object emptyCase(IMTList host, Object... inp) {
        return _listFact.makeNEList(inp[0], host);
    }

    /**
     * Recur!
     * @param host a non-empty IList.
     * @param inp inp[0] is an Integer to be inserted in order into host.
     */
    public Object nonEmptyCase(INEList host, Object... inp) {
        int n = (Integer)inp[0];
        int f = (Integer)host.getFirst();
        return n < f ?
                _listFact.makeNEList(inp[0], host):
                _listFact.makeNEList(host.getFirst(),
                                     (IList)host.getRest().execute(this, inp[0]));
    }
}
```

Table 3.24

The above algorithm only "talks" to the list structure it operates on at the highest level of abstraction specified by `IList` and `IListFactory`. It does know and does not care how `IList` and `IListFactory` are implemented. Yet it can be proved to be correct. This algorithm can be plugged into any system that subscribes to the abstract specification prescribed by `IList`, `IListAlgo`, and `IListFactory`.

- Provide a concrete implementation of the abstract factory that contains all concrete subclasses of `IList` as **private static** classes and thus hide them from all external code.

Concrete factory for instnatiating composite lists

CompositeListFactory.java

```
package listFW.factory;

import listFW.*;

/**
 * Manufactures concrete IMTList and INEList objects.  Has only one
 * instance referenced by CompositeListFactory.Singleton.
 * MTList and NEList are static nested classes and hidden from all external
 * client code.  The implementations for MTList and NEList are the same as
 * before but completely invisible to the outside of this factory.
 */
public class CompositeListFactory implements IListFactory {

    /**
     * Note the use of private static.
     */
    private static class MTList implements IMTList {
        public final static MTList Singleton = new MTList ();
        private MTList() {
        }

        final public Object execute(IListAlgo algo, Object... inp) {
            return algo.emptyCase(this, inp);
        }

        public String toString() {
            return "()";
        }
    }
```

continued on next page

```
/**
 * Note the use of private static.
 */
private static class NEList implements INEList {
    private Object _first;
    private IList _rest;

    public NEList(Object dat, IList rest) {
        _first = dat;
        _rest = rest;
    }

    final public Object getFirst() {
        return _first;
    }

    final public IList getRest() {
        return _rest;
    }

    final public Object execute(IListAlgo algo, Object... inp) {
        return algo.nonEmptyCase(this, inp);
    }


    public String toString() {
        return (String)ToStringAlgo.Singleton.nonEmptyCase(this);
    }
}
```

continued on next page

```
    /**
     * Singleton Pattern
     */
    public static final CompositeListFactory Singleton = new CompositeListFactory();
    private CompositeListFactory() {
    }

    /**
     * Creates an empty list.
     * @return an IMTList object.
     */
    public IMTList makeEmptyList() {
        return MTList.Singleton;
    }

    /**
     * Creates a non-empty list containing a given first and a given rest.
     * @param first a data object.
     * @param rest != null, the rest of the non-empty list to be manufactured.
     * @return an INEList object containing first and rest
     */
    public INEList makeNEList(Object first, IList rest) {
        return new NEList(first, rest);
    }
}
```

Table 3.25

- Pass such a concrete factory to all client code that need to make use of the abstract factory to manufacture concrete `IList` instances.

Below is an example of a unit test for the `InsertInOrderWithFactory` algorithm.

Test_InsertInOrderWithFactory.java

```
package listFW.visitor.test;
import listFW.*;
import listFW.factory.*;
import listFW.visitor.*;

import junit.framework.TestCase;

/**
 * A JUnit test case class.
 * Every method starting with the word "test" will be called when running
 * the test with JUnit.
 */
public class Test_InsertInOrderWithFactory extends TestCase {


  public void test_ordered_insert() {
    IListFactory fac = CompositeListFactory.Singleton;
    IListAlgo algo = new InsertInOrderWithFactory(fac);

    IList list0 = fac.makeEmptyList();
    assertEquals("Empty list", "()", list0.toString());

    IList list1 = (IList) list0.execute(algo, 55);
    assertEquals("55->()", "(55)", list1.toString());

    IList list2 = (IList) list1.execute(algo, 30);
    assertEquals("30->(55)", "(30, 55)", list2.toString());

    IList list3 = (IList) list2.execute(algo, 100);
    assertEquals("100 -> (30, 55)", "(30, 55, 100)", list3.toString());

    IList list4 = (IList) list3.execute(algo, 45);
    assertEquals("45->(30, 55, 100)", "(30, 45, 55, 100)", list4.toString());

    IList list5 = (IList) list4.execute(algo, 60);
    assertEquals("60 -> (30, 45, 55, 100)", "(30, 45, 55, 60, 100)", list5.toString());
  }
}
```

Table 3.26

The above design process is an example of what is called the **Abstract Factory Design Pattern**. The intent of this pattern is to provide an abstract specification for manufacturing a family of related objects (for examples, the empty and non-empty `IList`) without specifying their actual concrete classes thus hiding all details of implementation from the user.

Our example of the list structure framework successfully delineates specification from implementation and faithfully adheres to the principle of information hiding.

- `IList`, `IMTList`, `INEList`, `IListAlgo`, and `IListFactory` provide a minimal and complete abstract specification.
- `InsertInOrderWithFactory` is a concrete implementation of `IListAlgo` that performs a concrete operation on the host list. Yet this algorithm need only communicate with the list structure and the list factory via their public interface. It will work with any implementation of `IList` and `IListFactory`.
- `CompositeListFactory` is a concrete implementation of `IListFactory`. It uses the composite pattern and the visitor pattern to implement `IList`. It only communicates with `IListAlgo` at and knows nothing about any of its concrete implementation. The **private static** attributes provide the proper mechanism to hide all implementation details from all code external to the class.

Click here to access the complete javadoc documentation and UML class diagram of the list component[13] described in the above.

Click here to download the complete source code and documentation of the list component[14] described in the above.

3.5.4 4. Frameworks

The following is a direct quote from the **Design Patterns** book by Gamma, Helm, Johnson, and Vlissides (the Gang of Four - GoF).

"*Frameworks thus emphasizes design reuse over code resuse...Reuse on this level leads to an inversion of control between the application and the software on which it's based. When you use a toolkit (or a conventional subroutine library software for that matter), you write the main body of the application and call the code you want to reuse. When you use a framework, you reuse the main body and write the code it calls....*"

The linear recursive structure (`IList`) coupled with the visitors as shown in the above is one of the simplest, non-trivial, and practical examples of frameworks. It has the characteristic of "inversion of control" described in the quote. It illustrates the so-called Hollywood Programming Principle: Don't call me, I will call you. Imagine the `IList` union sitting in a library.

The above list framework dictates the design of all algorithms operating on the list structure:

- All algorithms must be some concrete implementation of `IListAlgo`.
  - Algorithms that require the construction of empty and/or non-empty lists, must do so via some abstract list factory, `IListFactory`.
- In order to apply an algorithm to a list, one must ask a list to "execute" that algorithm, giving it the required input parameter.

When we write an algorithm on an `IList` in conformance with its visitor interface, we are writing code for the `IList` to call and not the other way around. By adhering to the `IList` framework's protocol, all algorithms on the `IList` can be developed much more quickly. And because they all have similar structures, they are much easier to "maintain". The `IList` framework puts polymorphism to use in a very effective (and elegant!) way to reduce flow control and code complexity.

We do not know anything about how the above list framework is implemented, yet we have been able to write quite a few algorithms and test them for correctness. In order to obtain concrete lists and test an algorithm, we call on a concrete `IListFactory`, called `CompositeListFactory`, to manufacture empty and non-empty lists. We do not know how this factory creates those list objects, but we trust that it does the right thing and produces the appropriate list objects for us to use. And so far, it seems like it's doing its job, giving us all the lists we need.

[13] http://www.clear.rice.edu/comp201/08-spring/lectures/visitor/listFW/doc/
[14] http://www.clear.rice.edu/comp201/08-spring/lectures/lec17/listFW.zip

3.5.5 5. Bootstrapping Along

Let's take a look back at what we've done with a list so far:

1. Created an invariant list interface with two variant concrete subclasses (Composite pattern) where any algorithms on the list where implemented as methods of the interface and subclasses (Interpreter pattern)
2. Extracted the variant algorithms as visitors leaving behind an invariant "execute" method. Accessor methods for first and rest installed. The entire list structure now becomes invariant.
3. Abstracted the creation of a list into an invariant factory interface with variant concrete subclass factories.
4. Separated the list framework into an invariant hierarchy of interfaces and a variant implementation which was hidden inside of a variant factory class.

Is there something systematic going on here?

Notice that at every stage in our development of our current list framework, we have applied the **same** abstraction principles to the then current system to advance it to the next stage. Specifically, we have identified and separated the variant and invariant pieces of the system and defined abstract representations whenever needed.

This really tells us about some general characteristics of software development:

- Software development is an iterative process. You never get the right answer the first time and you have to slowly "evolve" your code closer and closer to what you want.
- Every time you change your code you learn something more and/or new about your system and/or problem. This is because every new formulation of your solution represents a new way, a new view as it were, on the problem and this new view highlights aspects you hadn't considered before.
- The revision process is driven along by a repetitive application of the abstract decomposition techniques such as separating the variant and invariant.

Are we done with our list refinements? Will we ever be "done"? What do the above characteristics say about the way we should approach software development?

Also, now that we have managed to abstract structure, behavior and construction, is there anything left to abstract? Or is there one more piece to our abstraction puzzle (at least)?

3.6 Inner Classes[15]

3.6.1 1. Helpers are the Variants

Consider again the familiar problem of computing a `String` representation of an `IList` that displays a comma-separated list of its elements delimited by a pair of matching parentheses. We have seen at least one way to compute such a `String` representation using a visitor called `ToStringAlgo`. Below are three different algorithms, `ToString1`, `ToString2` and `ToString3`, that compute the same `String` representation.

[15] This content is available online at <http://legacy.cnx.org/content/m17220/1.4/>.

Main Visitor `ToString1`	Tail-recursive helper `ToString1Help`

`package listFW.visitor;`
`import listFW.*;`
`public class ToString1 implements IListAlgo`
`{`
`  public static final ToString1`
`  Singleton = new ToString1();`
 	
`  private ToString1() {}`
 	
`  public Object emptyCase(`
`  IMTList host, Object... nu) {`
`  return "()";`
`  }`
 	
`  public Object nonEmptyCase(`
`  INEList host, Object... nu) {`
`  return host.getRest().execute(`
`  ToString1Help.Singleton,`
`  "(" + host.getFirst());`
`  }`
`}`
	`/**`
	` * Helps ToString1 compute the String`
	` * representation of the rest of the list.`
	` * It takes as input the accumulated`
	` * string representation of the preceding`
	` * list. This accumulated string contains`
	` * the left most parenthesis and all the`
	` * elements of the preceding list, each`
	` * separated by a comma.`
	` */`
	`class ToString1Help implements IListAlgo {`
	`  public static final ToString1Help`
	`  Singleton = new ToString1Help();`
	
	`  private ToString1Help() {}`
	
	`  public Object emptyCase(`
	`  IMTList host, Object... acc) {`
	`  return acc[0] + ")";`
	`  }`
	
	`  public Object nonEmptyCase(`
	`  INEList host, Object... acc) {`
	`  return host.getRest().execute(this,`
	`  acc[0] + ", "`
	`  + host.getFirst());`
	`  }`
	`}`

Table 3.27

Main Visitor `ToString2`

```
package listFW.visitor;
import listFW.*;

public class ToString2 implements IListAlgo
{
    public static final ToString2
        Singleton = new ToString2();

    private ToString2() {}

    public Object emptyCase(IMTList host,
                            Object... nu) {
        return "()";
    }

    public Object nonEmptyCase(
        INEList host, Object... nu) {

        return host.getRest().execute(
            ToString2Help.Singleton,
            host.getFirst().toString());
    }
}
```

Tail-recursive helper `ToString2Help`

```
/**
 * Helps ToString2 compute the String
 representation of the rest of the list.
 */
class ToString2Help implements IListAlgo {
    public static final ToString2Help
        Singleton = new ToString2Help();

    private ToString2Help() {}

    public Object emptyCase(
        IMTList host, Object... acc) {

        return "(" + acc[0] + ")";
    }

    public Object nonEmptyCase(
        INEList host, Object... acc) {

        return host.getRest().execute(this,
            acc[0] + ", "
            + host.getFirst());
    }
}
```

Table 3.28

Main Visitor `ToString3`	Non tail-recursive helper `ToString3Help`

`package listFW.visitor;`
`import listFW.*;`
 	
`public class ToString3 implements IListAlgo`
`{`
`  public static final ToString3`
`  Singleton = new ToString3();`
 	
`  private ToString3() {}`
 	
`  public Object emptyCase(`
`  IMTList host, Object... nu) {`
 	
`  return "()";`
`  }`
 	
`  public Object nonEmptyCase(`
`  INEList host, Object... nu) {`
`  return "(" + host.getFirst()`
`  + host.getRest().execute(`
`  ToString3Help.Singleton);`
`  }`
`}`
	`/**`
	` * Helps ToString3 compute the String`
	` * representation of the rest of the list.`
	` */`
	`class ToString3Help implements IListAlgo {`
	`  public static final ToString3Help`
	`  Singleton = new ToString3Help();`
	
	`  private ToString3Help() {}`
	
	`  public Object emptyCase(`
	`  IMTList host, Object... nu) {`
	
	`  return ")";`
	`  }`
	
	`  public Object nonEmptyCase(`
	`  INEList host, Object... nu) {`
	`  return ", " + host.getFirst() +`
	`  host.getRest().execute(this);`
	`  }`
	`}`

Table 3.29

What makes each of the above different from one another is its helper visitor. Each helper defined in the above will only perform correctly if it is passed the appropriate parameter. In a sense, each helper is an implementation of the main visitor. We should hide each of them inside of its corresponding main visitor to ensure proper usage and achieve full encapsulation of the main visitor.

3.6.2 2. Hiding Helpers

The most secure way to hide the helper visitor is to move it inside of the main visitor and make it a `private static` class.

Hiding Named Helper Visitor inside of `ToString1`	Comments

`package listFW.visitor;`
`import listFW.*;`
 	
`public class ToString1WithHiddenHelper`
`  implements IListAlgo {`
 	
`  public static final ToString1WithHiddenHelper`
`  Singleton = new ToString1WithHiddenHelper();`
 	
`  private ToString1WithHiddenHelper() {`
`  }`
	Singleton Pattern

`  private static class HiddenHelper`
`  implements IListAlgo {`
`  public static final HiddenHelper`
`  Singleton = new HiddenHelper();`
 	
`  private HiddenHelper() {}`
 	
`  public Object emptyCase(IMTList host,`
`  Object... acc) {`
`  return acc[0] + ")";`
`  }`
 	
`  public Object nonEmptyCase(INEList host,`
`  Object... acc) {`
`  return host.getRest().execute(this,`
`  acc[0] + ", " + host.getFirst());`
`  }`
`  }`
	The helper visitor has a name, `HiddenHelper`, and is defined privately and globally (`static`) inside of the main visitor `ToString1WithHiddenHelper`.

continued on next page

```
    public Object emptyCase(IMTList host,
                            Object... nu) {
        return "()";
    }

    public Object nonEmptyCase(INEList host,
                               Object... nu) {
        return host.getRest().execute(
            HiddenHelper.Singleton,
            "(" + host.getFirst());
    }
}
```

The main visitor calls on its hidden helper singleton to help complete the job.

Table 3.30

Hiding Named Helper Visitor inside of `ToString2`

```
package listFW.visitor;
import listFW.*;

public class ToString2WithHiddenHelper implements IListAlgo {
    public static final
        ToString2WithHiddenHelper Singleton
            = new ToString2WithHiddenHelper();

    private ToString2WithHiddenHelper() {
    }
```

continued on next page

```
    private static class HiddenHelper implements IListAlgo {
        public static final HiddenHelper Singleton = new HiddenHelper();

        private HiddenHelper() {
        }

        public Object emptyCase(IMTList host, Object... acc) {
            return "(" + acc[0] + ")";
        }

        public Object nonEmptyCase(INEList host, Object... acc) {
            return host.getRest().execute(this,
                acc[0] + ", " + host.getFirst());
        }
    }
```

```
    public Object emptyCase(IMTList host, Object... nu) {
        return "()";
    }

   public Object nonEmptyCase(INEList host, Object... nu) {
        return host.getRest().execute(HiddenHelper.Singleton,
            host.getFirst().toString());
    }
}
```

Table 3.31

Hiding Named Helper Visitor inside of `ToString3`	Comments

`package listFW.visitor;`
`import listFW.*;`
 	
`public class ToString3WithHiddenHelper implements IListAlgo {`
`  public static final`
`  ToString3WithHiddenHelper Singleton`
`  = new ToString3WithHiddenHelper();`
 	
`  private ToString3WithHiddenHelper() {`
`  }`	

`  private static class HiddenHelper implements IListAlgo {`
`  public static final HiddenHelper Singleton`
`  = new HiddenHelper();`
 	
`  private HiddenHelper() {`
`  }`
 	
`  public Object emptyCase(IMTList host, Object... nu) {`
`  return ")";`
`  }`
 	
`  public Object nonEmptyCase(INEList host, Object... nu) {`
`  return ", " + host.getFirst()`
`  + host.getRest().execute(this);`
`  }`
`  }`	

continued on next page

```
    public Object emptyCase(IMTList host, Object... nu) {
        return "()";
    }

    public Object nonEmptyCase(INEList host, Object...bu) {
        return "(" + host.getFirst()
            + host.getRest().execute(HiddenHelper.Singleton);
    }
}
```

Table 3.32

3.6.3 3. Anonymous Helpers

Anonymous Helper Visitor inside of `ToString1`	Comments

```
package listFW.visitor;
import listFW.*;

public class ToString1WithAnonymousHelper implements IListAlgo {
    public static final
        ToString1WithAnonymousHelper Singleton
            = new ToString1WithAnonymousHelper();

        private ToString1WithAnonymousHelper() {
    }
```

continued on next page

```
    private static final IListAlgo AnonymousHelper = new IListAlgo() {
        public Object emptyCase(IMTList host, Object... acc) {
            return acc[0] + ")";
        }

        public Object nonEmptyCase(INEList host, Object... acc) {
            return host.getRest().execute(this,
                acc[0] + ", " + host.getFirst());
        }
    }; // PAY ATTENTION TO THE SEMI-COLON HERE!
```

```
    public Object emptyCase(IMTList host, Object... nu) {
        return "()";
    }

    public Object nonEmptyCase(INEList host, Object... nu) {
        return host.getRest().execute(AnonymousHelper,
            "(" + host.getFirst());
    }
}
```

Table 3.33

Anonymous Helper Visitor inside of `ToString2`	Comments

```
package listFW.visitor;
import listFW.*;

public class ToString2WithAnonymousHelper implements IListAlgo {
    public static final
        ToString2WithAnonymousHelper Singleton
                    = new ToString2WithAnonymousHelper();
    private ToString2WithAnonymousHelper() {
    }
```

continued on next page

```
    private static final IListAlgo AnonymousHelper = new IListAlgo() {
        public Object emptyCase(IMTList host, Object... acc) {
            return "(" + acc[0] + ")";
        }

        public Object nonEmptyCase(INEList host, Object... acc) {
            return host.getRest().execute(this,
                        acc[0] + ", " + host.getFirst());
        }
    }; // PAY ATTENTION TO THE SEMI-COLON HERE!
```

```
    public Object emptyCase(IMTList host, Object... nu) {
        return "()";
    }

    public Object nonEmptyCase(INEList host, Object... nu) {
        return host.getRest().execute(AnonymousHelper,
                host.getFirst().toString());
    }
}
```

Table 3.34

Anonymous Helper Visitor inside of ToString3 | Comments

```
package listFW.visitor;
import listFW.*;

public class ToString3WithAnonymousHelper implements IListAlgo {
    public static final
        ToString3WithAnonymousHelper Singleton
            = new ToString3WithAnonymousHelper();

    private ToString3WithAnonymousHelper() {
    }
```

continued on next page

```
    private static final IListAlgo AnonymousHelper = new IListAlgo() {
        public Object emptyCase(IMTList host, Object... nu) {
            return ")";
        }

        public Object nonEmptyCase(INEList host, Object... nu) {
            return ", " + host.getFirst()
                + host.getRest().execute(this);
        }
    }; // PAY ATTENTION TO THE SEMI-COLON HERE!
```

```
    public Object emptyCase(IMTList host, Object... nu) {
        return "()";
    }

    public Object nonEmptyCase(INEList host, Object... nu) {
        return "(" + host.getFirst()
            + host.getRest().execute(AnonymousHelper);
    }
}
```

Table 3.35

3.6.4 4. Factory with Anonymous Inner Classes

	Comments

`package listFW.factory;`
`import listFW.*;`
 	
`public class InnerCompListFact implements IListFactory {`
`  public static final InnerCompListFact Singleton`
`  = new InnerCompListFact();`
 	
`  private InnerCompListFact() {`
`  }`	

	continued on next page

```
    private final static IListAlgo ToStrHelp = new IListAlgo() {
        public Object emptyCase(IMTList host, Object... acc) {
            return acc[0] + ")";
        }

        public Object nonEmptyCase(INEList host, Object... acc) {
            return host.getRest().execute(this,
                acc[0] + ", " + host.getFirst());
        }
    }; // PAY ATTENTION TO THE SEMI-COLON HERE!
```

```
    private final static IMTList MTSingleton = new IMTList (){
        public Object execute(IListAlgo algo, Object... inp) {
            return algo.emptyCase(this, inp);
        }

        public String toString() {
            return "()";
        }
    }; // PAY ATTENTION TO THE SEMI-COLON HERE!
```

```
    public IMTList makeEmptyList() {
        return MTSingleton;
    }
```

```
    public INEList makeNEList(final Object first,
                          final IList rest) {
        return new INEList() {
            public Object getFirst() {
                return first;
            }

            public IList getRest() {
                return rest;
            }

            public Object execute(IListAlgo algo,
                                  Object... inp) {
                return algo.nonEmptyCase(this, inp);
            }

            public String toString() {
                return (String)rest.execute(ToStrHelp,
                                            "(" + first);
            }
        };
    }
}
```

Note how the code inside the anonymous inner class references first and rest of the parameter list. `first` and `rest` are said to be in the **closure** of the anonymous inner class. **Here is an important Java syntax rule:** For an local inner class defined inside of a method to access a **local** variable of the method, this local variable must be declared as `final`.

Available for free at Connexions <http://legacy.cnx.org/content/col10213/1.37>

continued on next page

Table 3.36

Click here to download the code of all of the above[16].

3.6.5 5. Classes defined inside of another Class

Besides fields and methods, a Java class can also contain other classes. And just like a field or method defined inside of a class, a class defined inside of another class can be static or non-static. Here is the syntax.

```
    class X {
    // fields of X ...
    // methods of X ...
    /**
     * named class Y defined inside of class X:
     */
    [public|protected|private][static][final][abstract] class Y [extends A][implements B] {
        // fields of Y ...
        // methods of Y ...
        // classes of Y ...
    }
}
```

3.6.5.1 Scope Specifier

When an embedded class is defined as `static`, it is called a **nested class**. **The members (i.e. fields, methods, classes) of a (static) nested class can access only static members of the enclosing class.**

When an embedded class is non-`static`, it is called an **inner class**. The members of an inner class can access ALL members of the enclosing class. The enclosing class (and its enclosing class, if any, and so on) contains the environment that completely defines the inner class and constitutes what is called the **closure** of the inner class. As all functional programmers should know, closure is a powerful concept. One of the greatest strength in the Java programming language is the capability to express closures via classes with inner classes. We shall see many examples that will illustrate this powerful concept in other modules.

Inner classes do not have to be anonymous as shown in the above examples. They can be named as well.

3.6.5.2 Access Specifier

Just like any other class, a class defined inside of another class can be **public**, **protected**, **package** private, or **private**.

3.6.5.3 Extensibility Specifier

Just like a regular class, a **final** nested/inner class cannot extended.

3.6.5.4 Abstract Specifier

Just like a regular class, an **abstract** nested/inner class cannot be instantiated.

[16]See the file at <http://legacy.cnx.org/content/m17220/latest/listFW.zip>

3.6.5.5 Inheritance Specifier

Just like a regular class, an nested/inner can extend any non-final class and implement any number of interfaces that are within its scope.

3.6.5.6 Usage

Nested classes are used mostly to avoid name clash and to promote and enforce information hiding. For example the specialized double precision geometric point class, `Point2D.Double` is defined as an nested class of `Point2D` which is in turn a nested class of `Point`. In such, it has access to all the functionality of its parent plus it avoids the name clash with the double precision number class, `Double`

Inner classes are used to create (at run-time) objects that have direct access to the internals of the outer object and perform complex tasks that simple methods cannot do. Most of the time, they are defined anonymously. For examples, "**event listeners**" for Java GUI components are implemented as inner classes. The dynamic behavior and versatility of these "listeners" cannot be achieved by the addition of a set of fixed methods to a GUI component. We shall study Java event handling soon!

An inner object can be thought as an extension of the outer object.

3.6.6 6. Closure

In functional programming, the closure of a function (**lamdba**) consists of the function itself and an environment in which the function is well-defined. In Java, a function is replaced by a class. An inner class is only defined in the context of its outer object (and the outer object of the outer object, etc...). An inner class together with its nested sequence of outer objects in which the inner class is well-defined is the equivalent of the notion of closure in functional programming. Such a notion is extremely powerful. Just like knowing how to effectively use lambda expressions and higher order functions is key to writing powerful functional programs in Scheme, effective usage of anonymous inner classes is key to writing powerful OO programs in Java.

Some important points to remember about closures and inner classes:

- An object's closure is defined at the time of its creation.
- An object "remembers" its closure for its entire lifetime.
- An inner object's closure includes any local variables that are declared as `final`, plus all fields of the enclosing object, including `private` ones.
- Every time a factory method is run, where a new anonymous object is created, a new closure for that object is created. This is because the local variables could change between calls.
- Closures enable decoupled communication because an inner class can communicate with its outer class through its closure.

One of the most important ways in which we will use anonymous inner classes it to take advantage of their closure properties. Anonymous inner classes are the only objects in Java that can be instantiated in such a manner that the variables in their environments (closures) can be dynamically defined. That is, since an anonymous inner class can reference a local variable (that is declared `final`) and since local variables are created every time a method is called, then every the anonymous inner class object created has a different set of dynamically created variables that it is referencing. This means that we can make unique objects with unique behaviors at run time.

Factories are a natural partner with anonymous inner classes. With a factory that returns anonymous inner classes, we can instantiate unique objects with unique behaviors. If the factory method takes in parameters, there local variables can be used to alter the resultant object's behavior, a process called "currying[17] " (named after the famous mathematician/computer scientist Haskell Curry[18]). The objects made by the

[17] http://en.wikipedia.org/wiki/Currying
[18] http://www.haskell.org/bio.html

factory are then sent off to various odd and sundry different parts of our OO system but all the while retaining their closures, which were determined at the time of their instantiation. Thus they retain the ability to communicate back to the factory that made them even though they are being used in another part of the system that knows nothing about the factory. We like to call these "**spy objects**" because they act like spies from the factory. This gives us powerful communications even though the system is decoupled.

This is the last piece of our abstraction puzzle! We have

1. **Abstract Structure** – abstract classes, interfaces
2. **Abstract Behavior** – abstract methods, strategies, visitors.
3. **Abstract Construction** – factories
4. **Abstract Environments** – anonymous inner classes, closures.

3.6.6.1 Examples

Example 1: Reversing a list using factory and anonymous inner class helper
Write `Reverse` such that it takes one parameter, the `IListFactory`, but such that its helper only takes one parameter (other than the host list) which is the accumulated list.

```
public class Reverse implements IListAlgo {

    public static final Reverse Singleton = new Reverse();

    private Reverse() {}

    public Object emptyCase(IMTList host0, Object... fac) {
        return ((IListFactory)fac[0]).makeEmptyList();
    }

    public Object nonEmptyCase(INEList host0, Object... fac) {
        final IListFactory f = (IListFactory) fac[0];  // final so that the anon. inner
                                                       // class can access it.

        return host0.getRest().execute(new IListAlgo() {

            public Object emptyCase(IMTList host1, Object... acc) {
                return acc[0];
            }

            public Object nonEmptyCase(INEList host1, Object... acc) {
                return host1.getRest().execute(this,
                    f.makeNEList(host1.getFirst(), (IList) acc[0]));
            }
        },f.makeNEList(host0.getFirst(), f.makeEmptyList()));
    }
}
```

Example 2: Ant World
Imagine a world of ants that live in a one-dimensional space. A queen ant can make a bunch of worker ants. Each time she gives birth to a worker ant, she gives it a name. A worker ant can always tell what its name is. A worker ant from a particular colony can always calculate its distance from its queen. A worker ant can also move its queen to a different location. Wherever the queen moves to, ALL of her workers always know

their relative distance from her. We want to keep track of all the ants in our ant world and all the ants in each of the existing colonies. We want to model the fact that each queen produces its own worker ants, each one which can move its queen around without telling the other ants in the same colony, yet ALL of the ants in the same colony would know where their queen is.

The above can be modeled by a Queen class with an abstract Worker inner class as shown below. This example illustrates the differences between static and non-static fields, methods and embedded classes.

Queen.java

```
/**
 * Models ants living in a 1-dimensional world
 *
 *
 * The Worker inner objects have direct access to the location of its outer
 * Queen object.
 *
 * @author A.L. Cox
 * @author D.X. Nguyen
 * @author S.B. Wong
 * @since 02/07/2003
 */
public class Queen {
    /**
     * The total number of ants (queens and workers for all the queens) that
     * currently exist in our ant world.
     *
     * Why is this field static?
     */
    private static int _ants;

    /**
     * The location of this Queen object with respect to 0.
     *
     * Why is this field non-static?
     */
    private int _origin;

    /**
     * The total numbers of living worker ants produced by this Queen object.
     *
     * Why is this field non-static?
     */
    private int _workers;

    /**
     * Is part of a Queen instance, just like the origin field and the
     * makeWorker() method are parts of a Queen instance.
     * Any concrete implementation of Worker must implement the getName()
     * method to return the name given by its Queen at birth time.
     *
     * Why can't this class be static?
     */
```

```
public abstract class Worker {

    /**
     * The location of this Worker object.
     */
    private int _location;

    /**
     * Increments _ants and _workers because every time a Worker is
     * instantiated, a new ant is added to our ant world, a new worker ant
     * is added to the colony of this outer Queen object.
     * @param loc the starting location of this Worker object.
     */
    public Worker(int loc) {
        _location = loc;
    _ants++;            // The worker is an ant.
    _workers++;
    }

    /**
     * @return the relative distance between this Worker and its outer Queen
     * object.
     */
    public int calcDist() {
       return _location - _origin;
    }

    /**
    * The name will be given at birth.  The code cannot be written at
    * this point and thus must be abstract.
    * @return name of the worker
    */
    public abstract String getName();


    /**
    * Our worker has been stepped on!  (No one holds a reference
    * to this object anymore.  It's being garbage collected.)
    */
    protected void finalize() {
        _ants--;            // The worker is an ant.
        _workers--;
    }

    /**
     * Changes the location of this Worker object to a new location.
     * @param loc the new location for this Worler object.
     */
    public void moveTo(int loc) {
        _location = loc;
    }
```

```
    /**
     * Changes the origin of the outer Queen object to a new location.
     * @param org  the new origin for the outer Queen object.
     */
    public void moveQueen(int org) {
        moveTo(org);
        _origin = org;
    }
}

/**
 * Initializes the origin of this Queen object to a given location.
 * Increments _ants since this new Queen object is an ant.
 * @param org the starting origin of this Queen object.
 */
public Queen(int org) {
_ants++;            // The queen is an ant.
    _origin = org;
}

/**
 * Return the total number of all ants, including all of the
 * queens and their respective workers.
 *
 * Why is this method static?
 * Can it be non-static?
 */
public static int countAllAnts() {
return _ants;
}

/**
 * @return the total number of workers that belong to this Queen object
 *
 * Why isn't this method static?
 */
public int countMyWorkers() {
return _workers;
}

/**
 * Factory method: relegate the task of manufacturing concrete
 * Worker objects to the Queen object because the Queen object
 * intrinsically "knows" how to make its inner objects.
 * @param name The name of the Worker.
 */
public Worker makeWorker(final String name) {
    // Anonymously create a Worker object by overriding getName().
    return new Worker(_origin) {
        public String getName() {
```

```
                return name;  // requires the parameter name to be final.
            }
        };
    }
}
```

3.6.6.1.1 Exercise:

Write a JUnit test program for the Queen and Worker classes.

3.6.7 7. Changing "states" - progressing to non-functional programming

In the above Queen example, the locations of a `Queen` object and its `Worker` objects are subject to change. Every time a `Worker` ant moves its `Queen`, it has the side effect of changing the `Queen`'s origin. The `Queen` object remains the "same". This seems like a reasonable thing to have. In the functional programming paradigm, to move the `Queen`, the `Worker` ant would have to instantiate a new `Queen` object with a new location and find a way to associate itself to this new `Queen`, which does not seem to model the "real" ant world very well. By allowing our objects to change their internal states in appropriate situations, our programming model becomes more "realistic", more efficient, and in many cases "simpler".

To further illustrate the relevance of state changes, consider the problem of moving the minimum element of a list of integers to the front of the list. Since the current `IList` model is immutable, we can only solve the problem by returning a copy of the original list with its minimum at the front. Below is an algorithm to solve the problem that makes use of an accumulator to accumulate the current minimum and internally updates it as the list is being traversed. (Note: the list is assumed to hold no duplicate values).

Min2Front.java

```
public class Min2Front implements IListAlgo {

private Integer _accMin;  // accumulated min.
private IListFactory _fact;

public Min2Front(IListFactory f) {
    _fact = f;
}

public Object emptyCase(IMTList mtHost, Object... nu) {
    return mtHost;
}

public Object nonEmptyCase(INEList neHost, Object... nu) {
    // We assign _accMin the first of L as a candidate for minimum:
    _accMin = (Integer)neHost.getFirst();
    /**
    * Let us consider the set S of all elements in L that precede L.
    * S is clearly empty.  At this point we have established the following:
    * _accMin is an element of L and is smaller than all elements of S.
    * We now call on an anonymous helper to operate on L in order to find
    * the minimum and remove it from L.  This helper will recursively
    * travese L to the end in order to obtain the minimum, save it in
    * _accMin and reconstruct the host list L without the minimum on its way
    * back from the recursive list traversal.
```

```
*/
IList withoutMin = (IList)neHost.execute(new IListAlgo() {
    /**
    * Note: when L executes this helper, this case is called since L is
    * not empty.  Thus for the first call to this method, h is L.
    * We update _accMin to ensure that it is an element of L and is the
    * minimum of all elements in L that precedes the rest of the host
    * parameter h.  Then we recursively call this helper on h.getRest()
    * to save the minimum in _accMin and create a copy of h.getRest()
    * that does not contain _accMin.
    */
    public Object nonEmptyCase(INEList h, Object... nu) {
        if ((Integer)h.getFirst() < accMin) {
            _accMin = first;
        }
        /**
        * At this point, we have established the following invariant:
        * _accMin is an element of L and is the minimum of all elements
        * in L that precedes h.getRest().
        */
        IList accList = (IList)h.getRest().execute(this, null);
        /**
        * By induction, _accMin is now the minimum of the whole outer
        * host list L, and accList is a copy of h.getRest() that does
        * not contain _accMin.
        */
        if (!first.equals(_accMin)) {
            accList = _fact.makeNEList(first, accList);
        }
        // accList is now a copy of the host h without _accMin.
        return accList;
        /**
        * As noted earlier, L.execute(...) calls nonEmptyCase() since
        * L is not empty.  Thus the first call to nonEmptyCase() is the
        * call with L as the value for the host parameter h.  So, when
        * we return from this first call, accList is a copy of L without
        * the minimum stored in _accMin.
        */
    }

    /**
    * This method is only called from inside of the nonEmptyCase()
    * method.  The empty host parameter h marks the end of the outer
    * host list L.
    * _accmin is thus the minimum. The empty list is thus a copy of
    * the outer host list L from the current list (empty) to the end
    * that does not contain _accMin.
    */
    public Object emptyCase(IMTList h, Object... nu) {
        return h;
    }
```

```
        }, null);  // NOTE that the input argument is null since the helper does
                   // not need it.
        /**
        * "Cons" the minimum to the front of the copy of the host that does not
        *  contain this minimum.
        */
        return _fact.makeNEList(_accMin, withoutMin);
    }
}
```

In the above, the comments are longer than the code. The above code and comments can be greatly simplified if the list is mutable. What does it mean for a list to be mutable?

Chapter 4

Mutable Data Structures

4.1 State Design Pattern[1]

When modeling real-life systems, we often find that certain objects in our system seem to change "state" during the course of a computation.

Examples of changing state:

1. A kitten grows up into a cat
2. A car runs into a telephone pole and becomes a wreck.
3. A friend is sad one day, happy another, and grumpy on yet another day.
4. A list changes from empty to non-empty when an element is added.
5. A fractal becomes more complex when it grows
6. etc. etc.

The cat and the kitten are the same animal, but they don't act identically. A car can be driven but a wreck cannot–yet they are the same entity fundamentally. Your friend is the same human being, no matter what their mood. Why shouldn't a list be the same list and a fractal be the same fractal?

When something changes state, it is the same object, but yet it behaves differently. This phenomenon of having an objects change its behavior as if it were suddenly belonging to a whole different class of objects is called "**dynamic reclassification**".

So far we've been using immutable data, and to create a non-empty list from an empty one, required that we make a whole brand-new list. With our use **assignment** ("=") previously, we've changed the **value** of a variable, but never the **behavior** the object it references.

Consider this notion: We want to change the type of the object but we want to **encapsulate that change** so that the outside world does **not** see the type change, only the behavior change.

Let's work with an example:

Remember the old arcade game, "Frogger"? That's the one where a traffic-challenged amphibian attempts to hop across multiple lanes of speeding cars and trucks, hopefully without being converted into the road-kill-du-jour.

(Here's an on-line version: http://www.gamesgnome.com/arcade/frogger/[2])

Well, let's look at what a frog is here:

A live frog

- Has a well-defined position
- Has a green color
- Can move from place to place

[1] This content is available online at <http://legacy.cnx.org/content/m17225/1.5/>.

[2] http://www.gamesgnome.com/arcade/frogger/

- Dies when hit by a vehicle.

On the other hand, a dead frog

- Has a well-defined position
- Has a decided red color.
- Cannot move from place to place
- Doesn't die when hit by a vehicle because it is already dead.

Using our trusty separation of variant and invariant, we can see that the position of a frog is an invariant but all the other behaviors are variants. Thus we want to separate out these variants into their own subclasses of an invariant abstract class. We then use composition to model the frog having an abstract state, which could be either alive or dead:

Download the code here.[3]

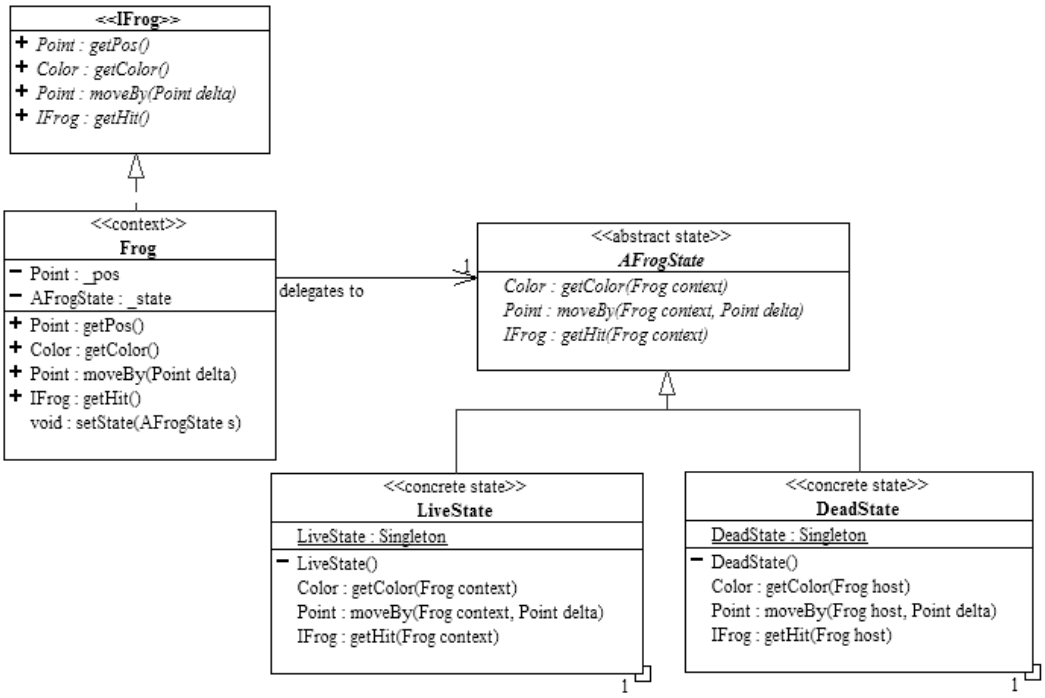


Figure 4.1

Click here to download the full javadoc documentation of the above code.[4]

The variant behaviors are represented by the abstract `AFrogState`, with the `DeadState` and `LiveState` implementing its abstract behaviors.

(Note: The `IFrog` interface is there simply to allow different formulations of the frog to exist. See the Question (Section 4.1.1: Question:) below.)

[3] See the file at <http://legacy.cnx.org/content/m17225/latest/frog.zip>
[4] See the file at <http://legacy.cnx.org/content/m17225/latest/frogDocs.zip>

For those variant behaviors, all the main `Frog` does is to **delegate** (pass the call on to and return the result of) to the `_state` that it has. If the `_state` is a `LiveState`, then the `Frog` will act as if were alive because the `LiveState` only contains live-like behaviors. On the other hand, if the state is a `DeadState`, then the delegation to the `_state` will produce dead-like behavior. The `LiveState`'s `getHit` behavior will cause the `Frog`'s `_state` to change from referencing a `LiveState` instance to referencing a `DeadState` instance.
No conditionals are needed!!
The Frog behaves the way it does because of what it is at that moment, not because of what it can figure out about itself then.

This is an example of the State Design Pattern. Click here for more information on the State design pattern.[5]

From the outside, nothing about the internal implementation of `Frog` can be seen. All one can see is its public behavior. The implementations of the state design pattern are completely encapsulated (within the frog package, in this formulation).. For instance, if one is moving a live `Frog`, it will dutifully move as directed, but if in the middle somewhere, the `Frog` is hit, then it will immediately stop moving, no matter how much it is asked to do so. If one is checking its color, the live `Frog` is a healthy green but right after its accident, it will report that it is deathly red.

Notice how the Frog changes its behavior and always behaves correctly for its situation, **with no conditional statements whatsoever**.

4.1.1 Question:

A very nice technique, when it is possible, is to implement the State pattern using anonymous inner classes. Can you write an `IFrog` implementation that encapsulates the states using nested class(es)and anonymous inner class(es)?

- It can be done using only one publicly visible class and no package visible classes at all.
- The only methods needed are those specified by `IFrog`.
- How does using anonymous inner class(es) reduce the number of parameters passed to the state?
- How does using the anonymous inner class(es) reduce the number of non-public methods?

4.1.2 Onward!

Looking way back to the beginning of the semester, we now have to ask, "Can we use this technology to create a **mutable** list? How will this affect the visitors and their execute function?" Hmmmm.....

4.2 Mutable Linear Recursive Structure[6]

4.2.1 1. State Pattern and Dynamic Reclassification

In programming, it is often necessary to have objects with which one can store data, retrieve data when needed, and remove data when no longer needed. Such objects are instances of what we call container structures.

A mutable container structure is a system that may change its state from empty to non-empty, and vice-versa. For example, an empty container changes its state to non-empty after insertion of an object; and when the last element of a container is removed, its changes its state to empty. Figure 1 below diagrams the state transition of a container structure.

[5] "State Design Pattern" <http://legacy.cnx.org/content/m17047/latest/>
[6] This content is available online at <http://legacy.cnx.org/content/m17265/1.5/>.

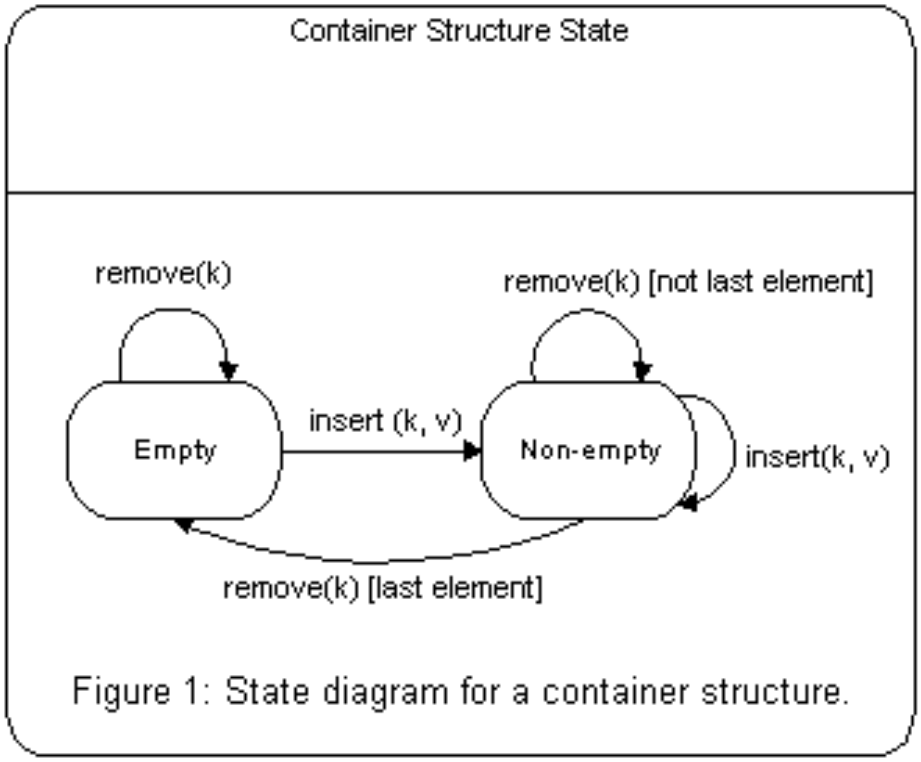


Figure 4.2: State transition diagram for container structures

For each distinct state, the algorithms to implement the methods differ. For example, the algorithm for the retrieve method is trivial in the empty state -it simply returns null- while it is more complicated in the non-empty state. The system thus behaves as if it changes classes dynamically. This phenomenon is called "dynamic reclassification." The state pattern is a design solution for languages that do not directly support dynamic reclassification. This pattern can be summarized as follow.

- Define an abstract class for the states of the system. This abstract state class should provide all the abstract methods for all the concrete subclasses.
- Define a concrete subclass of the above abstract class for each state of the system. Each concrete state must implement its own concrete methods.
- Represent the system by a class, called the context, containing an instance of a concrete state. This instance represents the current state of the system.
- Define methods for the system to return the current state and to change state.
- Delegate all requests made to the system to the current state instance. Since this instance can change dynamically, the system will behave as if it can change its class dynamically.

Below is the UML class diagram for the state design pattern[7].

[7] "State Design Pattern" <http://legacy.cnx.org/content/m17047/latest/>

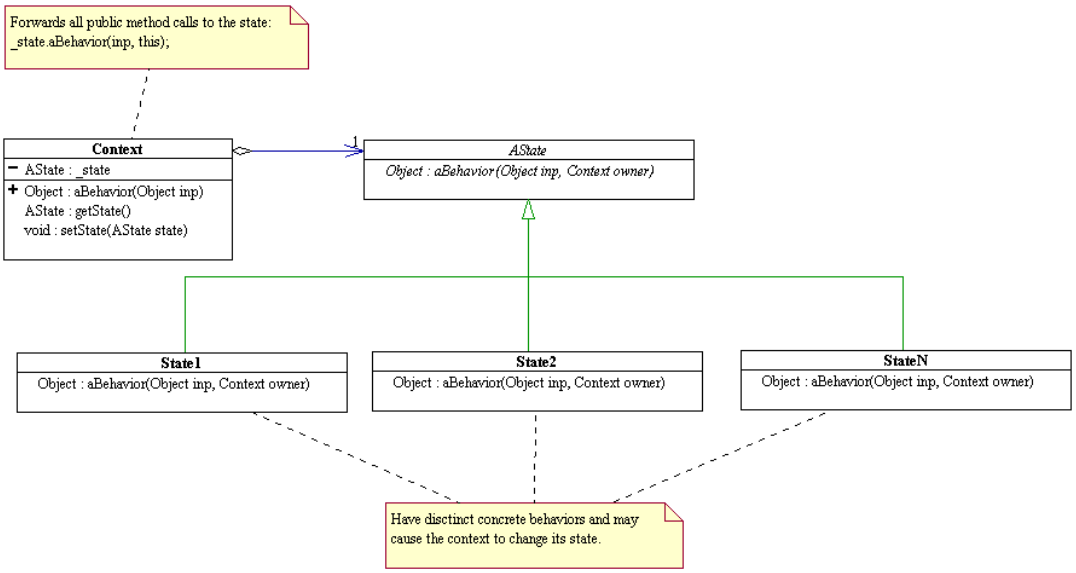


Figure 4.3: UML class diagram for the state design pattern

4.2.2 2. Mutable Linear Recursive Structure Framework

A mutable linear recursive structure (`LRStruct`) can be in the empty state or in a non-empty state. If it is empty, it contains no object. Otherwise, it contains an object called **first**, and a `LRStruct` object called **rest**. When we insert a data object into an empty `LRStruct`, it changes it state to non-empty. When we remove the last element from an non-empty `LRStruct`, it changes its state to empty. We model a `LRStruct` using the state pattern, and as in the case of the immutable list, (Section 3.4) we also apply the visitor pattern to obtain a **framework**. Below is the UML class diagram of the `LRStruct` framework. Because of the current limitation of our diagramming tool, we are using the `Object[] input` notation to represent the variable argument list `Object... input`. Click here to download the code[8]. Click here to download the javadoc documentation.[9]. We will study the implementation code in the next lecture.

[8] See the file at <http://legacy.cnx.org/content/m17265/latest/lrs.zip>
[9] See the file at <http://legacy.cnx.org/content/m17265/latest/lrsdocs.zip>

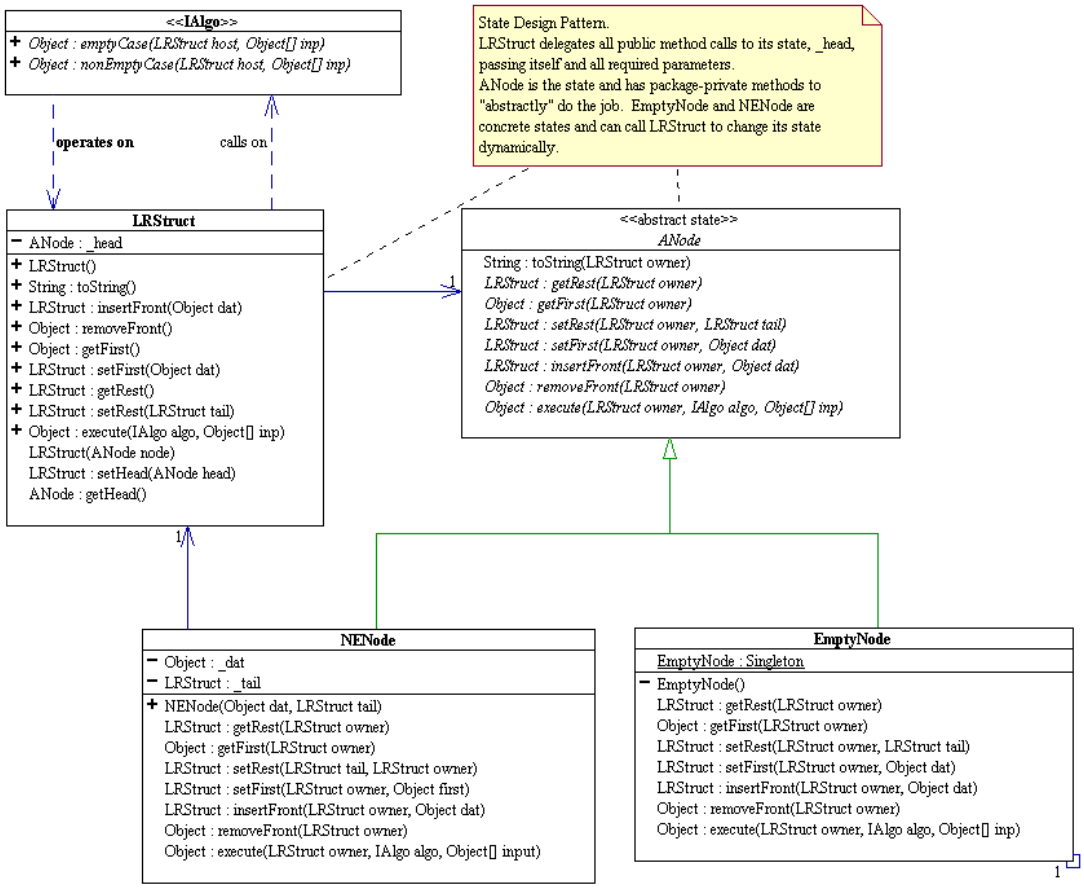


Figure 4.4: State and Visitor Patterns for Mutable Linear Recursive Structure

The public constructor:

- `LRStruct()`

and the methods:

- `insertFront(...)`
- `removeFront(...)`
- `getFirst()`
- `setFirst(...)`
- `getRest()`
- `setRest(...)`

of `LRStruct` expose the structure of an `LRStruct` to the client and constitute the **intrinsic** structural behavior of an `LRStruct`. They form a **minimal and complete** set of methods for manipulating an `LRStruct`. Using them, a client can create an empty `LRStruct`, store data in it, and remove/retrieve data from it at will.

The method,

- `Object execute(IAlgo algo, Object inp)`

is called the extensibility "**hook**". It allows the client to add an open-ended number of new application-dependent behaviors to the data structure `LRStruct`, such as computing its length or merging one `LRStruct` with another, without modifying any of the existing code. The application-dependent behaviors of `LRStruct` are **extrinsic** behaviors and are encapsulated in a union represented by a visitor interface called `IAlgo`.

When a client programs with `LRStruct`, he/she only sees the public methods of `LRStruct` and `IAlgo` . To add a new operation on an `LRStruct`, the client writes appropriate concrete classes that implements `IAlgo`. The framework dictates the overall design of an algorithm on `LRStruct`: since an algorithm on `LRStruct` must implement `IAlgo` , there must be some concrete code for `emptyCase(...)` and some concrete code for `nonEmptyCase(...)`. For example,

```
public class DoSomethingWithLRS implements IAlgo {
   // fields and constructor code...
   public Object emptyCase(LRStruct host, Object... inp) {
       // some concrete code here...
       return some Object; // may be null.
   }

   public Object nonEmptyCase(LRStruct host, Object... inp) {
       // some concrete code here...
       return some Object; // may be null.
   }
}
```

As illustrated in the above, an algorithm on `LRStruct` is "**declarative**" in nature. It does not involve any conditional to find out what state the `LRStruct` is in in order to perform the appropriate task. It simply "declares" what needs to be done for each state of the host `LRStruct`, and leaves it to the polymorphism machinery to make the correct call. Polymorphism is exploited to minimize flow control and reduce code complexity.

To perform an algorithm on an `LRStruct`, the client must "ask" the `LRStruct` to "execute" the algorithm and passes to it all the inputs required by the algorithm.

```
LRStruct myList = new LRStruct();  // an empty list
// code to call on the structural methods of myList, e.g. myList.insertFront(/*whatever*/)
// Now call on myList to perform DoSomethingWithLRS:
Object result = myList.execute(
    new DoSomethingWithLRS(/* constructor argument list */), -2.75, "abc");
```

Without knowing how LRStruct is implemented, let us look an example of an algorithm on an LRStruct .

4.2.3 3. Example

Consider the problem of inserting an Integer object **in order** into a **sorted** list of Integers. Let us contrast the insert in order algorithms between `IList`, the immutable list, and `LRStruct`, the mutable list.

Insert in order using factory

```
import listFW.*;

public class InsertInOrder
  implements IListAlgo {

  private IListFactory _fact;

  public InsertInOrder(IListFactory lf) {
    _fact = lf;
  }

  /**
  * Simply makes a new non-empty list with
  * the given parameter n as first.
  * @param host an empty IList.
  * @param n n[0] is an Integer to be
  *   inserted in order into host.
  * @return INEList.
  */
  public Object emptyCase(
    IEmptyList host, Object... n) {

    return _fact.makeNEList(n[0], host);
  }

  /**
  * Based on the comparison between first
  * and n, creates a new list or recur!
  * @param host a non-empty IList.
  * @param n an Integer to be inserted in
  *   order into host.
  * @return INEList
  */
  public Object nonEmptyCase(
    INEList host, Object... n) {

    return (Integer)n[0] <
      (Integer)host.getFirst() ?
        _fact.makeNEList(n[0], host):
        _fact.makeNEList(host.getFirst(),
          (IList)host.getRest().execute(
            this, n[0]));
  }
}
```

Insert in order using mutation

```
import lrs.*;

public class InsertInOrderLRS
  implements IAlgo {

  public static final InsertInOrderLRS
    Singleton = new InsertInOrderLRS();

  private InsertInOrderLRS() {}

  /**
  * Simply inserts the given parameter n at
  * the front.
  * @param host an empty LRStruct.
  * @param n n[0] isan Integer to be
  *  inserted in order into host.
  * @return LRStruct
  */
  public Object emptyCase(
    LRStruct host, Object... n) {

    return host.insertFront(n[0]);
  }

  /**
  * Based on the comparison between first
  * and n, inserts at the front or recurs!
  * @param host a non-empty LRStruct.
  * @param n n[0] is  an Integer to be
  *   inserted in order into host.
  * @return LRStruct
  */
  public Object nonEmptyCase(
    LRStruct host, Object... n) {

    if ((Integer)n[0] < (
      Integer)host.getFirst()) {

      return host.insertFront(n[0]);
    }
    else {
      return host.getRest().execute(
        this, n[0]);
    }
  }
}
```

Note that the insert in order algorithm for `LRStruct` need not create any new list and thus needs no factory.Download the above code: lrs.zip[10]

continued on next page

Table 4.1

4.3 Binary Tree Structure[11]

Up until now, we have been organizing data in a "linear" fashion: one item after another. The corresponding data structures are the immutable scheme list (`IList`) and the mutable linear recursive structure (`LRStruct`). Now, suppose we want to model something like the following organization chart.

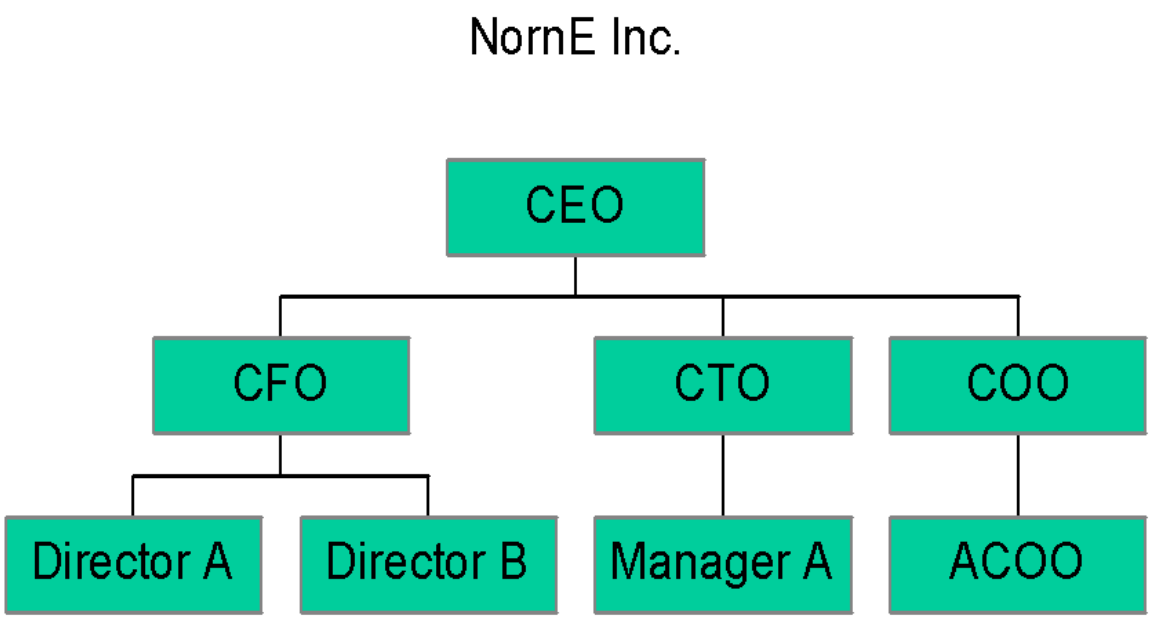


Figure 4.5

A structure of data such as the above is called a **tree**. We first design a special kind of **mutable** tree structure called binary trees.

4.3.1 1. Binary Tree Object Model

Definition 4.1: Binary Tree
A (mutable) binary tree, `BiTree`, can be in an empty state or a non-empty state:

- When it is empty, it contains no data.
- When it is not empty, it contains a data object called the root element, and 2 distinct `BiTree` objects called the left subtree and the right subtree.

[10] See the file at <http://legacy.cnx.org/content/m17265/latest/lrs.zip>
[11] This content is available online at <http://legacy.cnx.org/content/m17289/1.3/>.

We implement the above object structure with a combination of state/composite/visitor patterns, in a manner analogous to `LRStruct`[12].

Below is the public interface of the binary tree framework. Click here for javadoc documentation.[13]

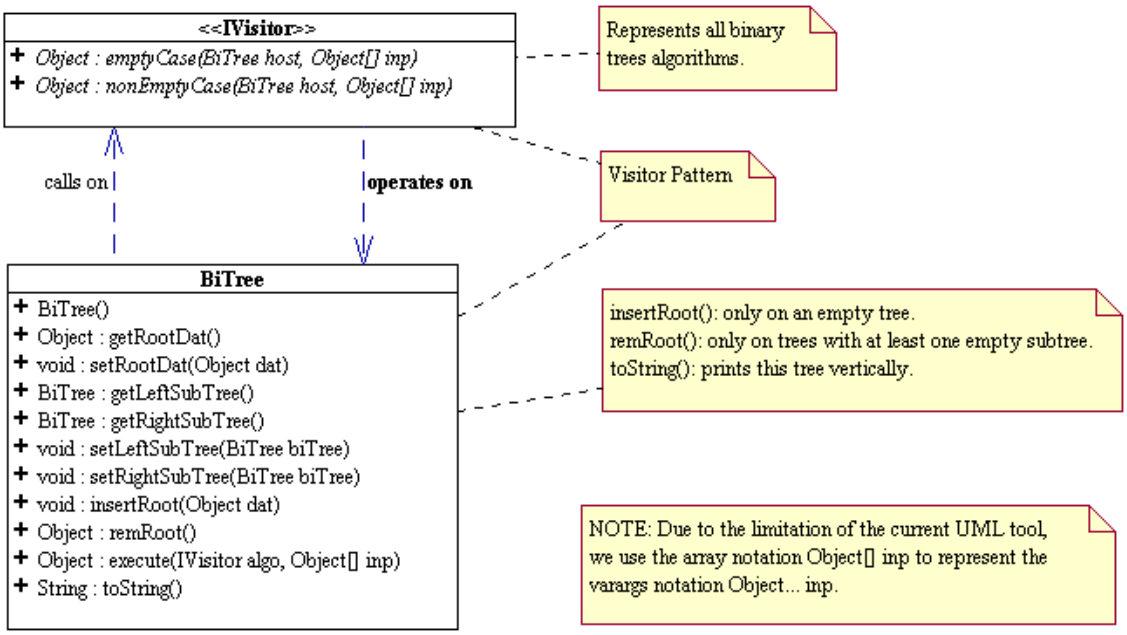


Figure 4.6

The implementation details are given in the following UML class diagram. Click here for javadoc documentation.[14]

[12] See the file at <http://legacy.cnx.org/content/m17289/latest/lrs-javadocs.zip>
[13] See the file at <http://legacy.cnx.org/content/m17289/latest/brs-javadocs.zip>
[14] See the file at <http://legacy.cnx.org/content/m17289/latest/brs-javadocs.zip>

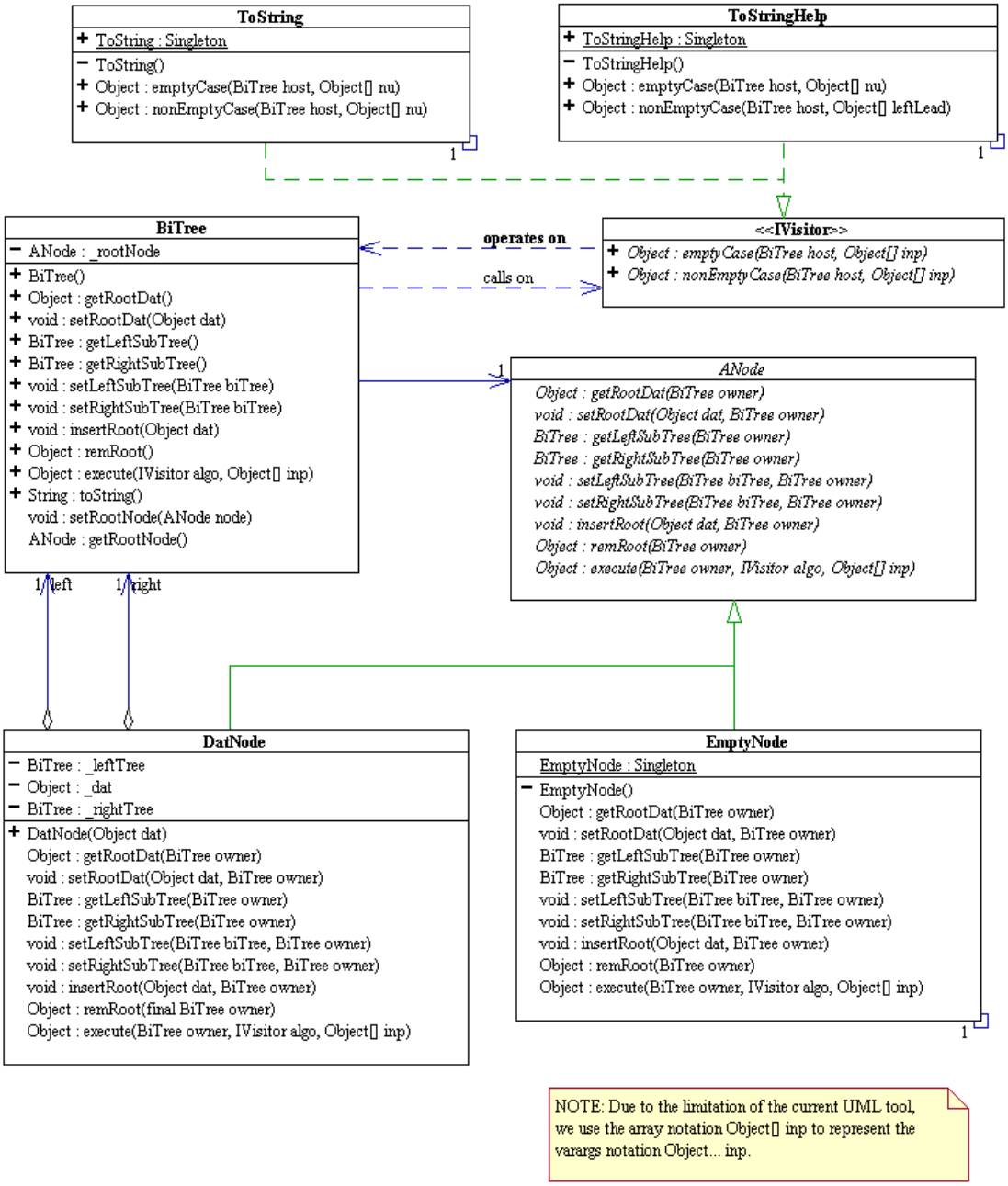

ToString
+ ToString : Singleton
- ToString()
+ Object : emptyCase(BiTree host, Object[] nu)
+ Object : nonEmptyCase(BiTree host, Object[] nu)
1
ToStringHelp
+ ToStringHelp : Singleton
- ToStringHelp()
+ Object : emptyCase(BiTree host, Object[] nu)
+ Object : nonEmptyCase(BiTree host, Object[] leftLead)
1
BiTree
- ANode : _rootNode
+ BiTree()
+ Object : getRootDat()
+ void : setRootDat(Object dat)
+ BiTree : getLeftSubTree()
+ BiTree : getRightSubTree()
+ void : setLeftSubTree(BiTree biTree)
+ void : setRightSubTree(BiTree biTree)
+ void : insertRoot(Object dat)
+ Object : remRoot()
+ Object : execute(IVisitor algo, Object[] inp)
+ String : toString()
void : setRootNode(ANode node)
ANode : getRootNode()
operates on
calls on
<<IVisitor>>
+ Object : emptyCase(BiTree host, Object[] inp)
+ Object : nonEmptyCase(BiTree host, Object[] inp)
1
ANode
Object : getRootDat(BiTree owner)
void : setRootDat(Object dat, BiTree owner)
BiTree : getLeftSubTree(BiTree owner)
BiTree : getRightSubTree(BiTree owner)
void : setLeftSubTree(BiTree biTree, BiTree owner)
void : setRightSubTree(BiTree biTree, BiTree owner)
void : insertRoot(Object dat, BiTree owner)
Object : remRoot(BiTree owner)
Object : execute(BiTree owner, IVisitor algo, Object[] inp)
1 left
1 right
DatNode
- BiTree : _leftTree
- Object : _dat
- BiTree : _rightTree
+ DatNode(Object dat)
Object : getRootDat(BiTree owner)
void : setRootDat(Object dat, BiTree owner)
BiTree : getLeftSubTree(BiTree owner)
BiTree : getRightSubTree(BiTree owner)
void : setLeftSubTree(BiTree biTree, BiTree owner)
void : setRightSubTree(BiTree biTree, BiTree owner)
void : insertRoot(Object dat, BiTree owner)
Object : remRoot(final BiTree owner)
Object : execute(BiTree owner, IVisitor algo, Object[] inp)
EmptyNode
EmptyNode : Singleton
- EmptyNode()
Object : getRootDat(BiTree owner)
void : setRootDat(Object dat, BiTree owner)
BiTree : getLeftSubTree(BiTree owner)
BiTree : getRightSubTree(BiTree owner)
void : setLeftSubTree(BiTree biTree, BiTree owner)
void : setRightSubTree(BiTree biTree, BiTree owner)
void : insertRoot(Object dat, BiTree owner)
Object : remRoot(BiTree owner)
Object : execute(BiTree owner, IVisitor algo, Object[] inp)
1
NOTE: Due to the limitation of the current UML tool, we use the array notation Object[] inp to represent the varargs notation Object... inp.

Figure 4.7

4.3.2 2. Implementation Details

SOURCE CODE: Click here for source code.[15]

The code for `BiTree` is trivial, as it simply delegates most calls to its state, the root node. The real work is done in the state. The code for `EmptyNode` and `DatNode` for the most part are equally trivial. The insertion and removal of the root data of a `BiTree` require some work and need some explanation. When does it make sense to remove the root node from a (binary) tree? That is, when can one unambiguously remove the root node from a binary tree?

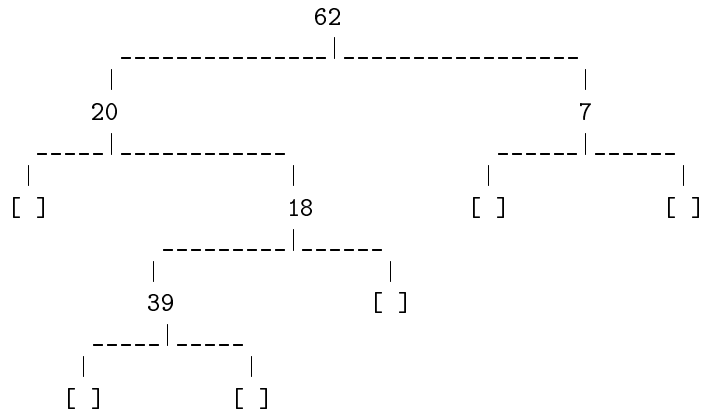
Clearly, when both subtrees of a root node are non-empty, then removing the root node is problematic. However, it makes sense to allow root removal when at least one of the subtrees is empty. Suppose one of the subtrees is empty, then `BiTree.remRoot()` will change the state of this BiTree to the state of the other subtree. For example, when the left subtree is empty, root removal of the parent tree will set the parent tree to its right subtree.

(a) (b)
EmpDatN-
tyNode
ode

Figure 4.8: (a) EmptyNode (b) DatNode

4.3.3 3. The Best Tree Printing Algorithm in Texas

Consider the binary tree displayed in the following "horizontal" manner:



Such horizontal display of a tree is not very convenient when the tree is wide. It is more convenient to display the tree in a "vertical" manner.

The following visitor and its helper print the above tree "vertically" as shown below:

```
62
```

[15] See the file at <http://legacy.cnx.org/content/m17289/latest/brs.zip>

```
|_ 20
| |_ []
| |_ 18
|    |_ 39
|    |  |_ []
|    |  |_ []
|    |_ []
|_ 7
   |_ []
   |_ []
```

Let's study the algorithm.

(a) (b)

Figure 4.9

4.4 Binary Search Tree[16]

4.4.1 Binary Search Tree

4.4.1.1 1. Binary Search Tree Property (BSTP)

Consider the following binary tree of Integer objects.

```
    -7
|_ -55
| |_ []
| |_ -16
|    |_ -20
|    | |_ []
|    | |_ []
|    |_ -9
|      |_ []
|      |_ []
|_ 0
  |_ -4
  | |_ []
  | |_ []
  |_ 23
    |_ []
    |_ []
```

Notice the following property:

- all elements in the left subtree are less than the root element,

[16] This content is available online at <http://legacy.cnx.org/content/m32616/1.1/>.

- and the root element is less than all elements in the right subtree.

Moreover, this property holds recursively for all subtrees. It is called the **binary search tree** (BST) property.

In general, instead of Integer objects, suppose we have a set of objects that can be compared for equality with "equal to" and "totally ordered" with an order relation called "less or equal to" . Define "less than" to mean "less or equal to" AND "not equal to". Let T be a BiTree structure that stores such totally ordered objects.

4.4.1.1.1 Definition of Binary Search Tree Property

The **binary search tree property**(BSTP) is defined on the binary tree structure as follows.

- An empty binary tree satisfies the BSTP.
- A non-empty binary tree T satisfies the BSTP if and only if
  - the left and right subtrees of T both satisfy BSTP, and
  - all elements in the left subtree of T are less than the root of T, and
  - the root of T is less than all elements in the right subtree of T.

We can take advantage of this property when looking up for a particular ordered object in the tree. Instead of scanning the whole tree for the search target, we can compare the search target against the root element and narrow the search to the left subtree or the right subtree if necessary. So in the worst possible case, the number of comparisons is proportional to the height of the binary tree. This is a big win if the tree is ***balanced***. It can be proven that when a tree containing N elements is balanced, its height is at most a constant multiple of logN. For example, the height of a balanced tree containing 10^6 elements is at most a fixed multiple of 6. Here is the definition of what a ***balanced tree*** is.

4.4.1.1.2 Definition of Balanced Tree

- An empty tree is ***balanced***.
- A non-empty tree is ***balanced*** if and only if
  - its subtrees are ***balanced*** and
  - the heights of the subtrees differ by a fixed constant or by a fixed constant factor.

A binary tree with the BST property is called a binary search tree. It can serve as an efficient way for storage/retrieval of data. We are lead to the following question: how to create and maintain a binary search tree?

4.4.1.2 2. Binary Search Tree Insertion

Suppose we start with an empty binary tree T and a ***Comparator*** that models a total ordering in a given set of objects S. Then T clearly has the BST property with respect the ***Comparator*** ordering of S. The following algorithm (visitor on binary trees) will allow us to insert elements of S into T and at the same time maintain the BST property for T. This algorithm also works for binary search tree containing Comparable objects.

```
    package brs.visitor;

import brs.*;
import java.util.*;

/**
* Inserts an Object into the host maintaining the host's binary search
```

```
* tree property via a given Comparator.
* Can also be used for Comparable objects.
* Duplication is not allowed: replaces old data object with the new one.
*/
public class BSTInserter implements IVisitor {
    private Comparator _order;

    /**
    * Used when the items in the tree are Comparable objects.
    */
    public BSTInserter() {
        _order = new Comparator() {
            public int compare(Object x, Object y) {
                return ((Comparable)x).compareTo(y);
            }
        };
    }


    /**
    * Used to order the items according to a given Comparator.
    * @param order a total ordering on the items to be stored in the tree.
    */
    public BSTInserter (Comparator order) {
        _order = order;
    }

    /**
    * Returns the host tree where the input is inserted as the root.
    * @param host an empty binary tree, which obviously satisfies BSTP.
    * @param input[0] new data to be inserted.
    * @return host (which is no longer empty).
    */
    public Object emptyCase(BiTree host, Object... input) {
        host.insertRoot (input[0]);
        return host;
    }

    /**
    * If the input is equal to host.getRootDat) then the old root data
    * is replaced by input. Equality here is implicitly defined by the
    * total ordering represented by the Comparator _order.
    * @param host non-empty and satisfies BSTP.
    * @param input[0] new data to be inserted.
    * @return the tree where input[0] is inserted as the root.
    */
    public Object nonEmptyCase(BiTree host, Object... input) {
        Object root = host.getRootDat();
        if (_order.compare(input[0], root) < 0) {
            return host.getLeftSubTree().execute(this, input[0]);
        }
        if (_order.compare(input[0], root) > 0) {
```

```
            return host.getRightSubTree().execute(this, input[0]);
        }
        host.setRootDat(input[0]);
        return host;
    }
}
```

4.4.1.3 3. Binary Search Tree Lookup

Suppose we have a binary search tree based on a given *Comparator* ordering. The following algorithm will allow us to lookup and retrieve a particular data object from the tree. This algorithm also works for binary search tree containing Comparable objects.

```
    package brs.visitor;
import brs.*;
import java.util.*;

/**
* Finds a data object in a binary host tree that satisfies the
* Binary Search Tree Property.
* The algorithm is similar to that of insertion.
*/
    public class BSTFinder implements IVisitor {
    private Comparator _order;
    /**
    * Used when the items in the tree are Comparable objects.
    */
    public BSTFinder() {
        _order = new Comparator() {
            public int compare(Object x, Object y) {
                return ((Comparable)x).compareTo(y);
            }
        };
    }



    /**
    * Used when the items are ordered according to a given Comparator.
    * @param order a total ordering on the items stored in the tree.
    */
    public BSTFinder(Comparator order) {
        _order = order;
    }

    /**
    * Returns the empty host tree itself.
    * @param host an empty binary (which obviously satisfies the
    * Binary Search Tree Property).
    * @param nu not used
    * @return BiTree the empty host tree.
```

```
    */
    public Object emptyCase(BiTree host, Object... nu) {
        return host;
    }

    /**
    * Returns the tree such that whose root, if it exists, is equal to input
    * via the given Comparator _order.
    * @param host non empty and satisfies BSTP.
    * @param input[0] object to be looked up.
    * @return BiTree
    */
    public Object nonEmptyCase(BiTree host, Object... input) {
        Object root = host.getRootDat();
        if (_order.compare(input[0], root) < 0) {
            return host.getLeftSubTree().execute(this, input[0]);
        }
        if (_order.compare(input[0], root) > 0) {
            return host.getRightSubTree().execute(this, input[0]);
        }
        return host;
    }
}
```

4.4.1.4 4. Binary Search Tree Deletion

The algorithm to remove a particular data object from a binary search tree is more involved. When the element to be removed is not at a leaf node, we can replace it with the largest element of the left subtree (if it's not empty) or the smallest element of the right subtree (if it's not empty). In preparation, we write a visitor to find the subtree containing the largest element at the root.

```
    package brs.visitor;
import brs.*;
import ordering.*;

/**
* Returns the subtree of the host with the max value in the root if the tree
* is not empty, otherwise returns the host itself, assuming the tree satisfies
* the BST property.
* @author DXN
*/
public class MaxTreeFinder implements IVisitor {
    public static final MaxTreeFinder Singleton = new MaxTreeFinder ();
    private MaxTreeFinder () {
    }

    /**
    * The host tree is empty: the tree containing the max is the host itself.
    * @param host satisfies the binary search tree property.
    * @param nu not used.
    * @return BiTree the empty host itself.
```

```
    */
    public Object emptyCase(BiTree host, Object... nu) {
        return host;
    }

    /**
    * Asks the right subtree of the host to find the max via an
    * anomymous helper, passing to it the host as a parameter.
    * @param host satisfies the binary search tree property.
    * @param nu not used.
    * @return BiTree the subtree with maximum root.
    */
    public Object nonEmptyCase (BiTree host, Object... nu) {
        return host.getRightSubTree().execute (new IVisitor () {
            /**
            * The BST parent of an empty right subtree contains the max
            * at the root.
            * @param hp[0] a Bitree, the parent of h.
            */
            public Object emptyCase (BiTree h, Object... hp) {
                return hp[0];
            }

            /**
            * Keep going to the right looking for the max.
            * @param hp[0] a Bitree, the parent of h.
            */
            public Object nonEmptyCase (BiTree h, Object... hp) {
                return h.getRightSubTree ().execute (this, h);
            }
        }, host);
    }
}
```

We are now ready to write the deletion algorithm for a binary search tree ordered according to a given Comparator. This algorithm also works for binary search tree containing Comparable objects.

```
    package brs.visitor;
import brs.*;
import java.util.*;

/**
* Deletes the given input from the host tree. Returns TRUE if the input is in
* the host tree, otherwise return FALSE.
Invariant: host contains unique objects and satisfies the BST property.
Post: host does not contains the input.
Algorithm:
Case host is empty:
    returns FALSE because there is nothing to remove.
Case host is not empty:
    if input < root
        return the result of asking the left subtree to delete the input;
```

```
    else if root < input
        return the result of asking the right subtree to delete the input;
    else if host's left subtree is empty (at this point, input == root)
        ask host to remove its root (and become its right subtree)
        returns TRUE
    else {
        ask host to replace the root with the maximum value of its left subtree;
        the subtree that contains the maximum must have an empty right subtree;
        thus it is safe to ask this subtree to remove its root;
        return TRUE
    }

    NOTE:

    As you have been indoctrinated, it is "uncouth" do ask something for what it
    is. Instead of checking a subtree for emptiness, we should ask the subtree to
    help carry out the deletion.
*/
public class BSTDeleter implements IVisitor {
    private Comparator _order;

    /**
    * Used when the items in the tree are Comparable objects.
    */
    public BSTDeleter() {
        _order = new Comparator() {
            public int compare(Object x, Object y) {
                return ((Comparable)x).compareTo(y);
            }
        };
    }


    /**
    * Used when the items are ordered according to a given Comparator.
    * @param order a total ordering on the items stored in the tree.
    */
    public BSTDeleter (Comparator order) {
        _order = order;
    }

    /**
    * There is nothing to delete.
    * @param host is empty and obviously satisfies the BST Property.
    * @param nu not used
    * @return Boolean.FALSE
    */
    public Object emptyCase(BiTree host, Object... nu) {
        return Boolean.FALSE;
    }

    /**
```

```
    if input < root
        ask the host's left subtree to delete the input;
    else if root < input
        ask the host's right subtree to delete the input
    else (at this point, input == root)
        ask the left subtree for help to remove the host's root using an
        anonymous helper.
    * @param host non-empty and satifies BST property
    * @param input[0] object to be deleted from the host.
    * @return Boolean.
    */
    public Object nonEmptyCase(final BiTree host, Object... input) {
        Object root = host.getRootDat();
        if (_order.compare(input[0], root) < 0) {
            return host.getLeftSubTree().execute (this, input[0]);
        }
        if (_order.compare(input[0], root) > 0) {
            return host.getRightSubTree().execute (this, input[0]);
        }
        // At this point: input is equal to root.
        return host.getLeftSubTree().execute(new IVisitor() {
            /**
            * The outer host can safely remove its root and
            * becomes its right subtree.
            * @param h the left subtree of the outer host.
            */
            public Object emptyCase (BiTree h, Object notUsed) {
                host.remRoot();
                return Boolean.TRUE;
            }

            /**
            * Finds the maximum value in the h paramter.
            * Asks the outer host parameter to set its root to this
            * maximum value, and removes this maximum value from the
            * subtree containing this maximum value.
            * @param h the left subtree of the outer host.
            */
            public Object nonEmptyCase (BiTree h, Object notUsed) {
                BiTree maxTree = (BiTree)h.execute(MaxTreeFinder.Singleton);
                host.setRootDat(maxTree.remRoot());
                return Boolean.TRUE;
            }
        });
    }
}
```

We now have all the needed ingredient to implement an efficient container for storage/retrieval/lookup, using the BiTree framework together with the above visitors!

4.5 Arrays and Array Processing[17]

There are many ways to store data. So far, we have covered linear recursive structures, lists, and binary recursive structures, trees. Let's consider another way of storing data, as a contiguous, numbered (indexed) set of data storage elements:

anArray =

itemA	itemB	itemC	itemD	itemE	itemF	itemG	itemH	itemI	itemJ
0	1	2	3	4	5	6	7	8	9

Table 4.2

This "**array**" of elements allows us to access any individual element using a numbered index value.

Definition 4.2: array
At its most basic form, a random access data structure where any element can be accessed by specifying a single index value corresponding to that element.

Example
`anArray[4]` gives us `itemE`. Likewise, the statement `anArray[7] = 42` should replace `itemH` with `42`.

NOTE: Notice however, that the above definition is **not** a recursive definition. This will cause problems.

4.5.1 Arrays in Java

- Arrays...
  - are contiguous (in memory) sets of object references (or values, for primitives),
  - are objects,
  - are dynamically created (via `new`), and
  - may be assigned to variables of type `Object` or primitives
- An array object contains zero or more **unnamed** variables of the **same** type. These variables are commonly called the **elements** of the array.
- A non-negative integer is used to name each element. For example, `arrayOfInts[i]` refers to the **i+1**st element in the `arrayOfInts` array. In computer-ese, an array is said to be a "random access" container, because you can directly (and I suppose, randomly) access any element in the array.
- An array has a limited amount of intelligence, for instance, it does know its maximum length at all times, e.g. `arrayOfInts.length`.
- Arrays have the advantage that they
  - provide random access to any element
  - are fast.
  - require minimum amounts of memory

More information on arrays can be found in the Java Resources web site page on arrays[18]

REMEMBER: **Arrays are size and speed at a price.**

[17] This content is available online at <http://legacy.cnx.org/content/m17258/1.3/>.
[18] http://www.exciton.cs.rice.edu/JavaResources/Java/declarations.htm#Arrays

4.5.1.1 Array Types

- An array type is written as the name of an element type followed by one or more empty pairs of square brackets.
  - For example, `int[]` is the type corresponding to a one-dimensional array of integers.
- An array's length is not part of its type.
- The element type of an array may be any type, whether primitive or reference, including interface types and abstract class types.

4.5.1.2 Array Variables

- Array variables are declared like other variables: a declaration consists of the array's type followed by the array's name. For example, `double[][] matrixOfDoubles;` declares a variable whose type is a two-dimensional array of double-precision floating-point numbers.
- Declaring a variable of array type does not create an array object. It only creates the variable, which can contain a reference to an array.
- Because an array's length is not part of its type, a single variable of array type may contain references to arrays of different lengths.
- To complicate declarations, C/C++-like syntax is also supported, for example,

```
double rowvector[], colvector[], matrix[][];
```

This declaration is equivalent to

```
double[] rowvector, colvector, matrix[];
```

or

```
double[] rowvector, colvector;
double[][] matrix;
```

Please use the latter!

4.5.1.3 Array Creation

- Array objects, like other objects, are created with `new`. For example, `String[] arrayOfStrings = new String[10];`declares a variable whose type is an array of strings, and initializes it to hold a reference to an array object with room for ten references to strings.
- Another way to initialize array variables is

```
int[] arrayOf1To5 = { 1, 2, 3, 4, 5 };
String[] arrayOfStrings = { "array",
                            "of",
                            "String" };
Widget[] arrayOfWidgets = { new Widget(), new Widget() };
```

- Once an array object is created, it never changes length! `int[][] arrayOfArrayOfInt = {{ 1, 2 }, { 3, 4 }};`
- The array's length is available as a final instance variable length. For example,

```
    int[] arrayOf1To5 = { 1, 2, 3, 4, 5 };
System.out.println(arrayOf1To5.length);
```

would print "5".

4.5.1.4 Array Accesses

- Indices for arrays must be `int` values that are greater than or equal to 0 and less than the length of the array. Remember that computer scientists always count starting at zero, not one!
- All array accesses are checked at run time: An attempt to use an index that is less than zero or greater than or equal to the length of the array causes an `IndexOutOfBoundsException` to be thrown.
- Array elements can be used on either side of an equals sign:
  - `myArray[i] = aValue;`
  - `someValue = myArray[j];`
- Accessing elements of an array is fast and the time to access any element is independent of where it is in the array.
- Inserting elements into an array is very slow because all the other elements following the insertion point have to be moved to make room, if that is even possible.

4.5.2 Array Processing Using Loops

More information on loops can be found at the Java Resources web site page on loops[19] .

The main technique used to process arrays is the **for loop**. A `for` loop is a way of processing each element of the array in a sequential manner.

Here is a typical `for` loop:

```
// Sum the number of elements in an array of ints, myArray.
int sum = 0;  // initialize the sum

for(int idx=0; idx < myArray.length; idx++) {  //start idx @ 0; end idx at length-1;
                                               //increment idx every time the loop is processed.
    sum += myArray[idx];  // add the idx'th element of myArray to the sum
}
```

There are a number of things going on in the above `for` loop:

- **Before the loop starts**, the index `idx` is being declared and initialized to zero. `idx` is visible only within the `for` loop body (between the curly braces).
- **At the begnning of each loop iteration**, the index `idx` is being tested in a "termination condition", in this case, `idx` is compared to the length of the list. If the termination condition evaluates to `false`, the loop will immediately terminate.
- **During each loop iteration**, the value of the `idx`'s element in the array is being added to the running sum.
- **After each loop iteration**, the index `idx` is being incremented.

One can traverse an array in any direction one wishes:

[19] http://www.exciton.cs.rice.edu/JavaResources/Java/loops.htm

```
// Sum the number of elements in an array of ints, myArray.
int sum = 0; // initialize the sum

for(int idx=myArray.length-1; 0<=idx; idx--) { //start idx @ length-1; end idx at 0;
                                               //decrement idx every time the loop is processed.
    sum += myArray[idx]; // add the idx'th element of myArray to the sum
}
```

The above loop sums the list just as well as the first example, but it does it from back to front. Note however, that we had to be a little careful on how we initialized the index and how we set up the termination condition.

Here's a little more complicated loop:

```
    // Find the index of the smallest element in an array of ints, myArray.
int minIdx = 0; // initialize the index. Must be declared outside the loop.

if(0==myArray.length) throw new NoSuchElementException("Empty array!"); //No good if array
                                                                         //is empty!
else {
    for(minIdx = 0, int j = 1; j<myArray.length; j++) { //start minIdx @ 0, start index @ 1;
                                                        //end index at length-1; increment
                                                        // index every time the loop is
                                                        //processed.
        if(myArray[minIdx] > myArray[j])
            minIdx = j; // found new minimum
    }
}
```

Some important things to notice about this algorithm:

- The empty case must be checked explicitly — no polymorphism to help you out here!
- The desired result index **cannot** be declared inside the for loop because otherwise it won't be visible to the outside world.
- Be careful about using the `minIdx` value if the array was indeed empty–it's an invalid value! It can't be set to a valid value because otherwise you can't tell the difference between a value that was never set and one that was.
- The `for` loop has two initialization statements separated by a comma.
- The loop does work correctly if the array only has one element, but only because the termination check is done before the loop body.
- Notice that to prove that this algorithm works properly, one must make separate arguments about the empty case, the one element case and the n-element case. Contrast this to the much simpler list algorithm that only needs an empty and non-empty cases.

For convenience, Java 5.0 now offers a compact syntax used for traversing **all** the elements of an array or of anything that subclasses type `Iterable`[20] :

```
MyType[] myArray;  // array is initialized with data somewhere

for(MyType x: myArray){
    // code involving x, i.e. each element in the array
}
```

[20] http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Iterable.html

It is important to remember that this syntax is used when one wants to process every element in an array (or an `Iterable` object) **independent of order of processing** because Java does not guarantee a traversal order.

Let's look at an algorithm where we might not want to process the entire array:

```
// Find the first index of a given value in an array

int idx = -1;  // initialize the index to an invalid value.

for(int j=0; j<myArray.length; j++) {  //no initialization ; end index at length-1;
                                       //increment index every time the loop is processed.
    if(desiredValue == myArray[j]) { // found match!
        idx = j;  // save the index.
        break;  // break out of the loop.
    }
}
```

Notes:

- The only way you can tell if the desired value was actually found or not is if the **value** of `idx` is -1 or not. Thus the value of `idx` must be checked before it is ever used.
- The resultant `idx` variable cannot be used as the index inside the loop because one would not be able to tell if the desired value was found or not unless one also checked the length of the array. This is because if the desired value was never found, `idx` at the end of the loop would equal the length of the array, which is only an invalid value if you already know the length of the array.
- The `break` statement stops the loop right away and execution resumes at the point right after end of the loop.

There is a counterpart to `break` called `continue` that causes the loop to immediately progress to the beginning of the next iteration. It is used much more rarely than `break`, usually in situations where there are convoluted and deeply nested `if-else` statements.

Can you see that the price of the compact syntax of for loops is a clear understandability of the process?

4.5.2.1 While loops

`for` loops are actually a specialized version of `while` loops. `while` loops have no special provisions for initialization or loop incrementing, just the termination condition.

`while` loops iterate through the loop body until the termination condition evaluates to a `false` value.

The following `for` loop:

```
for([initialization statement]; [termination expr] ; [increment statement]) {

    [loop body]

}
```

Is exactly equivalent to the following:

```
{
    [initialization statement];
```

```
    while([termination expr]) {

        [loop body]

        [increment statement];

    }
}
```

Note the outermost curly braces that create the scoping boundary that encapsulates any variable declared inside the `for` loop.

The Java compiler will automatically convert a `for` loop to the above `while` loop.

Here is the above algorithm that finds a desired value in an array, translated from a `for` loop to a `while` loop:

```
// Find the index of the first occurance of desiredValue in myArray, using a while loop.
{
    idx = -1;  // initialize the final result
    int j = 0; // initialize the index

    while(j < myArray.length) {   // loop through the array
        if(desiredValue == myArray[j]) {   // check if found the value
            idx = j;  // save the index
            break;   // exit the loop.
        }

        j++;   // increment the index
    }
}
```

Basically, `for` loops give up some of the flexibility of a `while` loop in favor of a more compact syntax.

`while` loops are very useful when the data is not sequentially accessible via some sort of index. Another useful scenario for `while` loops is when the algorithm is waiting for something to happen or for a value to come into the system from an outside (relatively) source.

`do-while` loops are the same as `while` loops except that the conditional statement is evaluated at the end of the loop body, not its beginning as in a `for` or `while` loop.

See the Java Resources web site page on loops[21] for more information on processing lists using while loops.

4.5.2.2 for-each loops

An exceedingly common `for`-loop to write is the following;

```
Stuff[] s_array = new Stuff[n];
// fill s_array with values

for(int i = 0; i < s_array.length; i++) {
    // do something with s_array[i]
}
```

[21] http://www.exciton.cs.rice.edu/JavaResources/Java/loops.htm

Essentially, the loop does some invariant processing on every element of the array.

To make life easier, Java implements the **for-each loop**, which is just an alternate `for` loop syntax:

```
Stuff[] s_array = new Stuff[n];
// fill s_array with values

for(Stuff s:s_array) {
    // do something with s
}
```

Simpler, eh?

It turns out that the for-each loop is not simply relegated to array. Any class that implements the `Iterable` interface will work. This is discussed in another module, as it involves the use of generics.

4.5.3 Arrays vs. Lists

In no particular order...

- Arrays:
  - Fast access to all elements.
  - Fixed number of elements held.
  - Difficult to insert elements.
  - Can run into problems with uninitialized elements.
  - Minimal safety for out-of-bounds indices.
  - Minimal memory used
  - Simple syntax
  - Must use procedural techniques for processing.
  - Often incompatible with OO architectures.
  - Difficult to prove that processing algorithms are correct.
  - Processing algorithms can be very fast.
  - Processing algorithms can be minimally memory intensive
- Lists:
  - Slow access except to first element, which is fast.
  - Unlimited number of elements held.
  - Easy to insert elements.
  - Encountering uninitialized elements very rare to impossible.
  - Impossible to make out-of-bounds errors.
  - Not optimized for memory usage.
  - More cumbersome syntax.
  - Can use OO and polymorphic recursive techniques for processing.
  - Very compatible with OO architectures.
  - Easy to prove that processing algorithms are correct.
  - Processing algorithms can be quite fast if tail-recursive and using a tail-call optimizing compiler.
  - Processing algorithms can be very memory intensive unless tail-recursive and using a tail-call optimizing compiler.

BOTTOM LINE: Arrays are optimized for size and random access speed at the expense of OO design and recursion. If you do not need speed or low memory, do not use an array. If you must use an array, tightly encapsulate it so that it does not affect the rest of your system.

4.6 Design Patterns for Sorting[22]

The following discussion is based on the the SIGCSE 2001 paper by Nguyen and Wong, "Design Patterns for Sorting"[23].

Merritt's Thesis

In 1985, Susan Merritt proposed that all comparison-based sorting could be viewed as "Divide and Conquer" algorithms.[24] That is, sorting could be thought of as a process wherein one first "divides" the unsorted pile of whatever needs to sorted into smaller piles and then "conquers" them by sorting those smaller piles. Finally, one has to take the the smaller, now sorted piles and recombines them into a single, now-sorted pile.

We thus end up with a recursive definition of sorting:

- To sort a pile:
  - Split the pile into smaller piles
  - Sort the smaller piles
  - Join the sorted smaller piles into a single pile

We can see Merritt's recursive notion of sorting as a split-sort-join process in a pictoral manner by considering the general sorting process as a "black box" process that takes an unsorted set and returns a sorted set. Merritt's thesis thus contends that this sorting process can be described as a splitting followed by a sorting of the smaller pieces followed by a joining of the sorted pieces. The smaller sorting process can thus be similarly described. The base case of this recursive process is when the set has been reduced to a single element, upon which the sorting process cannot be broken down any more as it is a trivial no-op.

Animation of the Merritt Sorting Thesis (Click the "Reveal More" button)

This media object is a Flash object. Please view or download it at
<split-join.swf>

Figure 4.10: Sorting can be seen as a recursive process that splits the unsorted items into multiple unsorted sets, sorts them and then rejoins the now sorted sets. When a set is reduced to a single element (blank boxes above), sorting is a trivial no-op.

Merritt's thesis is potentially a very powerful method for studying and understanding sorting. In addition, Merritt's abstract characterization of sorting exhibits much object-oriented (OO) flavor and can be described in terms of OO concepts.

Capturing the Abstraction

So, how do we capture the abstraction of sorting as described by Merritt? Fundamentally, we have to recognize that the above description of sorting contains two distinct parts: the **invariant** process of splitting into sub-piles, sorting the sub-piles and joining the sub-piles, and the **variant** processes of the actual splitting and joining algorithms used.

Here, we will restrict ourselves to the process of sorting an array of objects, in-place – that is, the original array is mutated from unsorted to sorted (as opposed to returning a new array of sorted values and leaving the original untouched). The `Comparator` object used to compare objects will be given to the sorter's constructor.

[22] This content is available online at <http://legacy.cnx.org/content/m17309/1.5/>.

[23] D. Nguyen and S. Wong, "Design Patterns for Sorting," SIGCSE Bulletin 33:1, March 2001, 263-267

[24] S. Merritt, "An Inverted Taxonomy of Sorting Algorithms," Comm. of the ACM, Jan. 1985, Volume 28, Number 1, pp. 96-99

Abstract Sorter Class

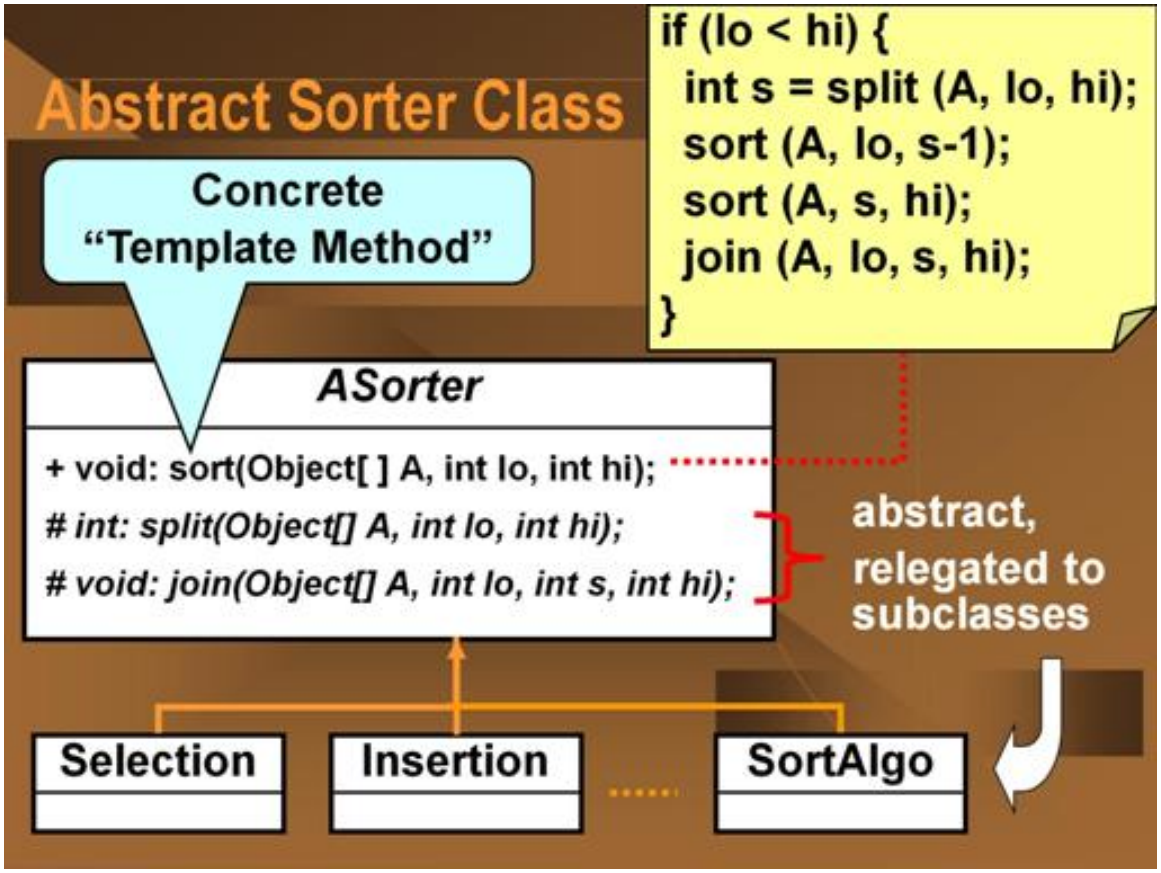


Figure 4.11: The invariant sorting process is represented as an abstract class

Here, the invariant process is represented by the concrete `sort` method, which performs the split-sort-sort-join process as described by Merritt. The variant processes are represented by the abstract `split` and `join` methods, whose exact behaviors are indeterminate at this time.

Above the methods are defined as following:

`final void sort(Object [] A, int lo, int hi)` – sorts the given unsorted array of objects, `A`, defined from index `lo` to index `hi`, inclusive. This method is implemented here and marked `final` to enforce its invariance with respect to the subclasses. It is this method that implements Merritt's split-sort-join process.

`abstract int split(Object [] A, int lo, int hi)` – splits the given unsorted array of objects, `A`, defined from index `lo` to index `hi`, inclusive, into two adjacent sub-arrays. The returned index is the index of the first element of the upper sub-array. The implementation of this abstract method is in the sub-classes.

`abstract void join(Object [] A, int lo, int s, int hi)` – joins two sorted adjacent sub-arrays of objects in the array `A`, where the lower sub-array is from index `lo` to index `s`, inclusive, and the upper

sub-array is from index `s` to index `hi`, inclusive. The implementation of this abstract method is in the subclasses.

Here's the full code for the abstract `ASorter` class, the abstract superclass for all concrete sorters and the implementation of Merritt's template for sorting: **ASorter class**

```
    package sorter;

public abstract class ASorter
{
 protected AOrder aOrder;
 /**
  * The constructor for this class.
  * @param aOrder The abstract ordering strategy to be used by any subclass.
  */
 protected ASorter(AOrder aOrder)
 {
  this.aOrder = aOrder;
 }


 /**
  * Sorts by doing a split-sort-sort-join.  Splits the original array into two subarrays,
  * recursively sorts the split subarrays, then re-joins the sorted subarrays together.
  * This is the template method.  It calls the abstract methods split and join to do
  * the work.  All comparison-based sorting algorithms are concrete subclasses with
  * specific split and join methods.
  * @param A the array A[lo:hi] to be sorted.
  * @param lo the low index of A.
  * @param hi the high index of A.
  */
 public final void sort(Object[] A, int lo, int hi)
 {
      if (lo < hi)
      {
         int s = split (A, lo, hi);
         sort (A, lo, s-1);
         sort (A, s, hi);
         join (A, lo, s, hi);
      }
 }

 /**
  * Splits A[lo:hi] into A[lo:s-1] and A[s:hi] where s is the returned value of this function.
  * @param A the array A[lo:hi] to be sorted.
  * @param lo the low index of A.
  * @param hi the high index of A.
  */
 protected abstract int split(Object[] A, int lo, int hi);

 /**
    * Joins sorted A[lo:s-1] and sorted A[s:hi] into A[lo:hi].
  * @param A A[lo:s-1] and A[s:hi] are sorted.
```

```
  * @param lo the low index of A.
  * @param hi the high index of A.
  */
 protected abstract void join(Object[] A, int lo, int s, int hi);

 /**
  * An accessor method for the abstract ordering strategy.
  * @param aOrder
  */
 public void setOrder(AOrder aOrder)
 {
  this.aOrder = aOrder;
 }
}
```

Note: `AOrder` is an abstract ordering operator whose concrete implementations define the binary ordering for the object being sorted. The examples below, only use the `AOrder.lt(Object x, Object y)` method, which returns `true` if `x` $<$ `y`. The sorting framework could easily be modified to use `java.util.Comparator` instead with no loss of generality.

Template Design Pattern

The invariant sorting process as described by Merritt is an example of the Template Method Design Pattern.

Template Method Design Pattern

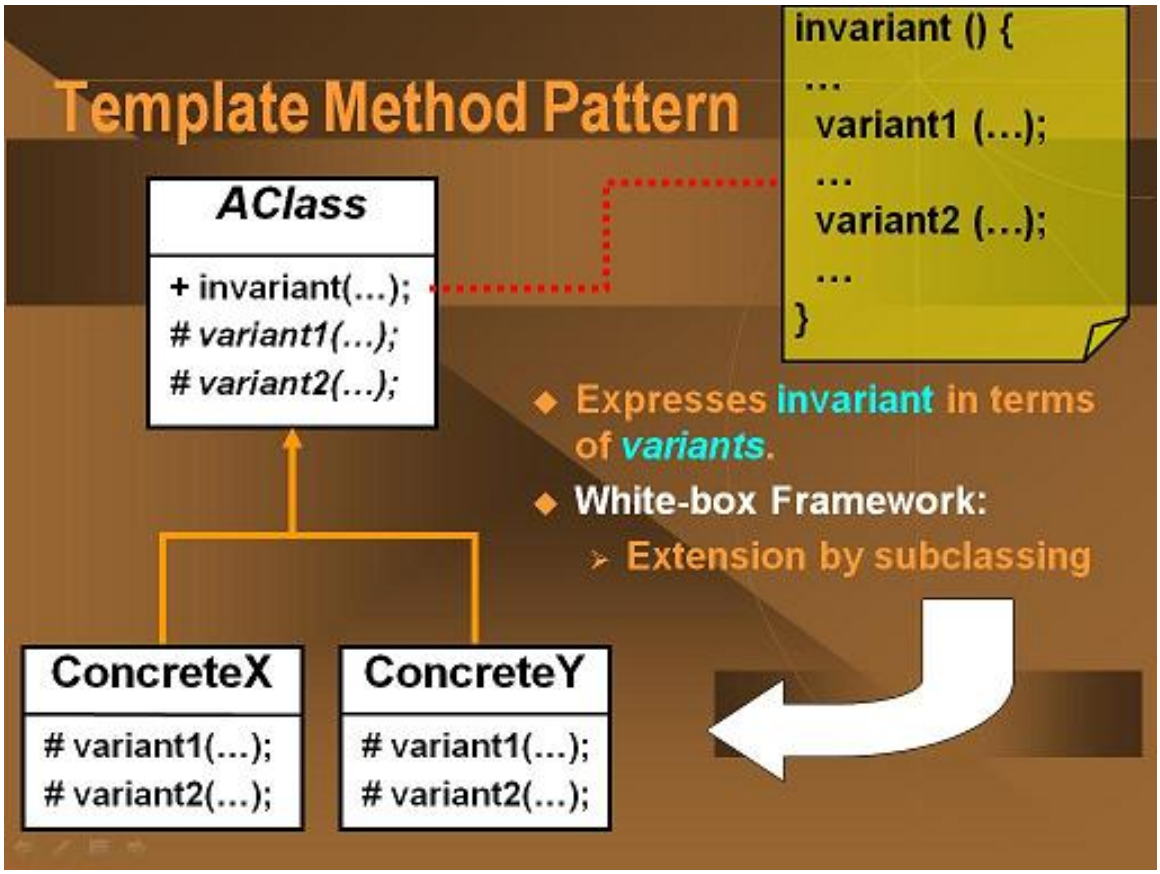


Figure 4.12: The Template Method Design Pattern describes an invariant concrete process in terms of variant, abstract methods.

Here, the invariant process is represented by a concrete method of an abstract superclass. This concrete method's implementation is in terms of abstract methods of the same class. These abstract methods represent the variant processes and are implemented in the sub-classes. This type of class organization where the variant processes are relegated to sub-classes is also known as a **white box framework**.

4.6.1 Concrete Sorters

In order to create a sorter that can actually perform a sorting operation, we need to subclass the above `ASorter` class and implement the abstract `split` and `join` methods. It should be noted that in general, the `split` and `join` methods form a matched pair. One can argue that it is possible to write a universal join methods (a merge operation) but it would be highly inefficent in most cases.

Example 4.1: Selection Sort
Tradionally, an in-place selection sort is performed by selecting the smallest (or largest) value in the array and placing it in the right-most location by either swapping it with the right-most element or by shifting all the in-between elements to the left. The selection and swapping/shifting process then repeated with the sub-array to the left of the newly placed element. This continues until only

one element remains in the array. A selection sort is commonly used to do something like a sort group of people into ascending height.

Below is an animation of a traditional selection sort algorithm:

Traditional Selection Sort Algorithm

This media object is a Flash object. Please view or download it at
<selection_sort_trad.swf>

Figure 4.13: The extrema values are removed from an ever-shrinking unordered set and placed into the resulting sorted array. Here, the smallest values are removed from the left and placed to the right in the array.

In terms of the Merritt sorting paradigm, a selection sort can be broken down into a splitting process that is the same as the above selection process and a trivial join process. Looking at the above selection and swap/shift process, we see that it is describing a the splitting off of a single element, the smallest, from an array. The process repeats recursively until there is nothing more to split off. The sorting of a single element is a no-op, so after that the recursion rolls back out though the joining process. But the joining process is trivial, a no-op, because the elements are already in their corret positions. The beauty of Merritt's insight is the realize that by considering a no-op as an operational part of a process, all the different types of binary comparison-based sorting could be unified under a common framework.

Below is an animation of a Merritt selection sort algorithm:

Merritt Selection Sort Process

This media object is a Flash object. Please view or download it at
<selection_sort_Merritt.swf>

Figure 4.14: The splitting process splits off one element at a time, the smallest element, from the left and placed to the right in the array. The join process is a no-op because the elements are already in their correct places.

The code to implement a selection sorter is straightforward. One need only implement the `split` and `join` methods where the split method always returns the `lo+1` index because the smallest value in the (sub-)array has been moved to the index `lo` position. Because the bulk of the work is being done in the splitting method, selection sort is classified as an "hard split, easy join" sorting process. In the Java implementation of the SelectionSorter class below, the `split` method splits off the extrema (minimum, here) value from the sub-array, while the `join` method is a no-op.

SelectionSorter class

```
package sorter;

/**
 * A concrete sorter that uses the Selection Sort technique.
 */
public class SelectionSorter extends ASorter
```

```
{

 /**
  * The constructor for this class.
  * @param iCompareOp The comparison strategy to use in the sorting.
  */
 public SelectionSorter(AOrder iCompareOp)
 {
  super(iCompareOp);
 }
 /**
  * Splits A[lo:hi] into A[lo:s-1] and A[s:hi] where s is the returned value of this function.
  * This method places the "smallest" value in the lo position and splits it off.
  * @param A the array A[lo:hi] to be sorted.
  * @param lo the low index of A.
  * @param hi the high index of A.
  * @return lo+1 always
  */
 protected int split(Object[] A, int lo, int hi)
 {
     int s = lo;
      int i = lo + 1;
      // Invariant: A[s] <= A[lo:i-1].
      // Scan A to find minimum:
      while (i <= hi)
      {
         if (aOrder.lt(A[i], A[s]))
            s = i;
         i++; // Invariant is maintained.
      } // On loop exit: i = hi + 1; also invariant still holds;
        // this makes A[s] the minimum of A[lo:hi].

      // Swapping A[lo] with A[s]:
      Object temp = A[lo];
      A[lo] = A[s];
      A[s] = temp;
      return lo + 1;
  }

 /**
  * Joins sorted A[lo:s-1] and sorted A[s:hi] into A[lo:hi].
  * This method does nothing.  The sub-arrays are already in proper order.
  * @param A A[lo:s-1] and A[s:hi] are sorted.
  * @param lo the low index of A.
  * @param s
  * @param hi the high index of A.
  */
 protected void join(Object[] A, int lo, int s, int hi)
 {
 }
}
```

What's interesting to note here is what is missing from the above code. A tradional selection sort aalgorithm is implemented using a nested double loop, one to find the smallest value and one to repeatedly process the ever-shrinking unsorted sub-array. Notice that the above code only has a single loop, which coresponds to the inner loop of a traditional implementation. The outer loop is embodied in the recursive nature of the sort template method in the `ASorter` superclass.

Notice also that the selection sorter implementation does not include any explicit connection between the split and join operations nor does it contain the actual `sort` method. These are all contained in the concrete `sort` method of the superclass. We describe the `SelectionSorter` class as a **component** in a **framework** (technically a "white box" framework, as described above). Frameworks display **inverted control** where the components provide **services** to the framework. The framework itself runs the algorithms, here the high level, templated sorting process, and call upon the services provided by the components to fill in the necessary processing pieces, e.g. the split and join procedures.

Example 4.2: Insertion Sort
Tradionally, an in-place insertion sort is performed by starting from one end of the arry, say the left end, and performing an in-order insertion of an element into the sub-array to its left. The next element to the right is then chosen and the insertion process repeated. At each insertion, the sorted sub-array on the left grows until encompasses the entire array. An insertion sort is a very typical way in which people will order a set of playing cards in their hand.

Below is an animation of a traditional insertion sort algorithm:

Traditional Insertion Sort Algorithm

This media object is a Flash object. Please view or download it at
<insertion_sort_trad.swf>

Figure 4.15: Starting from the left, elements from the immediate right are inserted into a growing sub-array to the left.

In the Merrit paradigm, the insertion sort first splits the array or sub-array into two pieces simply by separating the right-most element. Recursively, the splitting process proceeds to from the right to the left until a single element is left in the sub-array. Sorting a one element array is a no-op, so then the recursion unwinds with the join process. The join process combines each single split-off element with its sorted sub-array partner to its left by performing an in-order insertion. This proceeds as the recusion unwinds until the entire array is fully sorted. In contrast to the selection sort, the bulk of the work is being done in the join method, hence classifying insertion sort as an "easy split, hard join" sorting process.

Below is an animation of a Merritt insertion sort algorithm:

Merritt Insertion Sort Process

This media object is a Flash object. Please view or download it at
<insertion_sort_Merritt.swf>

Figure 4.16: The right-most elements are first split-off one by one, starting at the right and moving left. The split-off elements are then joined by performing an in-order insertion to the left, starting at the left.

In the Java implementation of the selection sorter below, the `split` method simply splits off the right-most element of the sub-array. The `join` method performs an in-order insertion of the single split-off element into the larger sub-array to its left. **InsertionSorter class**

```
package sorter;

/**
 * A concrete sorter that uses the Insertion Sort technique.
 */
public class InsertionSorter extends ASorter
{

 /**
  * The constructor for this class.
  * @param iCompareOp The comparison strategy to use in the sorting.
  */
 public InsertionSorter(AOrder iCompareOp)
 {
  super(iCompareOp);
 }
 /**
  * Splits A[lo:hi] into A[lo:s-1] and A[s:hi] where s is the returned value of this function.
  * This simply splits off the element at index hi.
  * @param A the array A[lo:hi] to be sorted.
  * @param lo the low index of A.
  * @param hi the high index of A.
  * @return hi always.
  */
 protected int split(Object[] A, int lo, int hi)
 {
  return (hi);
 }

 /**
  * Joins sorted A[lo:s-1] and sorted A[s:hi] into A[lo:hi].   (s = hi)
  * The method performs an in-order insertion of A[hi] into the A[lo, hi-1]
  * @param A A[lo:s-1] and A[s:hi] are sorted.
  * @param lo the low index of A.
  * @param s
  * @param hi the high index of A.
  */
 protected void join(Object[] A, int lo, int s, int hi)
 {
      int j = hi; // remember s == hi.
      Object key = A[hi];
      // Invariant: A[lo:j-1] and A[j+1:hi] are sorted and key < all elements of A[j+1:hi].
      // Shifts elements of A[lo:j-1] that are greater than key to the "right" to make room
      // for key.
      while (lo < j && aOrder.lt(key, A[j-1]))
      {
         A[j] = A[j-1];
```

```
         A[j-1] = key;
         j = j - 1;     // invariant is maintained.
      }   // On loop exit: j = lo or A[j-1] <= key. Also invariant is still true.
 //     A[j] = key;
 }
}
```

Exercise 4.6.1 ***(Solution on p. 132.)***

The authors were once challenged that the Merritt template-based sorting paradigm could not be used to describe the Shaker Sort process (a bidirectional Bubble or Selection sort). See for instance, http://en.wikipedia.org/wiki/Cocktail_sort[25] . However, it can be done is a very straightforward manner. There are a number of viable solutions. Hint: think about the State Design Pattern[26].

For more examples, please see download the demo code[27]. Please note that the ShakerSort code is disabled due to its use as a student exercise.

[25] http://en.wikipedia.org/wiki/Cocktail_sort
[26] "State Design Pattern" <http://legacy.cnx.org/content/m17047/latest/>
[27] See the file at <http://legacy.cnx.org/content/m17309/latest/Sorter.zip>

Solutions to Exercises in Chapter 4

Solution to Exercise 4.6.1 (p. 131)
The solution is left to the student but is available from the authors if proof of non-student status is provided.

Chapter 5

Restricted Access Containers

5.1 Restricted Access Containers[1]

5.1.1 Introduction

Stacks and queues are examples of containers with special insertion and removal behaviors and a special access behavior.

Insertion and removal in a stack must be carried out in such a way that the last data inserted is the first one to be removed. One can only retrieve and remove a data element from a stack by way of special access point called the "top". Traditionally, the insertion and removal methods for a stack are called push and pop, respectively. push inserts a data element at the top of the stack. pop removes and returns the data element at the top of the stack. A stack is used to model systems that exhibit **LIFO (Last In First Out)** insert/removal behavior.

Data insertion and removal in a queue must be carried out in such a way that the first one to be inserted is the first one to be removed. One can only retrieve and remove a data element from a queue by way of special access point called the "front". Traditionally, the insertion and removal methods for a queue are called **enqueue** and **dequeue**, respectively. enqueue inserts a data element at the "end" of the queue. dequeue removes and returns the data element at the front of the queue. A queue is used to model systems that exhibit **FIFO (First In First Out)** insertion/removal behavior. For example, one can model a movie ticket line by a queue.

We abstract the behaviors of special containers such as stacks and queues into an interface called `IRAContainer` specified as follows.

5.1.2 Restricted Access Containers

5.1.2.1 IRAContainer.java

```
    package rac;

import listFW.*;
/**
 * Defines the interface for a restricted access container.
 */
public interface IRAContainer {
    /**
     * Empty the container.
```

[1] This content is available online at <http://legacy.cnx.org/content/m17101/1.1/>.

```
     * NOTE: This implies a state change.
     * This behavior can be achieved by repeatedly removing elements from this IRAContainer.
     * It is specified here as a convenience to the client.
     */
    public void clear();
    /**
     * Return TRUE if the container is empty; otherwise, return
     * FALSE.
     */
    public boolean isEmpty();
    /**
     * Return TRUE if the container is full; otherwise, return
     * FALSE.
     */
    public boolean isFull();
    /**
     * Return an immutable list of all elements in the container.
     * @param fact for manufacturing an IList.
     */
    public IList elements(IListFactory fact);
    /**
     * Remove the next item from the container and return it.
     * NOTE: This implies a state change.
     * @throw an Exception if this IRAContainer is empty.
     */
    public Object get();
    /**
     * Add an item to the container.
     * NOTE: This implies a state change.
     * @param input the Object to be added to this IRAContainer.
     * @throw an Exception if this IRAContainer is full.
     */
    public void put(Object input);
    /**
     * Return the next element in this IRAContainer withour removing it.
     * @throw an Exception if this IRAContainer is empty.
     */
    public Object peek();
}
```

1. Restrict the users from seeing inside or working on the inside of the container.
2. Have simple put(data) and get() methods. Note the lack of specification of how the data goes in or comes out of the container.
3. However, a "policy" must exist that governs how data is added ("put") or removed ("get"). Examples:
   - First in/First out (FIFO) ("Queue")
   - Last in/First out (LIFO) ("Stack")
   - Retrieve by ranking ("Priority Queue")
   - Random retrieval
4. The policy is variant behavior –> abstract it.

- The behavior of the RAC is independent of exactly what the policy does.
- The RAC delegates the actual adding ("put") work to the policy.
- The RAC is only dependent on the existence of the policy, not what it does.
- The policy is a "strategy" for adding data to the RAC. See the Strategy design pattern[2] .
- Strategy pattern vs. State pattern[3] – so alike, yet so different!

The manufacturing of specific restricted access containers with specific insertion strategy will be done by concrete implementations of the following abstract factory interface.

5.1.2.2 IRACFactory.java

```
    package rac;

/**
 * Abstract Factory to manufacture RACs.
 */
public interface IRACFactory {
    /**
     * Returns an empty IRAContainer.
     */
    public IRAContainer makeRAC();
}
```

5.1.3 Examples

The following is an (abstract) implementation of `IRACFactory` using `LRStruct` as the underlining data structure. By varying the insertion strategy, which is an IAlgo on the internal `LRStruct`, we obtain different types of RAC: stack, queue, random, etc.

[2] http://cnx.org/content/m17037/latest/
[3] http://cnx.org/content/m17047/latest/

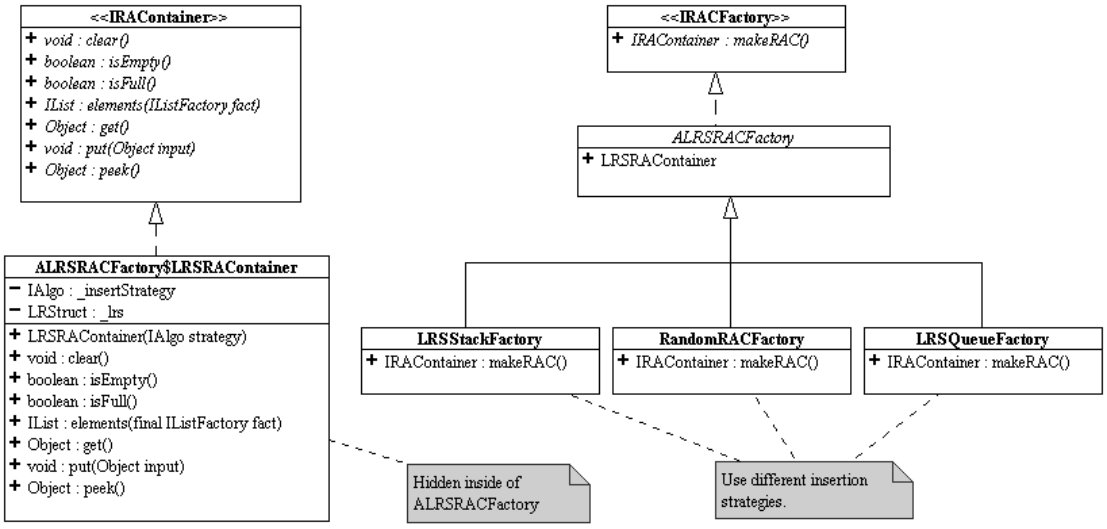


Figure 5.1: UML diagram of the abstract RAC and RAC factory definitions plus a few concrete RAC factories.

The source code for the following examples can be downloaded at this link[4] .

5.1.3.1 ALRSRACFactory.java

```
    package rac;

import listFW.*;
import listFW.factory.*;
import lrs.*;

/**
 * Implements a factory for restricted access containers.  These
 * restricted access containers are implemented using an LRStruct to
 * hold the data objects.
 */
public abstract class ALRSRACFactory implements IRACFactory {
    /**
     * Implements a general-purpose restricted access container using
     * an LRStruct.  How?
     *
     * The next item to remove is always at the front of the list of
     * contained objects.  This is invariant!
     *
     * Insertion is, however, delegated to a strategy routine; and
```

[4] http://legacy.cnx.org/content/m17101/latest/rac.zip

```
 * this strategy is provided to the container.  This strategy
 * varies to implement the desired kind of container, e.g., queue
 * vs. stack.
 *
 * This nested static class is protected so that classes derived from its
 * factory can reuse it to create other kinds of restricted access
 * container.
 */
protected static class LRSRAContainer implements IRAContainer {
    private IAlgo _insertStrategy;
    private LRStruct _lrs;

    public LRSRAContainer(IAlgo strategy) {
        _insertStrategy = strategy;
        _lrs = new LRStruct();
    }

    /**
     * Empty the container.
     */
    public void clear() {
        _lrs = new LRStruct();
    }

    /**
     * Return TRUE if the container is empty; otherwise, return
     * FALSE.
     */
    public boolean isEmpty() {
        return (Boolean)_lrs.execute(CheckEmpty.Singleton);
    }

    /**
     * Return TRUE if the container is full; otherwise, return
     * FALSE.
     *
     * This implementation can hold an arbitrary number of
     * objects.  Thus, always return false.
     */
    public boolean isFull() {
        return false;
    }

    /**
     * Return an immutable list of all elements in the container.
     */
    public IList elements(final IListFactory fact) {
        return (IList)_lrs.execute(new IAlgo() {
            public Object emptyCase(LRStruct host, Object... nu) {
                return fact.makeEmptyList();
            }
```

```
                public Object nonEmptyCase(LRStruct host, Object... nu) {
                    return fact.makeNEList(host.getFirst(),
                                           (IList)host.getRest().execute(this));
                }
            });
        }

        /**
         * Remove the next item from the container and return it.
         */
        public Object get() {
            return _lrs.removeFront();
        }

        /**
         * Add an item to the container.
         */
        public void put(Object input) {
            _lrs.execute(_insertStrategy, input);
        }

        public Object peek() {
            return _lrs.getFirst();
        }
    }
}

/**
 * Package private class used by ALRSRACFactory to check for emptiness of its internal LRStruct.
 */
class CheckEmpty implements IAlgo {
    public static final CheckEmpty Singleton= new CheckEmpty();
    private CheckEmpty() {
    }

    public Object emptyCase(LRStruct host, Object... input) {
        return Boolean.TRUE;
    }

    public Object nonEmptyCase(LRStruct host, Object... input) {
        return Boolean.FALSE;
    }
}
```

5.1.3.2 LRSStackFactory.java

```
package rac;

import lrs.*;
```

```
public class LRSStackFactory extends ALRSRACFactory {
    /**
     * Create a ``last-in, first-out'' (LIFO) container.
     */
    public IRAContainer makeRAC() {
        return new LRSRAContainer(new IAlgo() {
            public Object emptyCase(LRStruct host, Object... input) {
                return host.insertFront(input[0]);
            }

            public Object nonEmptyCase(LRStruct host, Object... input) {
                return host.insertFront(input[0]);
            }
        });
    }
}
```

5.1.3.3 LRSQueueFactory.java

```
package rac;

import lrs.*;

public class LRSQueueFactory extends ALRSRACFactory {
    /**
     * Create a ``first-in, first-out'' (FIFO) container.
     */
    public IRAContainer makeRAC() {
        return new LRSRAContainer(new IAlgo() {
            public Object emptyCase(LRStruct host, Object... input) {
                return host.insertFront(input[0]);
            }

            public Object nonEmptyCase(LRStruct host, Object... input) {
                return host.getRest().execute(this, input);
            }
        });
    }
}
```

5.1.3.4 RandomRACFactory.java

```
package rac;

import lrs.*;

/*
```

```
 * Implements a factory for restricted access containers, including a
 * container that returns a random item.
 */
public class RandomRACFactory extends ALRSRACFactory {
    /**
     * Create a container that returns a random item.
     */
    public IRAContainer makeRAC() {
        return new LRSRAContainer(new IAlgo() {
            public Object emptyCase(LRStruct host, Object... input) {
                return host.insertFront(input[0]);
            }

            public Object nonEmptyCase(LRStruct host, Object input) {
                /*
                 * Math.Random returns a value between 0.0 and 1.0.
                 */
                if (0.5 > Math.random())
                    return host.insertFront(input[0]);
                else
                    return host.getRest().execute(this, input);
            }
        });
    }
}
```

But can we push the abstraction further? Is the difference between a stack and a queue really anything more than how the data is ordered?

Now, let's go on an look at the ordering object and priority queues... [5]

5.2 Ordering Object and Priority Queue[6]

5.2.1 Order Relation

When comparing two objects, there are two ways to look at the comparison behavior, which creates a notion of ordering between the two objects:

A. **Each object knows how to compare itself** – Comparison is considered intrinsic to the object and is part of its behavior.
B. **Comparison is done by an external object** – A third object is introduced that contains the ability to compare two objects.

These two different outlooks on binary object ordering will be explored below in terms of the two Java interfaces that are used to model them.

5.2.1.1 `java.lang.Comparable`[7]

There are many computing tasks that require performing some sort of comparison between data objects. A few data types are endowed with a "natural" ordering of their values. The integers have a natural ordering "less or equal to", labeled "<=", defined as follows.

[5] http://cnx.org/content/m17064/latest/
[6] This content is available online at <http://legacy.cnx.org/content/m17064/1.4/>.
[7] http://java.sun.com/j2se/1.4/docs/api/java/lang/Comparable.html

n <= m iff m = n + k, for some non-negative integer k

(**Note**: a rigorous mathematical definition of the set of non-negative integers is beyond the scope of this lecture).

The above natural order of the integers is a concrete instance of an abstract concept called an order relation. An order relation on a set S is a boolean function R on S x S that is

- reflexive: R(x, x) is true for all x in S,
- anti-symmetric: R(x,y) and R(y,x) implies x = y, for all x, y in S, and
- transitive: R(x, y) and R(y, z) implies R(x, z), for all x, y, z in S.

To model order relations that are naturally endowed in certain types of data objects, Java provides an interface called `Comparable`[8] , which has exactly one method called
`int compareTo(Object rhs)`, defined abstractly as

- `x.compareTo(y)` $<$ `0` means `x is "less than" y`,
- `x.compareTo(y) == 0` means `x is "equal to" y`, and
- `x.compareTo(y)` $>$ `0` means `y is "less than" x`.

For example, the `Integer` class implements the `Comparable` interface as follows. If `x` and `y` are `Integer` objects then,

- `x.compareTo(y)` $<$ `0` means `x` $<$ `y`,
- `x.compareTo(y) == 0` means `x == y`, and
- `x.compareTo(y)` $>$ `0` means `y` $<$ `x`.

Common data types that have a natural ordering among their values, such as `Double`, `String`, `Character`, all implement `Comparable`.

Advantages and disadvantages

- Advantages
  - Objects carry their ability to be compared with them and thus can be involved in comparison operations anywhere.
  - The comparison operation is invariant and thus consistent in all situations.
- Disadvantages
  - Comparison operation cannot be changed to match different situations.

5.2.1.2 `java.util.Comparator` [9]

Most of the time, the ordering among the data objects is an extrinsic operation imposed on the object by the user of the objects. For example, the Pizza objects in homework 2[10] have no concepts of comparing among themselves, however, the user can impose an ordering on them by comparing their price/area ratios or their profits. To model extrinsic ordering relation, Java provides an interface in the `java.util` package called `Comparator`[11] , which has exactly two methods:

- *`int compare(Object x, Object y)`*, to model the ordering
  - `compare(x, y)` $<$ `0` means `x is "less than" y`,
  - `compare(x, y) == 0` means `x is "equal to" y`, and

[8] http://java.sun.com/j2se/1.4/docs/api/java/lang/Comparable.html
[9] http://java.sun.com/j2se/1.4/docs/api/java/util/Comparator.html
[10] http://www.owlnet.rice.edu/~comp201/08-spring/assignments/hw02
[11] http://java.sun.com/j2se/1.4/docs/api/java/util/Comparator.html

  - `compare(x, y) > 0` means `yis "less than" x`, and

- *`boolean equals(Object x)`*, to model equality of `Comparators`. Unlike the `equals` method of most objects, equality of `Comparators` also requires that their comparison behavior be identical.

In most applications, when we implement a `Comparator` interface, we only need to override the `compare(...)` method and simply inherit the `equals(...)` method from `Object`.

Advantages and disadvantages

- Advantages
  - Object ordering can be considered variant and be redefined for different situations.
  - Creating composite comparison behavior can be easily acheived by decorating or other composite techniques.
- Disadvantages
  - Objects are not intrinsically comparable, requiring the inclusion of an otherwise disconnected entity in the system.

Exercise 5.2.1
Implement a `Comparator` class that will compare `Integer` objects.

5.2.2 In-Order Insertion Example

Recall the problem of inserting an `Integer` object in order into a sorted list of `Integers`. Instead of writing an algorithm that only works for `Integer`, we can write an algorithm that will work for any `Object` that can be compared by some `Comparator`. The table below contrasts the insert in order algorithm for Integer and the insert in order algorithm that uses a `Comparator` as a strategy (as in strategy pattern) for comparison. Both are algorithms for `LRStruct`, the mutable list.

First, let's look at a regular in-order insertion algorithm that works only on `Integer` objects because the comparison technique is hard-coded in:

`InsertInOrderLRS` **Visitor class**

```
import lrs.*;

public class InsertInOrderLRS implements IAlgo {

  public static final InsertInOrderLRS Singleton = new InsertInOrderLRS();

  private InsertInOrderLRS() {
  }

  /**
  * Simply inserts the given parameter n at the front.
  * @param host an empty LRStruct.
  * @param n an Integer to be inserted in order into host.
  * @return LRStruct
  */
  public Object emptyCase(LRStruct host, Object... n) {
    return host.insertFront(n[0]);
  }
```

```
  /**
  * Based on the comparison between first and n,
  * inserts at the front or recurs!
  * @param host a non-empty LRStruct.
  * @param n an Integer to be inserted in order into host.
  * @return LRStruct
  */
  public Object nonEmptyCase(LRStruct host, Object... n) {

    if (n[0] < (Integer)host.getFirst();) {  // could use Integer.compareTo()
      return host.insertFront(n[0]);
    }
    else {
      return host.getRest().execute(this, n[0]);
    }
  }
}
```

A visitor to an LRStruct that performs an in-order insertion, but only for Integer objects.

Now let's look at how the simple substitution of a Comparator for the hard-coded comparison allows the otherwise identical code to be used for any objects. All one has to do is to supply the desired Comparator instance that matches the type of objects stored in the LRStruct. Here is the implementation of a visitor to an LRStruct that performs an in-order insertion, for any objects that are comparable with the given Comparator:

InsertInOrder Visitor class

```
import lrs.*;
import java.util.*;

public class InsertInOrder implements IAlgo {

  private Comparator _order;

  public InsertInOrder(Comparator ord) {
      _order = ord;
  }

  /**
  * Simply inserts the given parameter n at the front.
  * @param host an empty LRStruct.
  * @param n an Object to be inserted in order into host,
  *   based on the given Comparator.
  * @return LRStruct
  */
  public Object emptyCase(LRStruct host, Object... n) {
    return host.insertFront(n[0]);
  }

  /**
  * Based on the comparison between first and n,
  * inserts at the front or recurs!
  * @param host a non-empty LRStruct.
```

```
  * @param n an Object to be inserted in order into host,
  *   based on the given Comparator.
  * @return LRStruct
  */
  public Object nonEmptyCase(LRStruct host, Object... n) {
    if (_order.compare(n[0], host.getFirst()) < 0) {
      return host.insertFront(n[0]);
    }
    else {
      return host.getRest().execute(this, n[0]);
    }
  }
}
```

5.2.3 3. Priority Queue

The `InsertInOrder` algorithm given in the above can be used as part of a strategy for a Restricted Access Container (Section 5.1) ("RAC") called **priority queue**. In the code below, it is assumed that "smaller" objects have higher priority, i.e. will be retrieved from the RAC before "larger" objects. The code also shows how, if the RAC elements are `Comparable` objects, that a `Comparator` could be used to provide their ordering in the RAC, even to the point of reversing their order if desired.

`PQComparatorRACFactory` priority queue implementation

```
    package rac;

import lrs.*;
import lrs.visitor.*;
import java.util.*;

/*
 * Implements a factory for restricted access containers that
 * return the ‘‘highest priority’’ item.
 */
public class PQComparatorRACFactory extends ALRSRACFactory {

    private Comparator _comp;

    /**
     * Used when the items in the container are Comparable objects.
     */
    public PQComparatorRACFactory() {
        _comp = new Comparator() {
            public int compare(Object x, Object y) {
                /*
                 * Intentionally reverse the ordering so that the
                 * largest item will be first, just to show that it can be done.
                 */
                return ((Comparable)y).compareTo(x);
            }
        };
    }
```

```
    /**
     * Used when we want to prioritize the items according to a given Comparator.
     * @param comp the item that is smallest according to comp has the highest
     * priority.
     */
    public PQComparatorRACFactory(Comparator comp) {
        _comp = comp;
    }

    /**
     * Create a container that returns the item with the highest priority
     * according to a given Comparator.
     */
    public IRAContainer makeRAC() {
        return new LRSRAContainer(new InsertInOrder(_comp));
    }
}
```

A priority queue can easily be implemented from a RAC by using a Comparator.

Chapter 6

GUI Programming

6.1 Graphical User Interfaces in Java[1]

6.1.1 Graphical User Interfaces in Java

In Java **Graphical User Interface** (**GUI**) programming, we do not build GUI components from scratch. Instead we use GUI components provided to us by the JDK. Java has two types of GUI applications: stand-alone GUI applications and **applets**. We first study how to build stand-alone GUI applications (GUI app for short).

Every GUI app uses what is called a `JFrame` that comes with the JDK. A `JFrame` is a window with borders, a title bar, and buttons for closing and maximizing or minimizing itself. It also knows how to resize itself. Every GUI app subclasses `JFrame` in some way to add more functionality to the window frame. One common task is to tell the system to terminate all "threads" of the GUI app when the user clicks on the exit (close) button of the main GUI window. We encapsulate this task in an abstract frame class called `AFrame` described below.

6.1.2 0. Abstract Frame (AFrame.java)

NOTE: The source code for `AFrame` is available at the end of this section.

6.1.2.1 Event

When the user interacts with a GUI component such as clicking on it or holding the mouse down on it and drag the mouse around, the Java Virtual Machine (JVM) fires appropriate "events" and delivers them to the GUI component. It is up to the GUI component to respond to an event. The abstract notion of events is encapsulated in an abstract class called `AWTEvent` provided by Java. Specific concrete events are represented by appropriate concrete subclasses of `AWTEvent`.

For example, when the user clicks on the close button of a `JFrame`, the JVM fires a window event represented by the class `WindowEvent` and delivers it to the `JFrame`. By default, the `JFrame` simply hides itself from the screen while everything else that was created and running before the `JFrame` disappears from the screen is still alive and running! There is no way the user can redisplay the frame again. In the case when the `JFrame` is the main window of a GUI app, we should terminate everything when this main frame is closed. The best way to ensure this action is to "register" a special window event "listener" with the `JFrame` that will call the System class to terminate all threads related to the current program and exit the program. The code looks something like this:

[1] This content is available online at <http://legacy.cnx.org/content/m17185/1.5/>.

```
        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });
```

Every time we write the constructor for a main frame of a GUI app, we **invariably** find ourselves writing the above lines of code to exit the program plus some additional application specific code to initialize the frame. So, instead of "copy-and-paste" the above code ("opportunistic" re-use), we capture this **invariant** task in the constructor of an abstract class called `AFrame` and reuse the code by subclassing from it. The application specific code to initialize the main frame is relegated to the specific concrete subclass of `AFrame` and is represented by an abstract method called `initialize()`. The constructor for `AFrame` calls `initialize()`, which, at run-time, is the concrete initialization code of the concrete subclass of `AFrame` that is being instantiated, as shown below.

```
public AFrame(String title) {
    // Always call the superclass's constructor:
    super(title);

    addWindowListener(new java.awt.event.WindowAdapter() {
        public void windowClosing(java.awt.event.WindowEvent e) {
            System.exit(0);
        }
    });

    initialize();
}
```

NOTE: See the full code listing at the end of this section for more information

6.1.2.2 Template Method Pattern: Expressing Invariant Behavior in terms of Variant Behaviors

NOTE: The code for the classes `AFrame`, `FrameA`, `FrameAB`, and `FrameAC` are available at the end of this section.

The code for `AFrame` is an example of what is called the Template Method Pattern[2]. This design pattern is used to expresses an **invariant** and concrete behavior that consists of calls to one or more **abstract** methods. An abstract method represents a **variant** behavior. In other words, the template design pattern is a means to express an invariant behavior in terms of variant behaviors. We shall see this pattern used again in expressing sorting algorithms.

6.1.2.2.1 Caveat

As stated earlier, the call to `initialize()` in the constructor of `AFrame` will only invoke the concrete `initialize()` method of the concrete descendant class of `AFrame` that is being instantiated. Because `initialize()` is polymorphic, care must taken to ensure proper initialization of descendant classes that are

[2] "Template Design Pattern" <http://legacy.cnx.org/content/m17188/latest/>

more than one level deep in the inheritance hierarchy of `AFrame`. Consider the following inheritance tree as illustrated by the following UML class diagram.

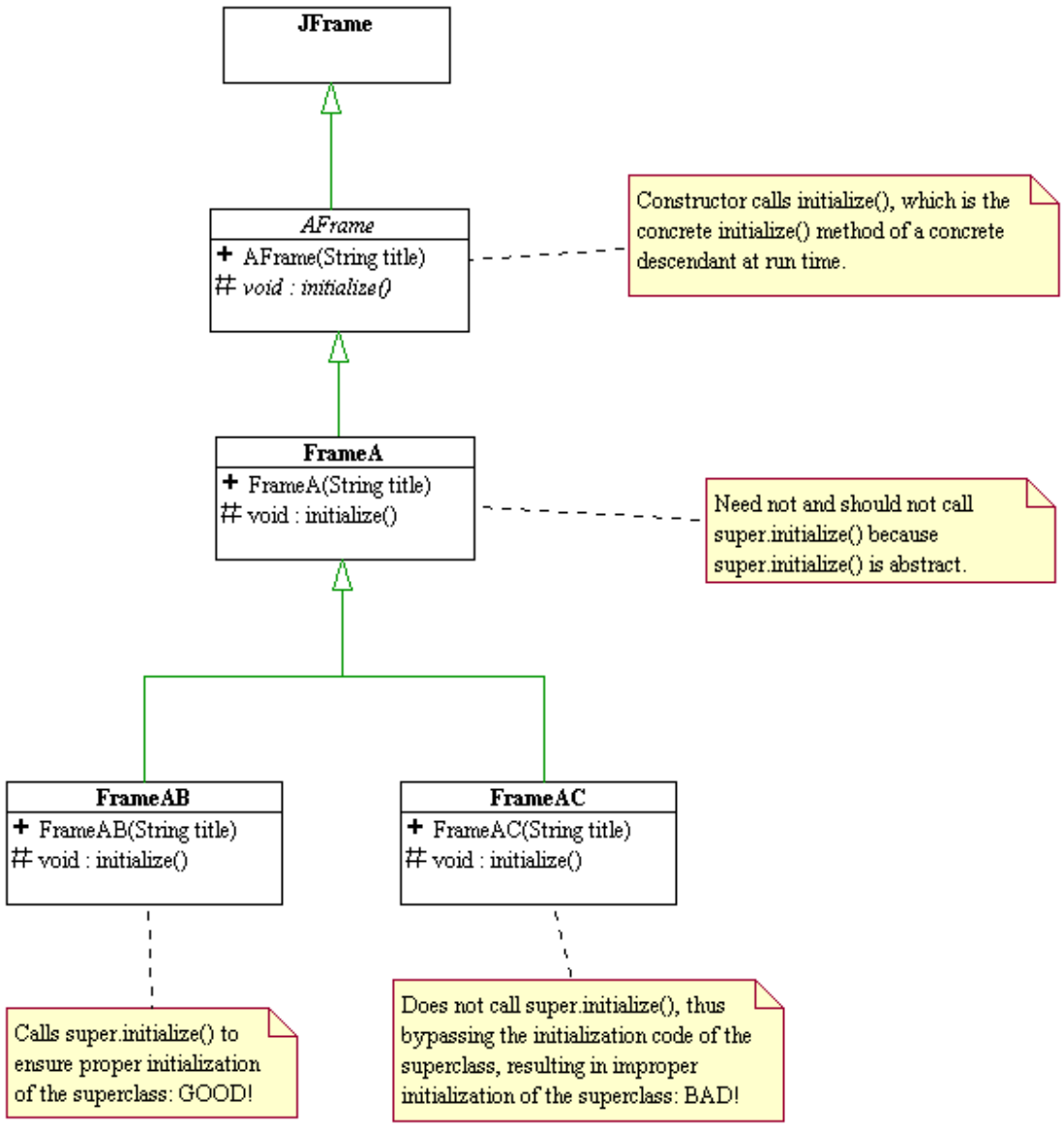


Figure 6.1

In overriding the `initialize()` method of `FrameA`, a direct subclass of `AFrame`, we should not invoke the `initialize()` method of the superclass `AFrame` via the call `super.initialize()` because `super.initialize()` is abstract. However, the `initialize()` method of any subclass of `FrameA` and below

should make a call to the initialize() method of its direct superclass in order to ensure proper initialization. For example, in the above diagram, `new FrameAB("ab")` calls the constructor `FrameAB`, which calls the constructor `FrameA`, which calls the constructor `AFrame`, which calls

- the constructor `JFrame` and then calls
- `initialize()`, which is the `initialize()` method of `FrameAB` which calls
  - `super.initialize()`, which should properly initialize `FrameA`, and then calls
  - the additional initialization code for `FrameAB`.

Exercise 6.1.1
What is the chain of calls when we instantiate a `FrameAC`?

6.1.2.3 Source code for the above frame classes

AFrame.java

```
package view;

import javax.swing.*;  // to use JFrame.

/**
 * Minimal reusable code to create a JFrame with a title and a window event
 * listener to call on System.exit() and force termination of the main
 * application that creates this JFrame.
 * @author D.X. Nguyen
 */
public abstract class AFrame extends JFrame {

    public AFrame(String title) {
        // Always call the superclass's constuctor:
        super(title);

        /**
        * Add an anonymous WindowAdapter event handler to call System.exit to
        * force termination of the main application when the Frame closes.
        * Without it, the main application will still be running even after
        * the frame is closed.
        * For illustration purpose, we use the full package name for
        * WindowAdpater and WindowEvent.
        * We could have imported java.awt.event.* and avoid using the full
        * package names.
        */
        addWindowListener(new java.awt.event.WindowAdapter() {
            public void windowClosing(java.awt.event.WindowEvent e) {
                System.out.println(e);  // For illustration purpose only.
                System.exit(0);
            }
        });
```

```
        /**
        * Subclasses are to do whatever is necessary to initialize the frame.
        * CAVEAT: At run-time, when a concrete descendant class of AFrame is
        * created, only the (concrete) initialize() method of this descendant
        * class is called.
        */
        initialize();
    }

    /**
     * Relegates to subclasses the responsibility to initialize the system to
     * a well-defined state.
     */
    protected abstract void initialize();
}
```

FrameA.java

```
package view;

/**
 * A direct concrete subclass of AFrame with its own initializaion code.
 */
public class FrameA extends AFrame {

    /**
     * Calls AFrame constructor, which calls the cocnrete inititialize() method
     * of this FrameA.  This is done dynamically at run-time when a FrameA
     * object is instantiated.
     */
    public FrameA(String title) {
        super(title);
    }

    /**
     * The superclass initialize() method is abstract: don't call it here!
     * Just write application specific initialization code for this FrameA here.
     */
    protected void initialize() {
        // Application specific intialization code for FrameA goes here...
    }
}
```

FrameAB.java

```
package view;
```

```
/**
 * A second level descendant class of AFrame
 * that initializes itself by calling the
 * initialize() method of its direct superclass
 * in addition to its own initialization code.
 */
public class FrameAB extends FrameA {

  /**
  * Calls the superclass constructor, FrameA,
  * which in turns calls its superclass constructor,
  * AFrame, which first calls its superclass
  * constructor, JFrame, and then executes the
  * initialize() method of this FrameAB.
  */
  public FrameAB(String title) {
    super(title);
  }

  /**
  * At run-time, when the constructor AFrame is
  * executed, it calls this FrameAB initialize()
  * method and NOT the initialize() method of the
  * superclass FrameA.  In order to reuse the
  * initialization code for FrameA, we must make
  * the super call to initialize().
  */
  protected void initialize() {
    // Call initialize() of FrameA:
    super.initialize();

    /**
    * Additional application specific intialization
    * code for FrameAB goes here...
    */
  }
}
```

FrameAC.java

```
package view;

/**
 * A second level descendant class of AFrame
 * that bypasses the initialize() method of
 * its direct superclass in its own
 * initialization code.  Since proper
 * initialization of a subclass depends on
 * proper initialization of the super class,
 * bypassing the initialization code of the
```

```
 * superclass is a BAD idea!
 */

public class FrameAC extends FrameA {

  /**
  * Calls the superclass constructor, FrameA,
  * which in turns calls its superclass constructor,
  * AFrame, which first calls its superclass
  * constructor, JFrame, and then executes the
  * initialize() method of this FrameAC.
  */
  public FrameAC(String title) {
    super(title);
  }

  /**
  * At run-time, when the constructor AFrame is
  * executed, it calls this initialize() method and
  * NOT the initialize() method of the superclass
  * FrameA.  This method does not call the super
  * class's initialize() method. As a result, the
  * initialization done in the superclass is
  * bypassed completely.  This is a BAD idea!
  */
  protected void initialize() {
    /**
    * Application specific intialization code for
    * FrameAC goes here...
     */
  }
}
```

6.1.3 1. Simple JFrame (Frame0.java)

NOTE: Frame0App.java is available in the source code archive file[3].

`Frame0App.java` represents one of the simplest GUI application in Java: it simply pops open an `Frame0` object. `Frame0` is subclass of `AFrame` that does nothing in its concrete `initialize()` method.

6.1.4 2. JFrame with JButtons but no event handlers (Frame1.java)

NOTE: `Frame1.java` is available in the source code archive file[4].

To display other GUI components on a `JFrame`, we first need to get the content pane of the `JFrame` and then add the desired GUI components to this content pane.

If we want to arrange the added GUI components in certain ways, we need to add an appropriate "**layout manager**" to the `JFrame`. The task of laying out the GUI component inside of a container GUI component

[3]See the file at <http://legacy.cnx.org/content/m17185/latest/GUI1.zip>
[4]See the file at <http://legacy.cnx.org/content/m17185/latest/GUI1.zip>

is specified by an interface called `LayoutManager`. In Frame1, we add the `FlowLayout` that arranges all GUI components in a linear fashion. If we do not add any layout manager to the `JFrame`, it has no layout manager and does a very bad job of arranging GUI components inside of it. As an exercise, comment out the code to add the `FlowLayout` to Frame1 and see what happens!

6.1.4.1 Strategy Pattern

`JFrame` is not responsible for arranging the GUI components that it contains. Instead it delegates such a task to its layout manager, `LayoutManager`. There are many concrete layout managers: `FlowLayout`, `BorderLayout`, `GridLayout`, etc. that arrange the internal components differently. There are said to be different strategies for layout. The interaction between `JFrame` and its layout manager is said to follow the Strategy Pattern[5].

The strategy pattern is powerful and important design pattern. It is based on that principle of delegation that we have been applying to model many of the problems so far. It is a mean to delineate the invariant behavior of the context (e.g. `JFrame`) and the variant behaviors of a union of algorithms to perform certain common abstract task (e.g. `LayoutManager`). We shall see more applications of the strategy design pattern in future lessons.

At this point the only the buttons in the `Frame1` example do not perform any task besides "blinking" when they are clicked upon. The example in the next lecture will show how to associate an action to the click event.

Click here to download code samples.[6]

6.2 More Java GUI Programming[7]

6.2.1 1. JFrame with JButtons and event handlers (Frame2.java)

NOTE: Source code for `Frame2.java` and the associated `Frame2App.java` is at the end of this module, or download the entire archive[8]

6.2.1.1 Command Design Pattern

When a user clicks on a button, an `ActionEvent` object is delivered to the button. In order for the button to perform an application specific task, we must register an `ActionEventListener` to the button and program this event listener to perform the task we want. The `ActionListener` class is an example of what is called a "**command**" from the Command Design Pattern[9] . The code to add an `ActionListener` to a button looks like:

```
    myButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        // code to perform a task goes here...
    }
});
```

The example `Frame2` (source code at end of the module) shows how to dynamically change the layout manager for a `JFrame` by calling its `setLayout(...)` method resulting in a dynamic re-arrangement of the GUI components in the `JFrame`.

[5] "Strategy Design Pattern" <http://legacy.cnx.org/content/m17037/latest/>
[6] See the file at <http://legacy.cnx.org/content/m17185/latest/GUI1.zip>
[7] This content is available online at <http://legacy.cnx.org/content/m17186/1.3/>.
[8] See the file at <http://legacy.cnx.org/content/m17186/latest/GUI2.zip>
[9] http://cnx.org/content/m17187/latest/

6.2.2 2. JFrame with JButtons and Adapters (Frame3.java)

NOTE: Source code for `Frame3.java` and the associated `IView2World.java`, `Frame3controller.java` and `Frame3App.java` is at the end of this module, or download the entire archive[10]

`Frame3` illustrates how to decouple the view from the rest of the world by having the view communicate to an interface called "adapter". A controller object, `Frame3Controller` connects the view `Frame3` to the world by installing a concrete adapter into the view. The adapter is instantiated as anonymous inner object and has access to all of its outer object. The view does not and should not know anything about the world to which it is connected. This adds flexibility to the design.

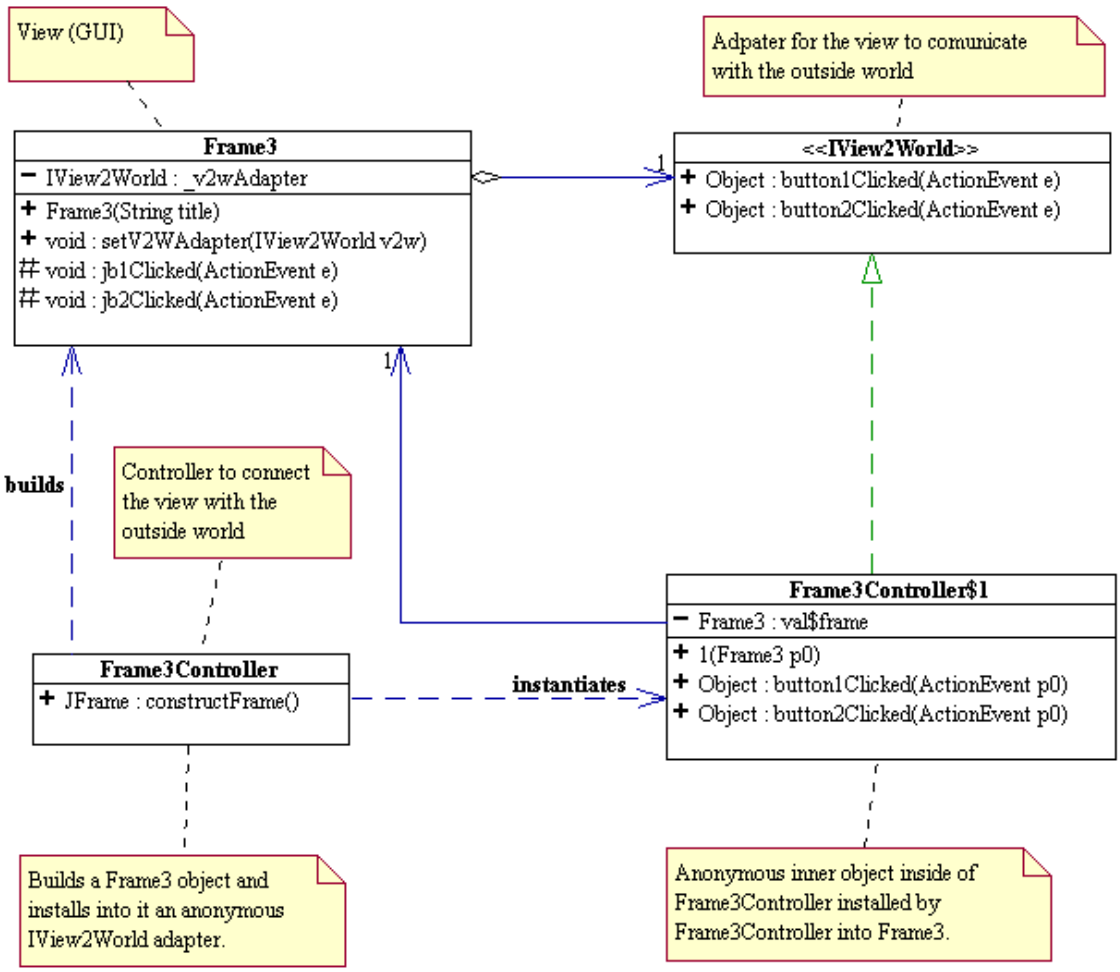


Figure 6.2

[10] See the file at <http://legacy.cnx.org/content/m17186/latest/GUI2.zip>

6.2.2.1 Null-Object Pattern

In much of the current programming practice, the special value `null` is often used as a flag to represent a gamut of different and often disconnected concepts such as emptiness and falseness. This can result in a lot of confusion and complex control structures in the code. In our view, `null` should only be used to denote the non-existence of an object. `null` is not an object and has no "intelligence" (i.e. behavior) for other objects to interact with. Therefore, whenever we work with a **union** of objects and discover that one or more of these objects may be referencing a `null` value, we almost always have to write code to distinguish `null` as a special case. To avoid this problem, we should think of adding a special object, called the **null object**, to the **union** with appropriate behavior that will allow us to treat all object references in a consistent and uniform way, devoid of special case consideration. This design pattern is called the null object pattern. We have used the null object pattern in one occasion: the `EmptyNode` to represent the empty state of a (mutable) list (`LRStruct`).

In `Frame3`, the view adapter, `_v2wAdapter`, is initialized to an (anonymous) `IView2World` object. It is there to guarantee that `_v2wAdapter` is always referencing a concrete `IView2World` object, and , since `setV2WAdapter(...)` only allows a non-null assignment, we can always call on `_v2WAdapter` to perform any method without checking for `null`.

6.2.3 Source code for above classes

Frame2.java

```
package view;

import java.awt.*;       // to use Container.
import java.awt.event.*; // to use WindowAdpater and WindowEvent.
import javax.swing.*;    // to use JFrame.

/**
 * A subclass of AFrame containing two JButtons that have event handler called
 * "commands" to dynamically switch layout managers when the buttons are clicked
 * upon.
 * @author D.X. Nguyen
 */
public class Frame2 extends AFrame {

    public Frame2(String title) {
        super(title);
    }

    protected void initialize() {
        Container cp = getContentPane();
        cp.setLayout(new FlowLayout());
        JButton jb1 = new JButton("Grid Layout");
        JButton jb2 = new JButton("Flow Layout");
        cp.add(jb1);
        cp.add(jb2);
        pack();

        /**
         * Registers an anonymous event handler for the clicking of button jb1.
         * When jb1 is clicked upon, this event handler will respond by
```

```
         * executing its actionPerformed(...) method.
         */
        jb1.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                jb1Clicked(e);
            }
        });

        /**
         * Same as the above.
         */
        jb2.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                jb2Clicked(e);
            }
        });
    }

    /**
     * Dynamically changes to a GridLayout with 2 rows and 1 column.
     */
    protected void jb1Clicked(ActionEvent e) {
        System.out.println("Set GridLayout...");
        getContentPane().setLayout(new GridLayout(2,1));
        validate();  // forces this frame to re-layout.
    }

    /**
     * Dynamically changes to FlowLayout.
     */
    protected void jb2Clicked(ActionEvent e) {
        System.out.println("Set FlowLayout...");
        getContentPane().setLayout(new FlowLayout());
        validate();  // forces this frame to re-layout.
    }
}
```

Frame2App.java

```
package app;

import javax.swing.*;
import view.*;

public class Frame2App {

    public static void main(String[] args) {
        JFrame f = new Frame2("A JFrame with 2 JButtons with event listeners");
        f.setVisible(true);
        f.setSize(300, 100);  // Guess what this does!
        f.setLocation(200, 200);  // Guess what this does!
```

```
        // Forces the frame to re-layout its components:
        f.validate();
    }
}
```

Frame3.java

```
package view;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/**
 * Same functionality as Frame2 but uses an adapter interface to communicate
 * with the outside world instead.  This design allows varying the communication
 * with the outside world.  How the outside world reacts to the events that
 * happen to the view is a variant behavior!
 * Uses the Null-Object Pattern to initialize the adapter avoiding checking for
 * null.
 * @author DXN
 */
public class Frame3 extends Frame2 {

    // Initializes the adapter to a null object:
    private IView2World _v2wAdapter = new IView2World() {
        public Object button1Clicked (ActionEvent e) {
            return null; // does nothing!
        }
        public Object button2Clicked (ActionEvent e) {
            return null; // does nothing!
        }
    };

    public Frame3(String title) {
        super(title);
    }

    public void setV2WAdapter(IView2World v2w) {
        if (null == v2w) {
            throw new IllegalArgumentException("Argument cannot be null!");
        }
        _v2wAdapter = v2w;
    }

    /**
     * Tells _v2wAdapter the button click event happens on jb1.
     */
    protected void jb1Clicked(ActionEvent e) {
        _v2wAdapter.button1Clicked(e);
```

```
    }

    /**
     * Tells _v2wAdapter the button click event happens on jb2.
     */
    protected void jb2Clicked(ActionEvent e) {
        _v2wAdapter.button2Clicked(e);
    }

}
```

IView2World.java

```
package view;

import java.awt.event.*;

/**
 * Adapter connecting Frame3 (the View) to the outside world.
 */
public interface IView2World {
    /**
     * Called by Frame3 when its button 1 is clicked.
     */
    public Object button1Clicked (ActionEvent e);

    /**
     * Called by Frame3 when its button 2 is clicked.
     */
    public Object button2Clicked (ActionEvent e);
}
```

Frame3Controller.java

```
package controller;

import view.*;
import java.awt.event.*;
import javax.swing.JFrame;
import java.awt.*; // For layout managers.

/**
 * The controller that instantiates a concrete IView2World object and connects
 * the world outside of Frame3 (the view) to Frame3.
 * The concrete adapter is implemented as an anonymous inner classe.
 */
public class Frame3Controller {

    public JFrame constructFrame()  {
```

```
        // Local variable needs to be final so that the local inner class
        // can access it:
        final Frame3 frame = new Frame3("View and Controller");

        /**
         * Install as concrete anonymous IView2World adapter in frame, without
         * frame knowing what the adapter does.
         */
        frame.setV2WAdapter(new IView2World()  {

            public Object button1Clicked (ActionEvent e) {
                frame.getContentPane().setLayout(new GridLayout(2,1));
                frame.validate();
                return null;
            }

            public Object button2Clicked (ActionEvent e) {
                frame.getContentPane().setLayout(new FlowLayout());
                frame.validate();
                return null;
            }
        });
        frame.setVisible(true);
        return frame;
    }
}
```

Frame3App.java

```
package app;

import view.*;
import controller.*;

/**
 * Instantiates the controller and builds the frame!
 */
public class Frame3App {

    public static void main(String[] args) {
        new Frame3Controller().constructFrame().validate();
    }
}
```

Chapter 7

Labs

7.1 DrJava[1]

DrJava is a lightweight pedagogical environment for Java development created by the Programming Languages Team (PLT)[2] at Rice University. DrJava provides a way to edit and save java code with key words highlighting, curly brace matching, and an interactive environment to manipulate objects and test code without having to write the main method. It can be freely downloaded from the web. Please see the DrJava home page[3]

Dr Java Tutorials

On the main DrJava web site[4] , one can find documentation and tutorials on its use:

- **Quick Start Guide**[5] : This guide covers the most important aspects of using DrJava, such as opening and saving files, compiling, testing, debugging and setting preferences.
- **Full Documentation**[6] : This is the full documentation for DrJava, covering all aspects of the system.
- **FAQ**[7] : This Frequently Asked Questions list covers many of the common issues that come up when running DrJava.
- **Video Tutorials**[8] : Some short introductory Flash vidoes on installing Java and DrJava and running DrJava. This is a good place to start for beginnings to get a feel for what can be done with DrJava. Note: The videos may not run in Internet Explorer, if not, please try accessing the videos using another browser, such as Firefox.

Below are some short descriptions of the various parts of DrJava.

7.1.1 Editing

Definitions Pane: When you run DrJava you will see a window appear. This window (GUI) consists of four subwindows. The top half of the GUI constitutes the Definitions pane. You type in all the class definitions here. After you type in some code, you need to click on the save button before you can compile your code. All the classes in the Definitions pane will be saved in a single file. There should only be one public class in the Definitions window, and the saved file should have the same name as that of the public class with the extension *.java*.

[1] This content is available online at <http://legacy.cnx.org/content/m11659/1.6/>.
[2] http://www.cs.rice.edu/CS/PLT/
[3] http://www.drjava.org
[4] http://www.drjava.org
[5] http://www.drjava.org/docs/quickstart/
[6] http://www.drjava.org/docs/user/
[7] http://www.drjava.org/faq.shtml
[8] http://www.drjava.org/videos/videos.shtml

7.1.2 Compiling

Compiler Output Pane: You compile your Java code by clicking on the *Compile All* button in the menu bar at the top. Every time you compile your code, DrJava will display all compile error messages here. Clicking on an error message will highlight the line where the error is suspected to take place in the Definitions pane. If there is no compile error, DrJava will declare success in this pane.

7.1.3 Running

Interactions pane: There are several ways to run your Java code. For now, we will restrict ourselves to the Interaction pane at the bottom of the main GUI window. This is where you can type in any valid Java statement. Usually, you would type in code to instantiate objects of classes defined in the Definitions window, and call their methods to check whether or not they perform correctly. Typing a valid Java expression terminated with a semi-colon and then pressing the *Return* (*Enter*) key, will cause DrJava to evaluate the expression but NOT printing the result. If you want DrJava to print the value of the result in the Interactions window, you should press *Return* without terminating the expression with a semi-colon. There is a menu item to reset (i.e. clear) the Interactions window. Another way to clear the Interactions window is to force a re-compile by editing the Definitions pane. If your code has printing statements, the output will be displayed in the Console Output pane.

7.1.4 Testing

There are many ways to test your code. The most formal way is to use JUnit testing facilities, which is covered in a separate module: Unit Testing with JUnit in DrJava (Section 7.2). For simple tests, you can test your code by directly interacting with it in the Interactions pane.

7.1.5 Debugging

To debug a program in DrJava, first put DrJava into **Debugging Mode** by setting the check box under the **Debugger** on the main menu. This will enable all the other debugging features. When debugging mode is active, the debugging pane will appear near the bottom of the DrJava window. This pane contains the **Watch** window and tabs that show the current **Stack** and **Threads** status. The debugging pane also has buttons for **Resume** debugging, **Step Into**, **Step Over** and **Step Out**. These features are described in detail below.

The basic technique for debugging is to set **breakpoints** on lines of code that are of interest/problematic and then to either step slowly through the code execution from that point and/or to examine the values of various variables.

Breakpoints: Under debugging mode, DrJava will stop the execution of a program whenever a breakpoint is encountered. Execution stops **before** the breakpoints line of code is run. To set or clear a breakpoint, place the cursor on the desired line of code and either use the **Debugger/Toggle Breakpoint on Current Line** or by simply pressing **Ctrl-B** or by right-clicking the desired line and selecting **Toggle Breakpoint**. Note that it only makes sense to put a breakpoint on a line of code that actually executes, which **excludes**

- Blank lines
- Comments
- Method signature lines
- Lines with a single curly brace on them.
- Field/attribute declarations that have no initialization code.

To **clear a breakpoint**, follow the same procedure used to set the breakpoint.

Whenever the program execution is stopped on a line of code, the **Interactions pane** can be used to examine the value of any variable that is in scope or even to execute methods on those variables (if they are objects). You can even set the values of any available variables by assigning a new value to them. This is

useful if they have the wrong value for some reason and you want to check if the rest of your code will run properly if the correct value is present.

Clear All Breakpoints: Clicking on **Debugger/Clear All Breakpoints** will clear all breakpoints that have been set in your program.

Clicking on **Debugger/Breakpoints** or pressing **Ctrl+Shift+B** will bring up a tabbed pane at the bottom of the DrJava window that enables you to conveniently examine, navigate to, delete, or disble (without deleting) all of the breakpoints in your program.

Step Into: When execution has stopped on a line of code, clicking on the **Step Into** button in the debugging pane or selecting the menu item **Debugger/Step Into** or pressing **F12** will cause the current line of code (highlighted in blue) to be executed. If the current line of code involves the invocation of a method, then the current line of code will be advanced **into** the first method to be called, as per Java's order of execution rules. If the current line of code is the last line in a method, the current line of code will become the next executable line of code in the method that **called** the current method. Note that if the caller's line of code involved multiple method calls, then the current line of code will return to that line of code in the caller and then execution will advance into the next method to be called, or if the current method was the last method to be called, the current line of code will be the next line of code in the caller.

Step Over: When execution has stopped on a line of code, clicking on the **Step Over** button in the debugging pane or selecting the menu item **Debugger/Step Over** or pressing **F11** will cause the current line of code to be executed. This is very similar to Step Into, but if the the execution will **not stop** inside of any methods called in the current line of code. Thus the new current line of code will always be in the same method, unless the executed line was the last line in the method, upon which execution will stop in the calling method as usual. This feature is very useful when you are confident that the methods being called work properly and you are not interested in their detailed execution.

Step Out: When execution has stopped on a line of code, clicking on the **Step Out** button in the debugging pane or selecting the menu item **Debugger/Step Out** or pressing **Shift+F12** will cause the execution to resume until the current method exits. Excution will stop at the next executable line in the **calling method**. This is very useful to quickly exit a method whose detailed execution no longer interests you (for instance, if you accidentally stepped into a method that you wanted to step over).

Resume Debugging: When execution has stopped on a line of code, clicking on the **Resume** button in the debugging pane or selecting the menu item **Debugger/Resume Debugging** or pressing **F7** will cause the program execution to resume and continue until the next breakpoint is encountered. This is useful when you are no longer interested in the detailed execution of the code until the next breakpoint.

Watches: Watches are used when you want to continuously monitor the value of a particular variable. Simply type the name of the variable or field into the Watch pane and when code execution has paused, if that variable name is in scope, its value will be displayed.

Stack: The stack contains an ordered list of all methods with pending operations at any given point of a program's execution. That is, when the program execution has stopped at a particular line of code, the stack contains the method that line of code is in, the method that called that method, the method that called that method, etc. In DrJava, the **Stack** tab will show the stack with the current method at the top and the calling method below it. Looking at the stack is very useful for determining where in your program you actually are, as often a method is called from locations that are unintended or one discovers that a method is not being called from where you wish it to be called.

Threads: A Java program run on many simultaneous execution paths or "threads", each performing different tasks necessary for the total operation of a Java application. A simple program written by a programmer may have a single thread that runs the code written by that programmer, but may also have many other threads performing such tasks as managing the graphical user interface ("**GUI**") or managing the memory allocated to the program. It is also possible for a programmer to write code that utilizes multiple threads of execution to accomplish the desired goals. The **Threads** tab is most useful for debugging these multi-threaded programs by enabling the developer to better understand which threads are accessing which objects.

7.2 Unit Testing with JUnit in DrJava[9]

Object oriented systems derive great complexity through the interactions between the objects in the system. It is often impossible to test the entire range of the total complex behavior that a system is designed to exhibit. What one can say however is that if any given part of the system does not perform as desired, then one can make no assurances whatsoever about the integrity of the system as a whole. Thus the testing of these smaller parts, or "units" is a crucial element of developing any large-scale OO system. A **unit test** is a **complete** test of a small, atomic sub-system. Note that this is **not** the same from a partial test of some aspect of the whole system!

Unit tests are often performed at the individual class level, meaning that a test class that runs all the complete battery of tests on the tested class, is written for each class to be tested. In some cases however, certain systems of classes are not divisible into isolated, atomic units and in such must be tested as a whole. The key is to test as small a piece of the system as possible and to test it as thoroughly as possible.

7.2.1 Using JUnit in DrJava

Suppose we have a class such as the following that we wish to test. Note that class is `public` and that the method we wish to test is also `public`.

Sample Code To Be Tested

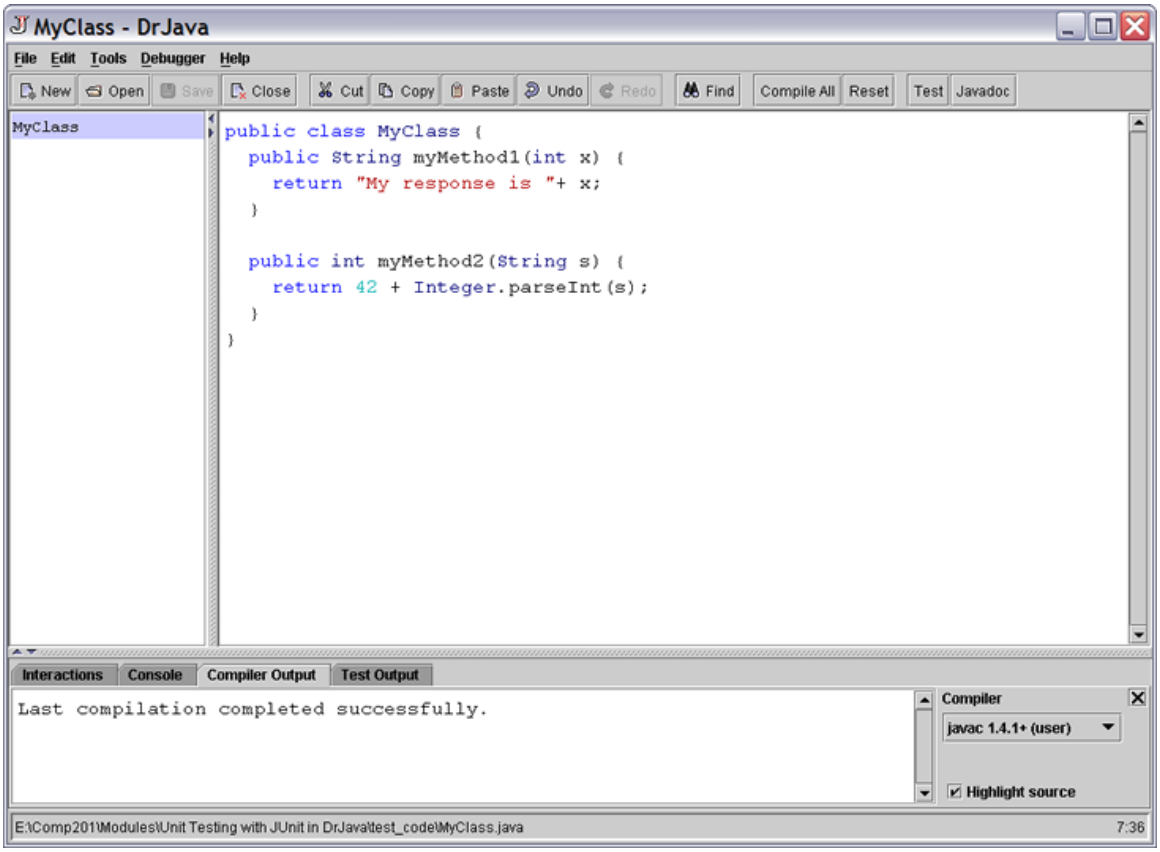


Figure 7.1: Some example code to be tested.

[9] This content is available online at <http://legacy.cnx.org/content/m11707/1.4/>.

In DrJava, select "File/New JUnit Test Case...". Enter a name that is descriptive of the test(s) you wish to make. A recommendation is to use the class name being tested prepended with "`Test_`", such as "`Test_MyClass`". This enables all your test classes to be easily identified, DrJava will then automatically create a valid JUnit test class, such as below. (Note that DrJava does not initially name the new class file, but the default name suggested by DrJava when you attempt to save it will be correct.)

Autogenerated Unit Test Class

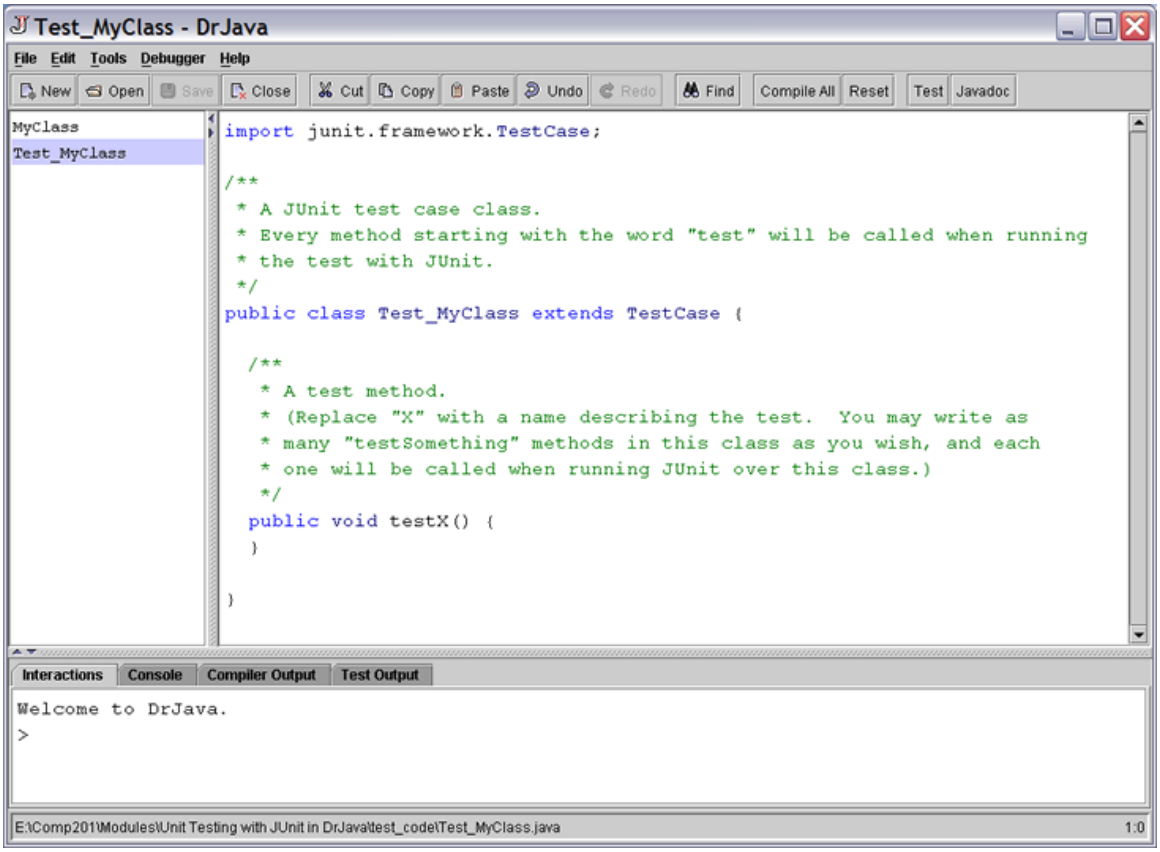


Figure 7.2: Test class autogenerated by DrJava.

Rename the auto-generated "`testX()`" method to something more descriptive of the particular test you'd like to perform. The new name **must** start with "`test`" and **must** return `void` and take no input parameters. You can create as many test methods as you wish to test your code. JUnit will automatically run all methods that begin with "`test`" as test methods. Typically, a single test method will test a single method on the class under test. There are situations however where a single method under test may require several test methods to properly test its full range of operation. In the code that follows, the `testX()` method has been renamed to "`test_myMethod1()`".

7.2.1.1 assertEquals(...)

There are a number of ways in which one can test that the proper result is produced. The simplest method is to compare a single output value to an expected value. For instance, suppose you are testing a method that returns a result. One can compare the actual returned value to the value one expects for the given

inputs. There are two methods supplied by the junit.framework.TestCase class, which is the superclass of the test class generated by DrJava. The first is `void assertEquals(String failResponse, Object actual, Object expected)`. The actual input parameter is the actual result of the method under test. The expected parameter is the expected result. failResponse is a String that JUnit/DrJava will display if the actual value does not "equal" the expected value. The `assertEquals(...)` method is overridden such that actual and expected values can be either Objects or primitives. `assertEquals(...)` uses the `equals(...)` method of `Object` to makes it determination. For primitives, the usual `==` is used. For instance, we could write the test case below. If the actual value does not equal the expected value, `assertEquals` throws a exception that is caught by the JUnit framework and an error message will result. To test the code, first compile all the code then select the test class. Right-click the test class and select "Test Current Document "or click on "Tools/Test Current Document." The following result will be displayed, given the test code we wrote to purposely fail:

Using assertEquals(...) in a test case

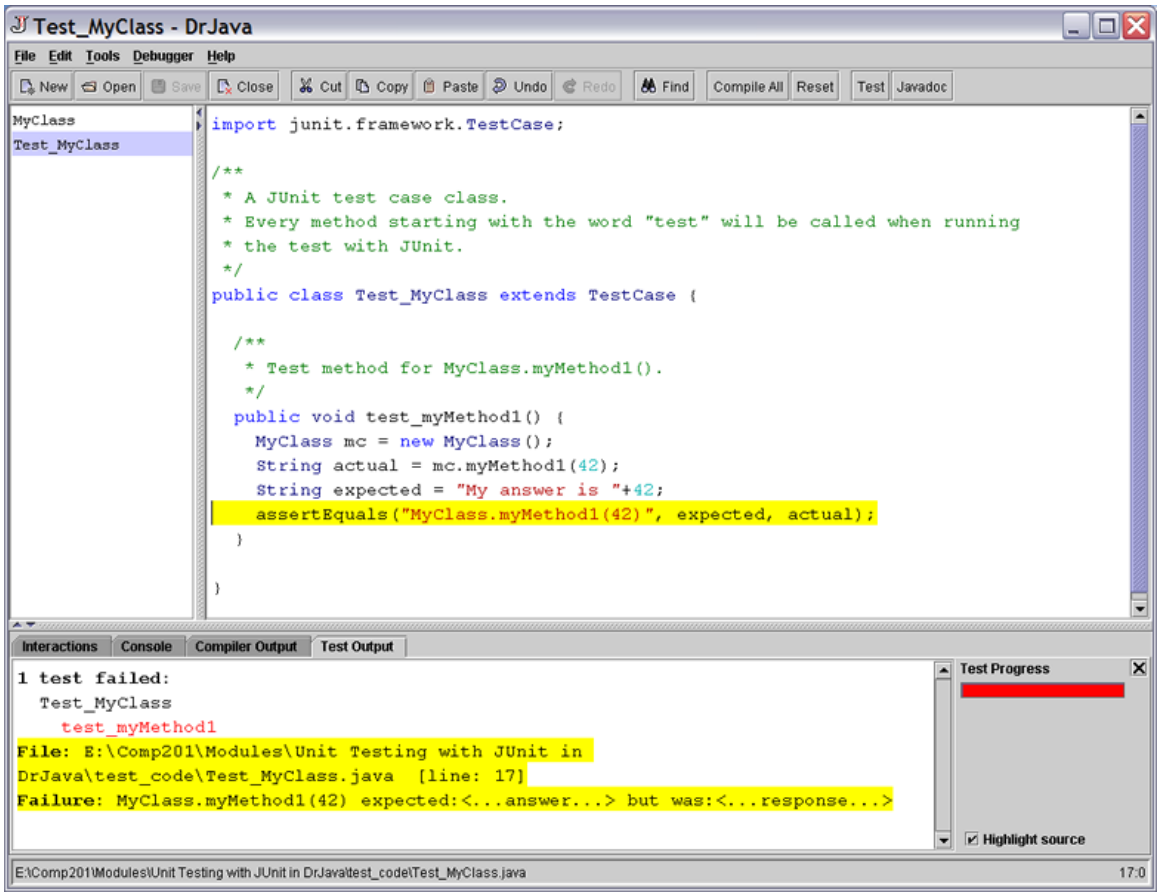


Figure 7.3: `assertEquals(...)` displays the error string as well as comparative information if the `actual` and `expected` values are not equal.

As shown on the screen shot, DrJava clearly indicates in red the test method that failed and the difference between the expected and actual values. It also highlights the line of code in the test method where the failure occured.

There is one exception to the syntax of `assertEquals`, and that is when the values being compared are `doubles`. This is because round-off errors in calculating floating point values means that it is almost always impossible to generate the same value from two different calculations, even if mathematically they are the same. For instance, compate `2.3*2.3` and `5.29`. Mathematically identical, but on a PC running Windows, Java calculates them to be different by approximately `0.00000000000000089` (or `8.9e-16` in Java syntax). Thus, if the expected and actual values are to be of type `double`, then a 4'th input parameter is required. This last parameter is a tolerance value, a plus-or-minus amount within which one considers the expected and actual values to be equal. For instance:

```
assertEquals("Testing doubles: 5.29 vs. 2.3*2.3", 5.29, 2.3*2.3, 9e-16);
```

should pass, but

```
assertEquals("Testing doubles: 5.29 vs. 2.3*2.3", 5.29, 2.3*2.3, 8e-16);
```

should fail. Note that the tolerance value should always be a **positive** value.

7.2.1.2 assertTrue(...)

Another method provided by `TestCase` is `void assertTrue(String failResponse, boolean result)`. This is a simplified version of `assertEquals(...)` used when the result can be expressed as a `boolean` true or false. Note that any test using `assertTrue` can also be written as `assertEquals(failResponse, result, true)`. For instance we could write another test method to test the `myMethod2()` method of `MyClass`. The test class now has two test methods and JUnit will run both ot them when we click on "Test Current Document." Here, the second test method passes as shown in green below. The first method still fails and its error messages are still shown. Clicking on the error message will highlight the line where the error occured. Correcting the code in `myMethod1(...)` would result in all the test methods being listed in green with no error messages.

Using assertTrue(...) in a test method

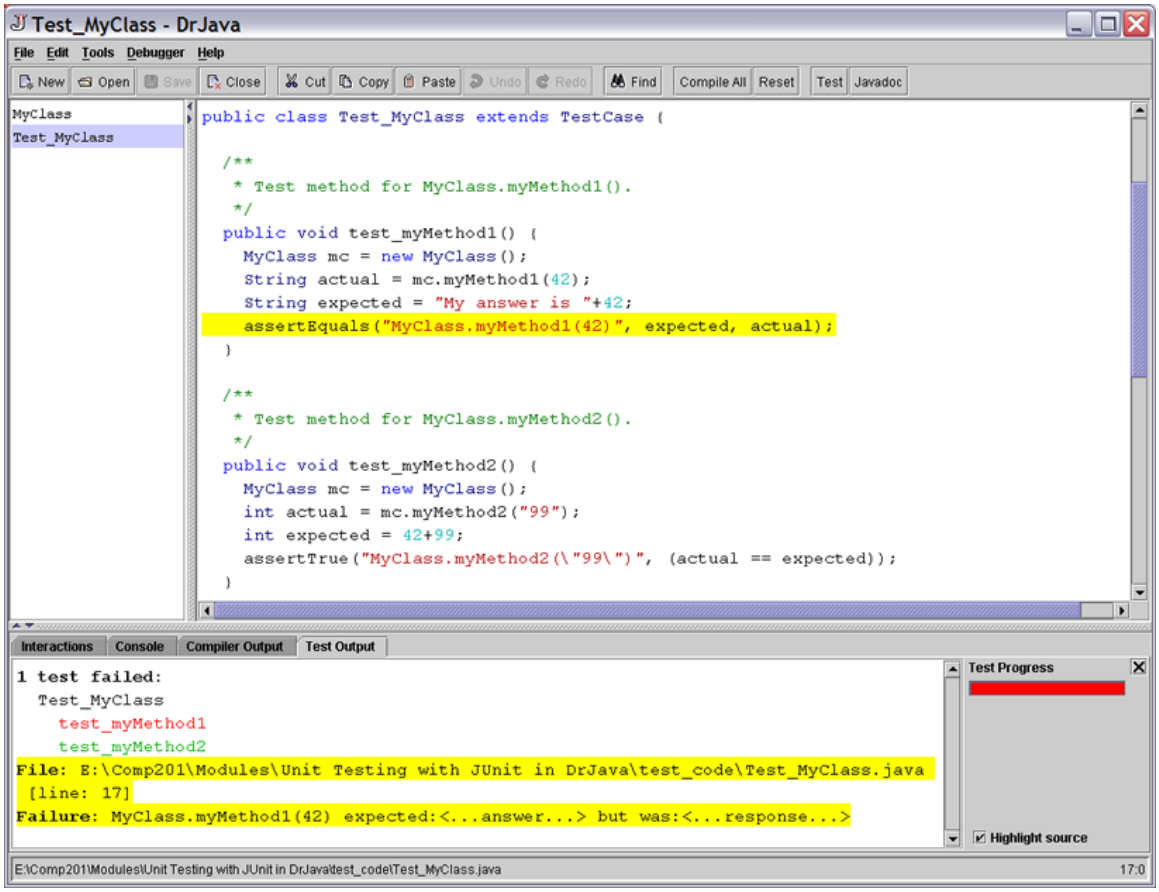


Figure 7.4: `assertTrue(...)` is used in `test_myMethod2(...)` above and does not generate an error because it is executed with a boolean `true` value.

7.2.1.3 fail(...)

For more complex testing, `TestCase` provides the `void fail(String failResponse)` method. Calling this method will immediately cause the test method to fail and the `failResponse` string will be displayed. Since this method does not know the actual or expected results, the `failResponse` string should contain more detailed information about what exactly failed and how it failed. In the next screen shot is an example of using `fail(...)` where the test code was written such that it would fail:

Using fail(...) in a test method

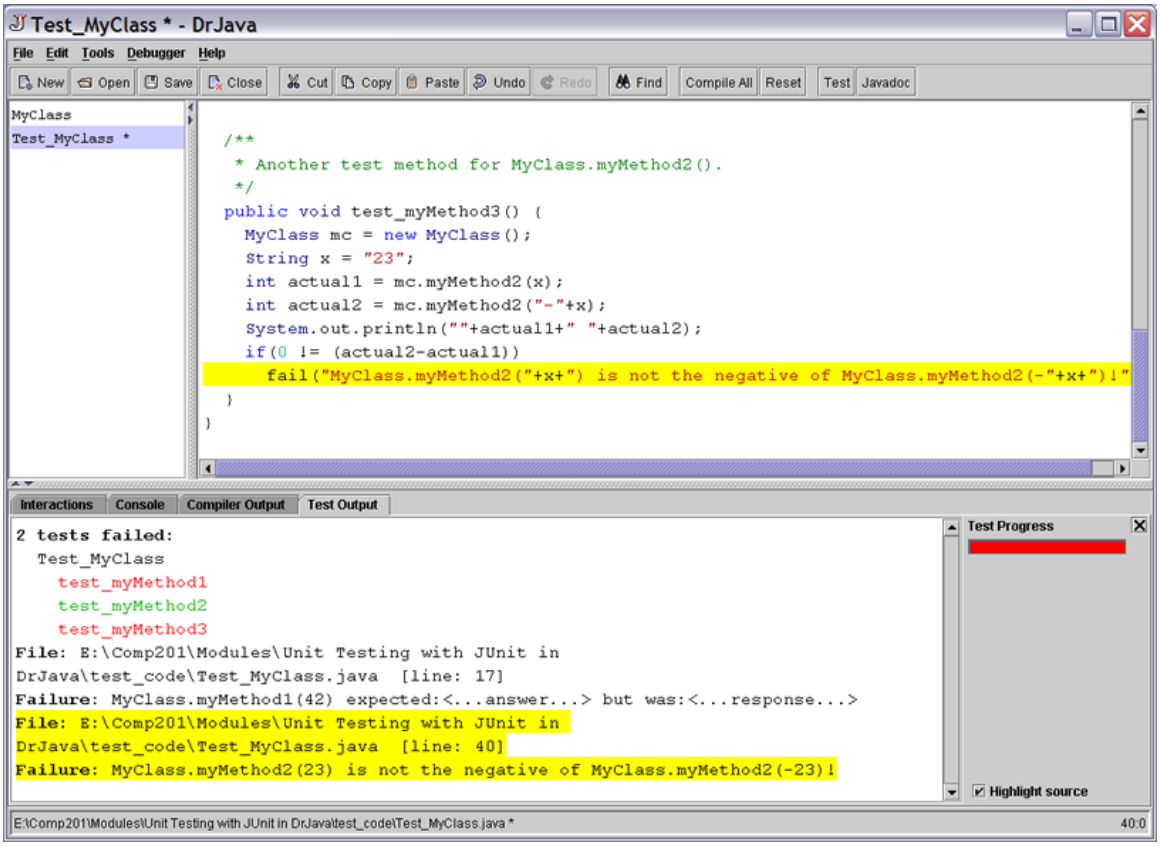


Figure 7.5: The `fail(...)` method immediately throws an error exception when it is executed.

7.2.1.4 Additional Capabilities

Note that any test method can call other methods if needed . This is useful when the testing procedure is complex and needs to be broken down into smaller pieces. Do not name these helper methods starting with "test" or they will be run separately as test cases! Using helper methods is also useful when certain types of tests are performed only under certain conditions. `assertEquals(...)`, `assertTrue(...)` and `fail(...)` can be called from anywhere, but the DrJava will only highlight the line in the main test method that generated the error, i.e. the line that called the helper method. It will not highlight the line in the helper method where the actual call to `assertEquals(...)`, `assertTrue(...)` or `fail(...)`was made.

Sometimes in complex testing scenarios, there is a significant amount of initialization work required to set up the system to be tested. This is often the case when testing a system of interdependent objects where the entire system must be set up before any single object can be tested. If the initialization code is the same for all the tests being conducted, the method `protected void setup()` can be declared and be used to execute the necessary initializations. To insure "clean" test runs, JUnit/DrJava re-instantiates the test class before running each test method. The `setup()` method, if it exists, is called before any test method is executed. This means that the test class **cannot** utilize any internal field values that one test method modifies and another test method expects to use as modified.

Likewise, in the event that after a test is run that significant amounts of common "clean-up" code is

required, one can declare the method protected void `tearDown()`. This method runs after each test method and is useful for insuring, for instance, that files and network connections are properly closed and thus keep them from interfering with other tests.

The help system in DrJava has more detailed information on testing with JUnit, including how to create a test suite of multiple tests.

Chapter 8

Resources

8.1 Java Syntax Primer[1]

8.1.1 General:

A Java program consists of one or more classes one of them must be public and must have a method with the following signature:

```
public static void main (String[] args)
```

Basically, the `main()` method will instantiate appropriate objects and send them "messages" (by calling their methods) to perform the desired tasks. The `main()` method should not contain any complicated program logic nor program flow control. In the example shown below, `PizzaClient` is the main class with the `main()` method.

[1] This content is available online at <http://legacy.cnx.org/content/m11791/1.2/>.

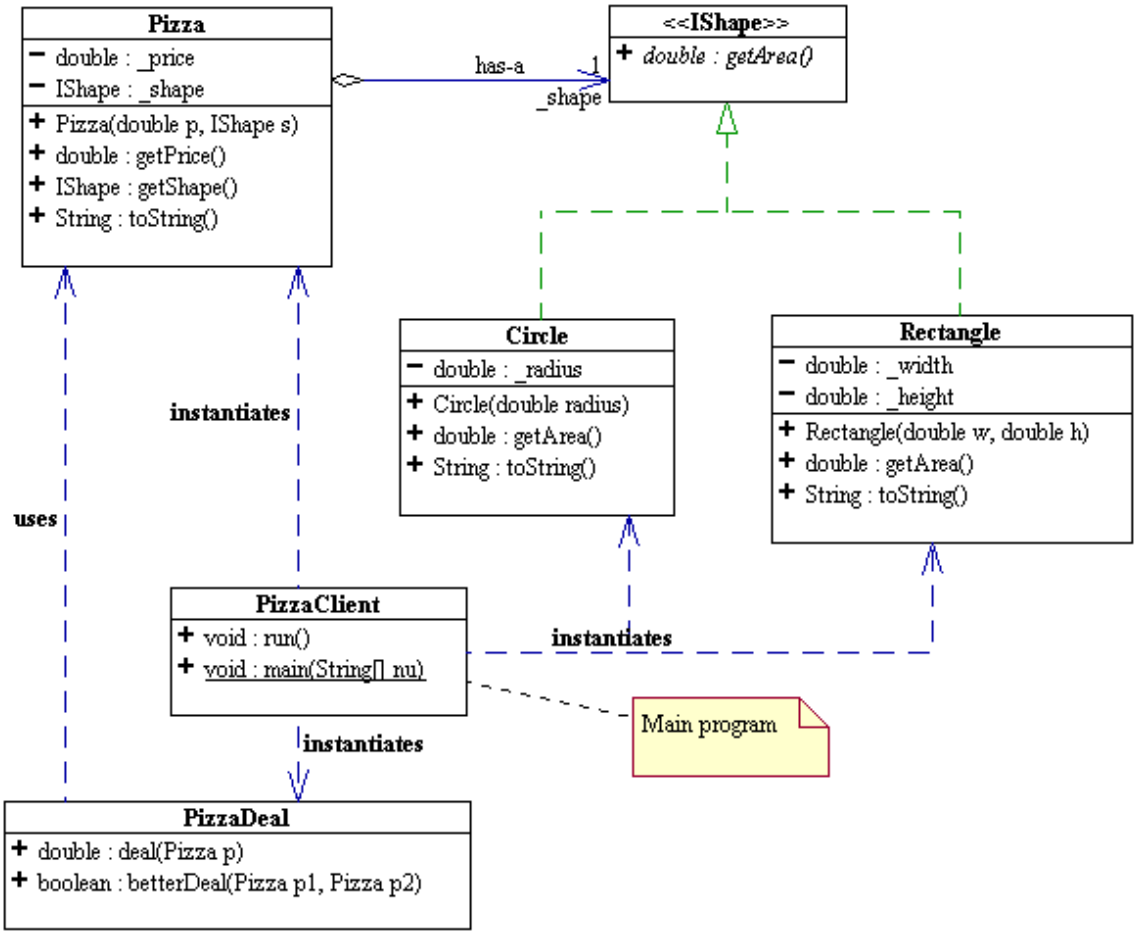


Figure 8.1: UML Class Diagram of a Java program

Example:

```
public class PizzaClient {
/**
 * Prints the answer to the problem stated in the above..
 */
public void run() {

    Pizza round = new Pizza (3.99, new Circle (2.5));
    Pizza rect =  new Pizza (4.99, new Rectangle (6, 4));

    PizzaDeal pd = new PizzaDeal();
```

```
        System.out.println(round + " is a better deal than " + rect + ": " +
                           pd.betterDeal(round, rect));

    }

    /**
    * Main entry to the program to find the better deal.
    * Instantiates an instance of PizzaClient and tells it to run.
    * This is what all main() should do: instantiates a bunch of objects and
    * "turn them loose"!
    * There should be no complicated logic and/or control in main().
    * @param nu not used
    */
    public static void  main (String[] nu) {
        new PizzaClient().run();
    }
}

public class Rectangle implements IShape {

    private double _width;
    private double _height;

    /**
    * Initializes this Rectangle width a given width and a given height
    * @param w width of this Rectangle, >= 0.
    * @param h height of this Rectangle, >= 0.
    */
    public Rectangle(double w, double h) {
        _height  = h;
        _width   = w;
    }

    /**
    *  @returns this Rectangle's area.
    */
    public double getArea() {
        return _height * _width;
    }

    /**
    * Overrides the inherited method from class Object.
    * @returns a String describing a Rectangle with its width and height.
    */
    public String toString() {
        return "Rectangle(h = " + _height + ", w = " + _width + ")";
    }
}
```

Notes on the `toString()` method:

`toString()` is a method that is inherited all the way from the base class, Object. It is the method

that the Java system calls by default whenever a string representation of the class is needed. For instance, `"This is "+ myObject` is equivalent to `"This is " + myObject.toString()`. DrJava will call an object's `toString()` method if you type the object's name in the interaction window, *without* terminating the line with a semicolon. The return value of `toString()` is what prints out on the next line.

8.1.2 Comments syntax:

```
// Line-oriented - comment goes to end of the current line.

    /*
    block-oriented
    can span several lines.
*/
```

8.1.3 Class definition syntax:

[...] means optional.

```
    [public] class class-name [inheritance-specification] {
    [field-list;]
    [constructor-list;]
    [method-list;]


}
```

Inheritance Specification looks like
`[extends SomeClass] [implements Interface1,..., InterfaceN]`
We will discuss the meaning of inheritance in due time.

8.1.4 Java Statement syntax:

NOTE: Each Java statement must terminate with a semi-colon.

8.1.5 Field list syntax:

A field list consists of zero or more field declarations of the form
`[static] [final] [public | private | protected] field-type field-name [assignment];`

8.1.6 Constructor list syntax:

A constructor list consists of zero or more constructor definitions of the form

```
    [public | private | protected] class-name ([parameter-list]){
    [statement-list;]
}
```

NOTE: The constructor's name must the same as the class name. Constructors are used for initialization of the object during the object's instantiation only.

8.1.7 Method list syntax:

A method list consists of zero or more method definitions of the form

```
[static] [final] [public | private | protected]
return-type method-name ([param-list]) {
    [statement-list;]
}
```

A return type `void` means the method does **not** return any value.
`param-list` can be:

- empty;

for example,

```
public double getArea() {
// code ...
}
```

- of a fixed number of parameters

```
type1 param1, type2 param2, ...  , typeN paramN
```

for example,

```
private void doSomethingWith(int n, double x, Pizza p) {
// code ...
}
```

- of a variable number of parameters (*this is a new feature for Java 5.0 and upward*)

```
type1 param1, type2 param2, .., typeN... params
```

In the above, `typeN... params` is called a variable argument list. It must appear at the end of the whole parameter list.
for example,

```
private void doSomethingWith(int n, double x, Pizza... p) {
// code ...
}
```

The variable argument list can have zero or more arguments. Here is a simple example: use DrJava to define the following class in the definition pane.

```
public class VarArgExample {
/**
* returns the number of arguments in the variable argument list.
*/
int argNum(int... nums) {
    return nums.length; // the var arg list is viewed as an array
}
}
```

Compile the above class and in the interactions pane, type the following:

```
VarArgExample vae = new VarArgExample(); // creates a VarArgExample object
vae.argNum() // returns 0
vae.argNum(42) // returns 1
vae.argNum(34, -99) // returns 2
```

8.2 Design Patterns Resources[2]

8.2.1 Links

Connexions lens on Design Patterns[3]
This collection of modules covers most of the design patterns used in introductory and intermediate level object-oriented programming and design. This lens was created as a resource for the CNX "Principles of Object-Oriented Design" course[4]. The modules are separate from the course materials and contain reference and supplemental information.
Research papers by D. X. Nguyen and S. Wong[5]
These are the research papers upon which much of the CNX "Principles of Object-Oriented Design" course[6] is based. Some of the papers cover more advanced material however. All of the papers are in Java.

8.2.2 Books

Design Patterns, Elements Of Reusable Object-Oriented Software, by Gamma, Helm, Johnson, and Vlissides, Addison-Wesley, 1995. ISBN 0201633612
This is the bible of the pattern community. It predates Java. The authors are affectionately referred to as the Gang of Four (GoF). All examples are written in C++ and/or Smalltalk. It can be hard to read if you are not familiar with these two languages, though the ideas are language independent. The "pattern language" used in this book also requires a lot of getting used to. We have learned so much from this book, and we are still reading it and learning from it. There is a version on CD-ROM which is cheaper (and perhaps, more versatile). We strongly recommend it, especially to any student considering further work in object-oriented programming.
Head First Design Patterns, First Edition, by Eric Freeman and Elizabeth Freeman, O'Reilly, 2004. ISBN 0-596-00712-4
A very approachable, easy-reading book on design patterns for beginners that is written in Java. highly recommended for anyone just starting out with design patterns.

[2] This content is available online at <http://legacy.cnx.org/content/m32615/1.1/>.
[3] http://cnx.org/lenses/swong/design-patterns
[4] *Principles of Object-Oriented Programming* <http://legacy.cnx.org/content/col10213/latest/>
[5] http://www.bandgap.cs.rice.edu/personal/adrice_swong/public/WebPages/research/default.htm
[6] *Principles of Object-Oriented Programming* <http://legacy.cnx.org/content/col10213/latest/>

Glossary

A **array**

At its most basic form, a random access data structure where any element can be accessed by specifying a single index value corresponding to that element.

Example: `anArray[4]` gives us `itemE`. Likewise, the statement `anArray[7] = 42` should replace `itemH` with `42`.

assignment

To set a variable to a particular data value.

B **Binary Tree**

A (mutable) binary tree, `BiTree`, can be in an empty state or a non-empty state:

- When it is empty, it contains no data.
- When it is not empty, it contains a data object called the root element, and 2 distinct `BiTree` objects called the left subtree and the right subtree.

C **Class**

1. An abstract definition of a set of related objects. A class definition specifies all the invariant behaviors and other attributes common to all the objects in the set.
2. In Java, the keyword `class` denotes the beginning of a class definition.

D **Definitions pane**

The pane at the upper right of the DrJava window where one edits class definitions.

I **Instantiate**

To create an object based on the specifications defined in a class definition. The object created is called an **instance** of that class.

Interactions pane

The pane at the lower edge of the DrJava window where one can interactively execute Java statements.

invariant

The parts of a probram, such as values or programmatic behaviors, that do not vary from one invocation to the next. Note that while a value may be variant, an abstraction of that value may be invariant.

L **literal**

An explicit, concrete textual reprepresentation of a value of a given type. Literals are often used to set the values for variables.

T **type**

A set of values with certain common characteristics. In Java, all data values must be of some type.

Example: `int` is a type that is used to represent integer number values. `double` is a type that is used to represent real number values. `String` is a type that is used to represent a string of characters.

U **UML**

Unified Modeling Language, developed by the Object Management Group ("OMG")

Unit Test

The testing of a single class or small collection of classes (a "unit") to verify correct behavior at a fine-grained level.

V **variable**

A memory location to hold a particular value of a given type. In a strongly-typed language such as Java, all variables must have a type. This is not true in all languages however. In a Java program,

variables have names called indentifiers, which are sequences of characters put together according to the following rule. A must begin with an alphabet character (e.g. 'a', 'b', 'X', 'Y', etc.) and may be followed by zero or more alphabet characters and/or digit characters (e.g. '0', '1', etc.) and/or the underscore character ('_'). For examples, cp3PO is a valid variable name while Darth Vader is not because it has a blank character between the 'h' and the 'V'.

variant

The parts of a program, such as values or programmatic behaviors, that vary from one invocation to the next. Note that while a value may be variant, its abstraction may be invariant.

Index of Keywords and Terms

Keywords are listed by the section with that keyword (page numbers are in parentheses). Keywords do not necessarily appear in the text of the page. They are merely associated with that section. *Ex.* apples, § 1.1 (1) **Terms** are referenced by the page they appear on. *Ex.* apples, 1

Attributions

Collection: *Principles of Object-Oriented Programming*
Edited by: Stephen Wong, Dung Nguyen
URL: http://legacy.cnx.org/content/col10213/1.37/
License: http://creativecommons.org/licenses/by/3.0/

Module: "Abstraction"
By: Stephen Wong, Dung Nguyen
URL: http://legacy.cnx.org/content/m11785/1.21/
Pages: 1-5
Copyright: Stephen Wong, Dung Nguyen
License: http://creativecommons.org/licenses/by/1.0

Module: "Objects and Classes"
By: Stephen Wong, Dung Nguyen
URL: http://legacy.cnx.org/content/m11708/1.7/
Pages: 5-8
Copyright: Stephen Wong, Dung Nguyen
License: http://creativecommons.org/licenses/by/3.0/

Module: "Object Relationships"
By: Stephen Wong, Dung Nguyen
URL: http://legacy.cnx.org/content/m11709/1.5/
Pages: 9-12
Copyright: Stephen Wong, Dung Nguyen
License: http://creativecommons.org/licenses/by/1.0

Module: "UML Diagrams"
By: Stephen Wong, Dung Nguyen
URL: http://legacy.cnx.org/content/m11658/1.3/
Pages: 12-15
Copyright: Stephen Wong, Dung Nguyen
License: http://creativecommons.org/licenses/by/1.0

Module: "Union Design Pattern: Inheritance and Polymorphism"
By: Stephen Wong, Dung Nguyen
URL: http://legacy.cnx.org/content/m11796/1.11/
Pages: 15-21
Copyright: Stephen Wong, Dung Nguyen
License: http://creativecommons.org/licenses/by/1.0

Module: "Ballworld, inheritance-based"
By: Stephen Wong, Dung Nguyen
URL: http://legacy.cnx.org/content/m11806/1.8/
Pages: 21-26
Copyright: Stephen Wong, Dung Nguyen
License: http://creativecommons.org/licenses/by/1.0

Module: "Ballworld, composition-based"
By: Stephen Wong, Dung Nguyen
URL: http://legacy.cnx.org/content/m11816/1.6/
Pages: 27-34
Copyright: Stephen Wong, Dung Nguyen
License: http://creativecommons.org/licenses/by/1.0

Module: "List Structure and the Composite Design Pattern"
By: Stephen Wong, Dung Nguyen
URL: http://legacy.cnx.org/content/m15111/1.1/
Pages: 37-41
Copyright: Stephen Wong, Dung Nguyen
License: http://creativecommons.org/licenses/by/2.0/

Module: "List Structure and the Interpreter Design Pattern"
By: Dung Nguyen, Stephen Wong
URL: http://legacy.cnx.org/content/m15110/1.3/
Pages: 41-43
Copyright: Dung Nguyen, Stephen Wong
License: http://creativecommons.org/licenses/by/3.0/

Module: "Recursion"
By: Stephen Wong, Dung Nguyen
URL: http://legacy.cnx.org/content/m17306/1.4/
Pages: 43-47
Copyright: Stephen Wong, Dung Nguyen
License: http://creativecommons.org/licenses/by/3.0/

Module: "Visitor Design Pattern"
By: Dung Nguyen, Stephen Wong
URL: http://legacy.cnx.org/content/m16707/1.3/
Pages: 47-62
Copyright: Dung Nguyen, Stephen Wong
License: http://creativecommons.org/licenses/by/3.0/

Module: "Abstract Factory Design Pattern"
By: Dung Nguyen, Stephen Wong
URL: http://legacy.cnx.org/content/m16796/1.3/
Pages: 62-72
Copyright: Dung Nguyen, Stephen Wong
License: http://creativecommons.org/licenses/by/3.0/

Module: "Inner Classes"
By: Stephen Wong, Dung Nguyen
URL: http://legacy.cnx.org/content/m17220/1.4/
Pages: 72-93
Copyright: Stephen Wong, Dung Nguyen
License: http://creativecommons.org/licenses/by/3.0/

Module: "State Design Pattern"
By: Dung Nguyen, Stephen Wong
URL: http://legacy.cnx.org/content/m17225/1.5/
Pages: 95-97
Copyright: Dung Nguyen, Stephen Wong
License: http://creativecommons.org/licenses/by/2.0/

Module: "Mutable Linear Recursive Structure"
By: Dung Nguyen, Stephen Wong
URL: http://legacy.cnx.org/content/m17265/1.5/
Pages: 97-103
Copyright: Dung Nguyen, Stephen Wong
License: http://creativecommons.org/licenses/by/4.0/

Module: "Binary Tree Structure"
By: Stephen Wong, Dung Nguyen
URL: http://legacy.cnx.org/content/m17289/1.3/
Pages: 103-107
Copyright: Stephen Wong, Dung Nguyen
License: http://creativecommons.org/licenses/by/4.0/

Module: "Binary Search Tree"
By: Dung Nguyen
URL: http://legacy.cnx.org/content/m32616/1.1/
Pages: 107-114
Copyright: Dung Nguyen
License: http://creativecommons.org/licenses/by/3.0/

Module: "Arrays and Array Processing"
By: Stephen Wong, Dung Nguyen
URL: http://legacy.cnx.org/content/m17258/1.3/
Pages: 115-121
Copyright: Stephen Wong, Dung Nguyen
License: http://creativecommons.org/licenses/by/3.0/

Module: "Design Patterns for Sorting"
By: Stephen Wong, Dung Nguyen, Alex Tribble
URL: http://legacy.cnx.org/content/m17309/1.5/
Pages: 122-131
Copyright: Stephen Wong, Dung Nguyen, Alex Tribble
License: http://creativecommons.org/licenses/by/3.0/

Module: "Restricted Access Containers"
By: Stephen Wong
URL: http://legacy.cnx.org/content/m17101/1.1/
Pages: 133-140
Copyright: Stephen Wong
License: http://creativecommons.org/licenses/by/2.0/

Module: "Ordering Object and Priority Queue"
By: Stephen Wong, Dung Nguyen
URL: http://legacy.cnx.org/content/m17064/1.4/
Pages: 140-145
Copyright: Stephen Wong, Dung Nguyen
License: http://creativecommons.org/licenses/by/3.0/

Module: "Graphical User Interfaces in Java"
By: Stephen Wong, Dung Nguyen
URL: http://legacy.cnx.org/content/m17185/1.5/
Pages: 147-154
Copyright: Stephen Wong, Dung Nguyen
License: http://creativecommons.org/licenses/by/3.0/

Module: "More Java GUI Programming"
By: Stephen Wong, Dung Nguyen
URL: http://legacy.cnx.org/content/m17186/1.3/
Pages: 154-160
Copyright: Stephen Wong, Dung Nguyen
License: http://creativecommons.org/licenses/by/3.0/

Module: "DrJava"
By: Stephen Wong, Dung Nguyen
URL: http://legacy.cnx.org/content/m11659/1.6/
Pages: 161-163
Copyright: Stephen Wong, Dung Nguyen
License: http://creativecommons.org/licenses/by/1.0

Module: "Unit Testing with JUnit in DrJava"
By: Stephen Wong, Dung Nguyen
URL: http://legacy.cnx.org/content/m11707/1.4/
Pages: 164-170
Copyright: Stephen Wong, Dung Nguyen
License: http://creativecommons.org/licenses/by/1.0

Module: "Java Syntax Primer"
By: Stephen Wong, Dung Nguyen
URL: http://legacy.cnx.org/content/m11791/1.2/
Pages: 171-176
Copyright: Stephen Wong, Dung Nguyen
License: http://creativecommons.org/licenses/by/1.0

Module: "Design Patterns Resources"
By: Stephen Wong
URL: http://legacy.cnx.org/content/m32615/1.1/
Page: 176
Copyright: Stephen Wong
License: http://creativecommons.org/licenses/by/3.0/

Principles of Object-Oriented Programming
An objects-first with design patterns introductory course

About OpenStax-CNX

Rhaptos is a web-based collaborative publishing system for educational material.