Design Patterns

Collection Editor:
ukmie chano

Design Patterns

Collection Editor:

ukmie chano

Authors:

Antonio Garcia Castañeda
Dung Nguyen
Alex Tribble
Stephen Wong

Online:

< http://cnx.org/content/col10678/1.2/ >

C O N N E X I O N S

Rice University, Houston, Texas

Table of Contents

Chapter 1

uml

Chapter 2

Command Design Pattern[1]

2.1 Command Design Pattern

When two objects communicate, often one object is sending a command to the other object to perform a particular function. The most common way to accomplish this is for the first object (the **issuer**) to hold a reference to the second (the **recipient**). The issuer executes a specific method on the recipient to send the command.

But what if the issuer is not aware of, or does not care who the recipient is? That is, the issuer simply wants to abstractly issue the command?

The **Command design pattern** encapsulates the concept of the command into an object. The issuer holds a reference to the command object rather than to the recipient. The issuer sends the command to the command object by executing a specific method on it. The command object is then responsible for dispatching the command to a specific recipient to get the job done.

Note the similarities between the Command design pattern and the Strategy design pattern[2].

[1]This content is available online at <http://cnx.org/content/m17187/1.2/>.
[2]"Strategy Design Pattern" <http://cnx.org/content/m17037/latest/>

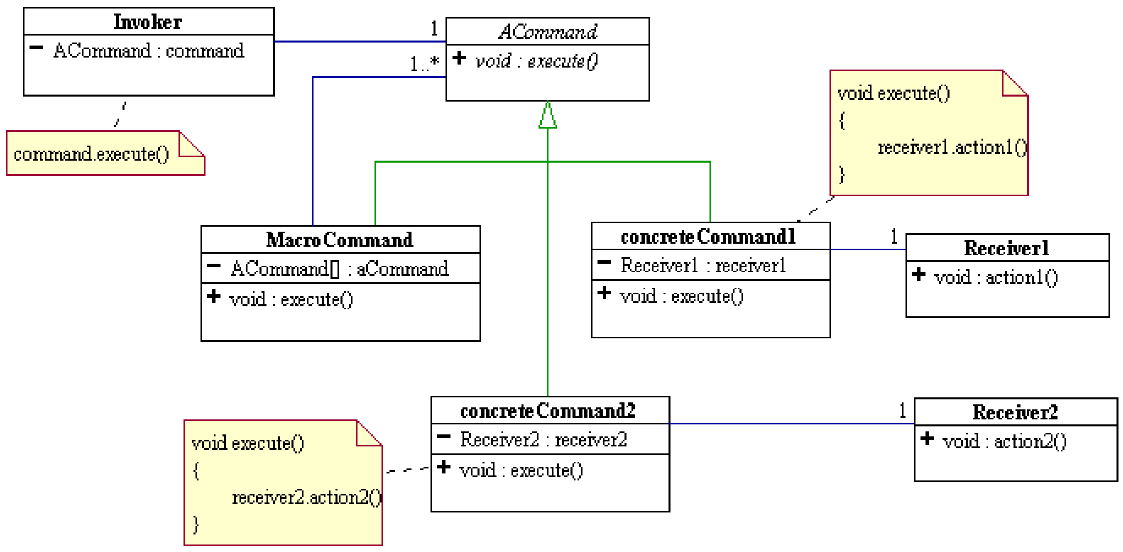


Figure 2.1

In the above diagram, the invoker holds an abstract command and issues a command by calling the abstract `execute()` method. This command is translated into a specific action on a specific receiver by the various concrete command objects. It is also possible for a command to be a collection of commands, called a **macro** command. Calling the `execute` method of a macro command will invoke a collection of commands.

Chapter 3

Abstract Factory Design Pattern[1]

3.1 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.

[1] This content is available online at <http://cnx.org/content/m16796/1.1/>.

```
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 3.1

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[2] to hide the concrete implementation from the user.

[2]http://www.exciton.cs.rice.edu/JavaResources/DesignPatterns/FactoryPattern.htm

3.2 2. Abstract List Behaviors

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

```
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); }
```

```
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 fi
 * isomorphic subcomponent called rest.  Its structural be
 * provides access to its internal data (first) and substr
 */
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 3.2

3.3 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 3.3

`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 3.4

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.

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 "()";
        }
    }

    /**
     * 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);
        }
```

Table 3.5

- 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

```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 3.6

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[3] described in the above.

Click here to download the complete source code and documentation of the list component[4] described in the above.

3.4 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,

[3]http://cnx.org/content/m16796/latest/file:///C:%5CUsers%5Cdxnguyen.ADRICE%5CDocuments%5CMy%20Web%20Sites%5CRice%5Ccom
spring%5Clectures%5Clec20%5ClistFW%5Cdoc%5C

[4]http://cnx.org/content/m16796/latest/file:///C:%5CUsers%5Cdxnguyen.ADRICE%5CDocuments%5CMy%20Web%20Sites%5CRice%5Ccom
spring%5Cds%5ClistFW.zip

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.

3.5 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)?

Chapter 4

Abstract Factory Design Pattern[1]

One of the nice things about **abstraction** is that it lets you take care of the bigger picture and worry about the details later. Wouldn't it be great if we could do that when creating objects? It sure would be handy to be able to rely upon some other class to fill in the details for you. With the help of the **Abstract Factory Pattern**, we can.

The great advantage of this is that we don't have to worry about what kind of car we're building. Let's say that you are making various types of car, and each car has different types of components you need to worry about. For example, say that you have the ability to build a Cadillac Dreadnaught, a Citroën BX-TRD, and a Fiat Avanti. Each type of vehicle has the same overall structure that all cars share in common: i.e. an engine, wheels, steering, brakes, etc. For the sake of simplicity, let's say that a car consists of three components: Engine, Chassis, and Steering. Naturally, you can't have the same types of Engine, Chassis or Steering in each car, because then the cars would all be the same. It won't do to mix-and-match components, either. You need all Cadillac parts to go with the Cadillac, and all Citroën parts to go with the Citroën, etc. The different parts of the cars don't need to do the same thing, either: they need merely conform to an interface. Let's encapsulate the decision making process for assigning the parts when you build a car.

Should the main car building program care what kind of car is being built? No, it shouldn't. It should just expect the car to behave as all cars should, and not care about the specifics. All of the knowledge about the specific cars we're building is not contained in the main class of the program, but can be updated and changed easily, since the information is only in one place. This gives us great flexibility, since we are not limited in our choices at all, and can make many different types of car. Let's say that we have a simulated car-building plant. It can produce one type of car at a time, but can produce any type of car, so long as the workers have the proper directions. Each set of plans will dictate how each component of a particular car is built. So, in order to change the type of car we're producing, we only need to switch the directions. All of the factory workers will follow the new directions, producing a new car, based upon the directions.

That's nice, you say, but won't it get confusing with all kinds of different directions running around? Which set do we follow? The solution is pretty simple. We have one main set of directions which everyone refers to, and we can change what that set of directions looks like. We expect each set of directions to provide certain information about how to build a car. It should comply to an abstract set of directions that we have in mind. Each real set of directions should have the same structure as the abstract directions. Let's call the sets of directions Factories. A Factory will build a certain type of car, depending upon the type of factory. The specification for what directions should contain would be called an Abstract Factory.

[1]This content is available online at <http://cnx.org/content/m17203/1.3/>.

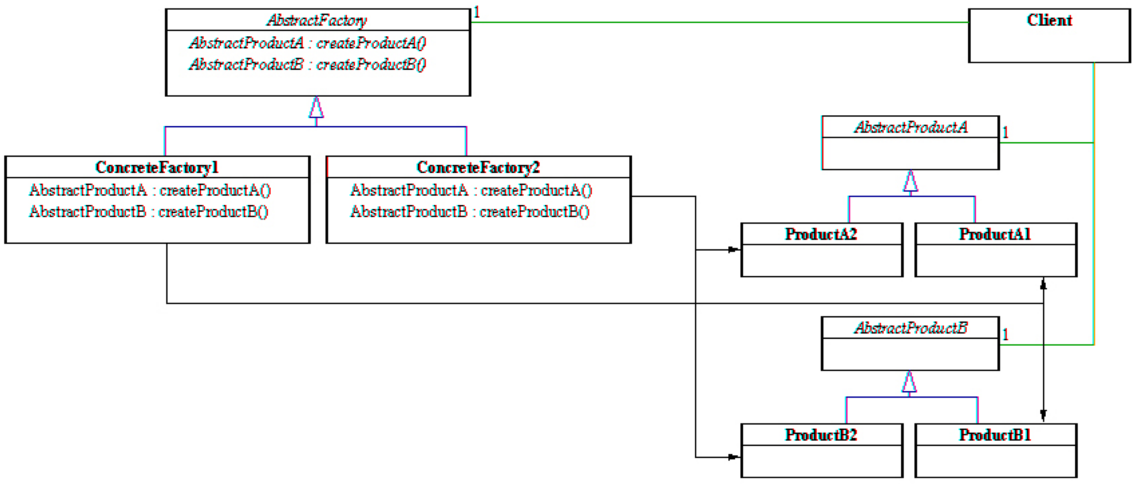

AbstractFactory
AbstractProductA : createProductA()
AbstractProductB : createProductB()
1
Client
ConcreteFactory1
AbstractProductA : createProductA()
AbstractProductB : createProductB()
ConcreteFactory2
AbstractProductA : createProductA()
AbstractProductB : createProductB()
AbstractProductA
1
ProductA2
ProductA1
AbstractProductB
1
ProductB2
ProductB1

Figure 4.1: The Client here only knows about the `AbstractFactory` and `AbstractProducts`, shielding it from needing to know about the actual behavior of the `ConcreteFactorys` and actual `Products`

A good analogy for this is a pasta maker. A pasta maker will produce different types of pasta, depending what kind of disk is loaded into the machine. All disks should have certain properties in common, so that they will work with the pasta maker. This specification for the disks is the Abstract Factory, and each specific disk is a Factory. You will never see an Abstract Factory, because one can never exist, but all Factories (pasta maker disks) inherit their properties from the abstract Factory. In this way, all disks will work in the pasta maker, since they all comply with the same specifications. The pasta maker doesn't care what the disk is doing, nor should it. You turn the handle on the pasta maker, and the disk makes a specific shape of pasta come out. Each individual disk contains the information of how to create the pasta, and the pasta maker does not.

Chapter 5

Composite Design Pattern[1]

5.1 Description

The **Composite Design Pattern** allows a client object to treat both single components and collections of components identically. It accomplishes this by creating an abstraction that unifies both the single components and composed collections as abstract equivalents. Mathematically, we say that the single components and composed collections are **homomorphically** equivalent (from the Latin: *homo* – same and *morph* – form ==> to have the same form).

This equivalence of single and composite components is what we call a **recursive data structure**. In recursive data structures, objects are linked to other objects to create a total object structure made of many "nodes" (objects). Since every node is abstractly equivalent, the entire, possibly infinitely large and complex data structure can be succinctly described in terms of just three distinct things: the single components, the composite components and their abstract representation. This massive reduction in complexity is one of the cornerstones of computer science.

The Composite Design Pattern is an object-oriented representation of a recursive data structure.

5.2 UML Diagram and Example

In the UML class diagram below, the `Client` uses an abstract component, `AComponent`, for some abstract task, `operation()`. At run-time, the `Client` may hold a reference to a concrete component such as `Leaf1` or `Leaf2`. When the operation task is requested by the `Client`, the specific concrete behavior with the particular concrete component referenced will be performed.

[1] This content is available online at <http://cnx.org/content/m14795/1.1/>.

UML Diagram of Composite Design Pattern Example

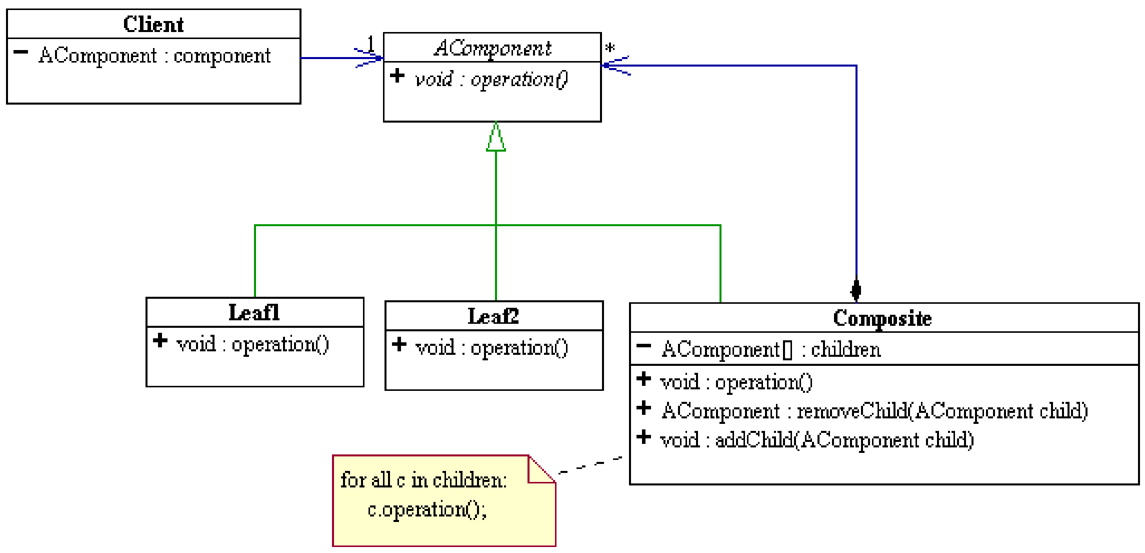


Figure 5.1: An example of a composite design pattern implementation with 2 single, concrete components and one composite, collection component.

The `Composite` class is a concrete component like `Leaf1` and `Leaf2`, but has no `operation()` behavior of its own. Instead, `Composite` is composed with a collection of other abstract components, which may be of any other concrete component type including the composite itself. The unifying fact is that they are all abstractly `AComponents`. When the `operation()` method of a `Composite` object is called, it simply dispatches the request sequentially to all of its "children" components and perhaps, also does some additional computations itself. For instance, a `Composite` object could hold references to both a `Leaf1` and a `Leaf2` instance. If a client holds a reference to that `Composite` object and calls its `operation()` method, the `Composite` object will first call operation on its `Leaf1` instance and then `operation()` on its `Leaf2` instance. Thus composite behavior of `Leaf1` plus `Leaf2` behaviors is achieved without either duplicating code or by having the `Client` object knowing that the two leaf components were involved.

Composite patterns are often used to represent recursive data structures. The recursive nature of the Composite structure naturally gives way to recursive code to process that structure.

NOTE: The collection of `AComponents` held by `Composite`, "children", is shown above as an array. However, the behavior of a Composite pattern is independent of exactly how the collection of `AComponents` is implemented. If access speed is not an issue, a vector or a list may be a better implementation choice. The `addChild()` and `removeChild()` methods are optional.

In *Design Patterns*, the abstract component `AComponent` is shown as having accessor methods for child `AComponents`. They are not shown here because it is debatable as to whether one wants the `Client` to fundamentally view the `AComponent` as a single component or as a collection of components. *Design Patterns* models all `AComponents` as collections while the above design models them all as single components. The exact nature of those accessor methods is also debatable.

Chapter 6

List Structure and the Composite Design Pattern[1]

6.1 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.

6.2 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

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

- bread
- 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.

6.3 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

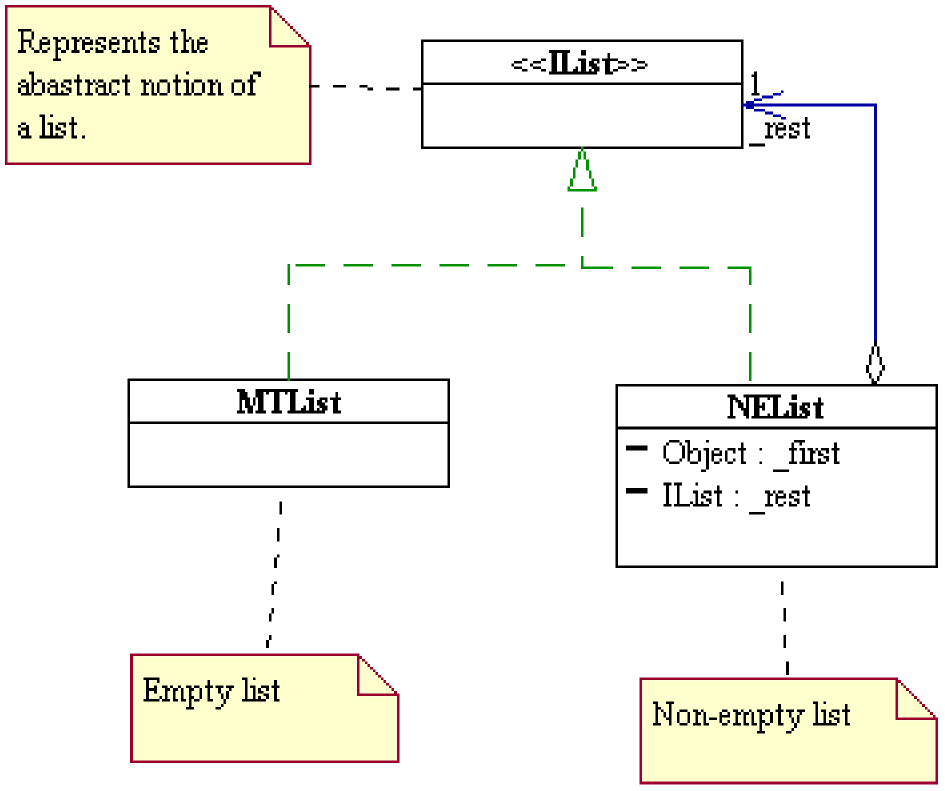


Figure 6.1: A list can be represented using the composite design pattern

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 6.1

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.)

6.4 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.

6.5 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.

6.5.1 In class role-acting exercises:

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

Chapter 7

List Structure and the Interpreter Design Pattern[1]

7.1 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.

7.1.1 What can a list do?

7.1.1.1 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

[1]This content is available online at <http://cnx.org/content/m15110/1.1/>.

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.

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

```
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 7.1

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.

7.1.2 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.

interface IList	
public abstract returnType methodName(parameter list); // returnType may be 'void'	

MTList
 // no data
	NEList
	  Object _first;
	  IList _rest;

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

Table 7.2

7.1.2.1 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.

Chapter 8

Null Design Pattern[1]

The **Null Pattern** is perhaps the most "intelligent" pattern of them all. It knows exactly what to do all the time, every time: nothing. Its usefulness is a little more subtle than that of other Design Patterns, but it can be used in many different situations.

The Null Pattern is somewhat difficult to describe, since it resides in an abstract hierarchy tree, having no particular place at all, but occupying many roles. It is somewhat like the mathematical concept of zero: it is a placeholder, but is in itself nothing, and has no value. However, this means that Null is abstractly equivalent to any of the other concrete classes in the abstract hierarchy. Thus, it can be treated identically to any other class by the system. This gives consistent and predictable behavior for the null situation. Some objects' behavior will depend upon its values, and since the Null Pattern has no values, it knows exactly what to do every time. It really does do **nothing**. And that's the most reliable code you'll ever see.

One of the uses of the Null pattern is in the context of a Strategy Pattern[2]. You have several behaviors that you want your host object to perform, but what if you want it to just do nothing? That's where you use a Null Pattern. The Null Pattern complies with the interface for a Strategy, but the bodies of its methods are blank, so it does nothing.

Another popular use of the Null Pattern is in the context of a State Pattern (Chapter 9). The easiest way to separate objects into two categories is along the difference between Null and Non-Null. Non-Null objects have values and do things, whereas Null ones do not. When working with a singly linked list, the Null node of the list is arguably the most important node in the list. It is the only node in the entire list that "knows what to do." When you perform a tail recursive operation on a singly linked list, each node asks the remaining portion of the list for more information in order to complete the assigned task. (remember: a singly linked list is defined in terms of nodes. Each node has a value (**car**) and a reference to the rest of the list. (**cdr**) The end of the list is reached when a node has no cdr.) It is only upon reaching the last node, the Null node, that any actual operations can be completed. The Null node will know exactly what to do: Nothing. Based upon this, the recursive calls made upon the elements of the list can be resolved, since there is a definite value to work with.

Let's say that you have a list of objects that need to be drawn in a certain Container object (context). These objects are extensions of list nodes, and have only one extra method in them, which is the paint method. Each node's paint method consists of it asking its cdr to draw, and then actually drawing whatever it is that they draw onto the context. When you ask the first node to paint, it will ask the next node to paint, which will ask the next node to paint, and so on. How can we make this useful and not just a silly infinite delegation? The paint method of the last (and, supposedly, Null) node is empty. This means that it will do nothing, ant then return to the previous method that called it, so that all of the recursive calls down the list can resolve. This allows us to have an indefinite number of things to be painted in the list that can appear in any order. The only node in the list that matters is the last, Null node, which is the only one that

[1]This content is available online at <http://cnx.org/content/m17227/1.3/>.
[2]"Strategy Design Pattern" <http://cnx.org/content/m17037/latest/>

actually knows what to do.

Chapter 9

State Design Pattern[1]

Objects are often discussed in terms of having a "state" that describes their exact conditions in a given time, based upon the values of their properties. The particular values of the properties affect the object's behavior. For instance, one can say that the exact behavior of an object's `getColor()` method is different if the "color" property of the given object is set to "blue" instead of "red" because `getColor()` returns a different value in the two situations.

Furthermore, the object may make decisions at run time as to exactly what to do dependent upon the values its properties possess. For instance, if the sky is blue `(sky.setColor(Color.blue))`, then the sun should be visible.

```
public boolean sunIsVisible() {
    if(getColor()==Color.blue) {
        return true;
    }
    else {
        return false;
    }
}
```

One issue with the above solution is that it is a hard-coded logic solution, not an architected solution. The sky does not intrinsically behave a certain way if it is blue, but rather it should figure out what to do in that situation.

Wouldn't it be better if the sky intrinsically acted properly if it were blue? One could imagine two objects: a `SkyBlue` and a `SkyNonBlue`. The `SkyBlue` class' `sunIsVisible()` method would always return true while the `SkyNonBlue` version would always return false.

What one needs now is the ability for a sky object to dynamically (i.e. at run time) change its class to/from `SkyBlue` and `SkyNonBlue`. What we'd like to accomplish is called "**dynamic reclassification**".

We've seen code that does change its specific behavior depending on what particular strategy was installed. So, the `setColor()` method could install a strategy that would always return true if its `sunIsVisible()` method were to be called.

Exercise 9.1 ***(Solution on p. 33.)***

But does the user of the Sky class care about the stratregy?

[1]This content is available online at <http://cnx.org/content/m17047/1.3/>.

9.1 The State Design Pattern is a fully encapsulated, self-modifying Strategy Design Pattern.

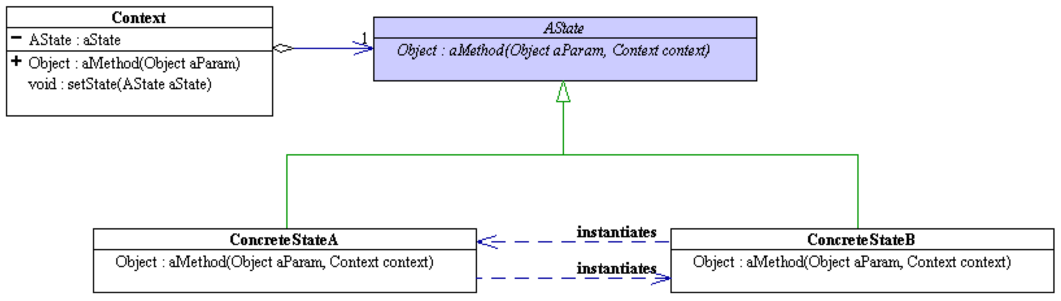

Context
− AState : aState
+ Object : aMethod(Object aParam)
void : setState(AState aState)
1
AState
Object : aMethod(Object aParam, Context context)
ConcreteStateA
Object : aMethod(Object aParam, Context context)
instantiates
instantiates
ConcreteStateB
Object : aMethod(Object aParam, Context context)

Figure 9.1: UML Class Diagram of the State Design Pattern

One design pattern that is used very often in conjunction with the state pattern is the Null Object Pattern (Chapter 8).

Notice these things about the pattern:

1. Any methods whose behaviors depend on the state of the object are simply delegated on in to the state, and handled there. Thus you will see the same methods in the context as in the states. Since the states are separate objects from the context, all the properties of the context need to have accessor methods that are at least package visible.
2. The "**Context**" object needs to add a "**set**" accessor method so the states can modify which state is the active state. This method would be package visible so as to encapsulate the behavior away from the sight of the user.

Solutions to Exercises in Chapter 9

Solution to Exercise 9.1 (p. 31)

Of course not. The user only cares that it does its job.

Chapter 10

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
- Dies when hit by a vehicle.

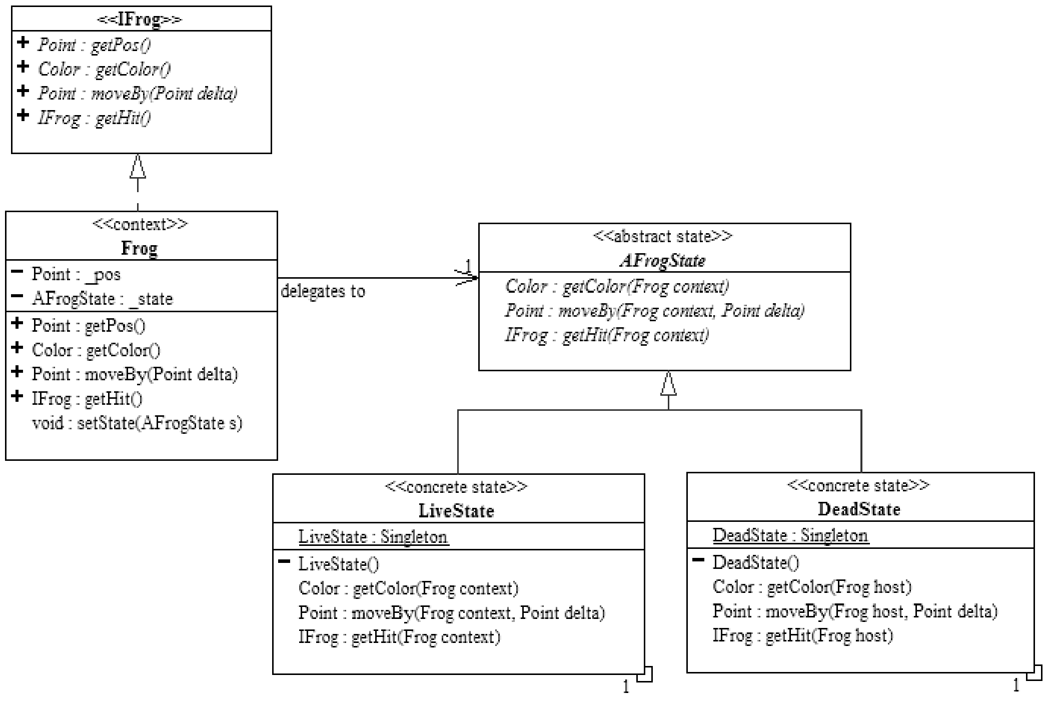
On the other hand, a dead frog

[1]This content is available online at <http://cnx.org/content/m17225/1.5/>.
[2]http://www.gamesgnome.com/arcade/frogger/

- 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]



<<IFrog>>
+ Point : getPos()
+ Color : getColor()
+ Point : moveBy(Point delta)
+ IFrog : getHit()
<<context>>
Frog
− Point : _pos
− AFrogState : _state
+ Point : getPos()
+ Color : getColor()
+ Point : moveBy(Point delta)
+ IFrog : getHit()
void : setState(AFrogState s)
delegates to
1
<<abstract state>>
AFrogState
Color : getColor(Frog context)
Point : moveBy(Frog context, Point delta)
IFrog : getHit(Frog context)
<<concrete state>>
LiveState
LiveState : Singleton
− LiveState()
Color : getColor(Frog context)
Point : moveBy(Frog context, Point delta)
IFrog : getHit(Frog context)
1
<<concrete state>>
DeadState
DeadState : Singleton
− DeadState()
Color : getColor(Frog host)
Point : moveBy(Frog host, Point delta)
IFrog : getHit(Frog host)
1

Figure 10.1

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 10.1: Question:) below.)

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

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

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

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. (Chapter 9)

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**.

10.1 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?

10.2 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.....

Chapter 11

Union Design Pattern[1]

11.1 UML Diagram and Characteristics

In general, the UML class diagram for the union design pattern looks like such:

Union Design Pattern

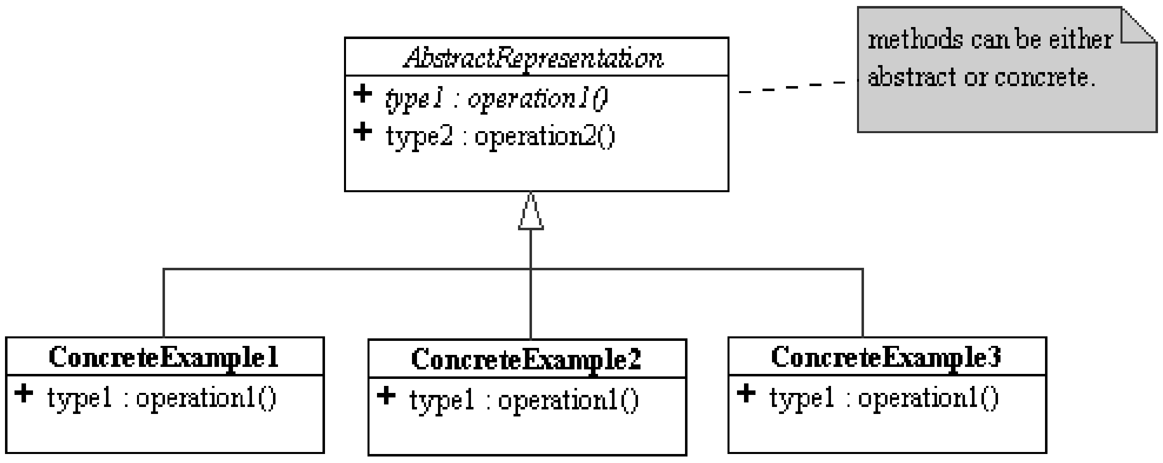


Figure 11.1: UML diagram of the union design pattern

The characteristics of the union pattern are

- 1 abstract superclass that represents the abstraction of all the subclasses. (Note that the superclass could be an interface.) The superclass can never instantiated because it does not represent any particular concrete entity. The methods of the superclass could be abstract (variant behaviors defined in the subclasses) or concrete (pure invariant behavior)–See below .
- 2 or more subclasses (It is arguable whether or not a single subclass would qualify–if one considers that it might be possible that other subclasses could exist and are simply not coded yet, then one could

[1]This content is available online at <http://cnx.org/content/m14622/1.1/>.

argue that there is actually more than one subclass.) A subclass could be abstract, in which case, it would represent another abstraction layer.

Corollary: Another way of thinking about the union pattern is to consider that the abstract superclass represents **any** of the subclasses. That is, anywhere the superclass is referenced, an instance of any of subclasses could be used. This is the essence of **polymorphism**[2] .

11.2 Examples

Consider the following examples:

1. Oranges, apples, pears, peaches, and mangos can all be abstractly represented as "fruit". Note that there is no such thing as a fruit that is not of some concrete sort, e.g. plum or strawberry. The concept of "fruit" is an abstraction that represents the union of oranges, apples, plums, etc. The notion of a fruit contains only the essence of what is common to all the concrete instances, such as containing seeds, having a sweet juicy flesh, etc. On the other hand, the notion of a fruit does not include specific information that pertains to only oranges–the number of orange sections it has, for instance–or to only strawberries – the nature of its vine, for instance– or to only any other particular kind of fruit. To talk about a fruit is to talk about all fruit at once – the union of all types of fruit.

Union of different fruits

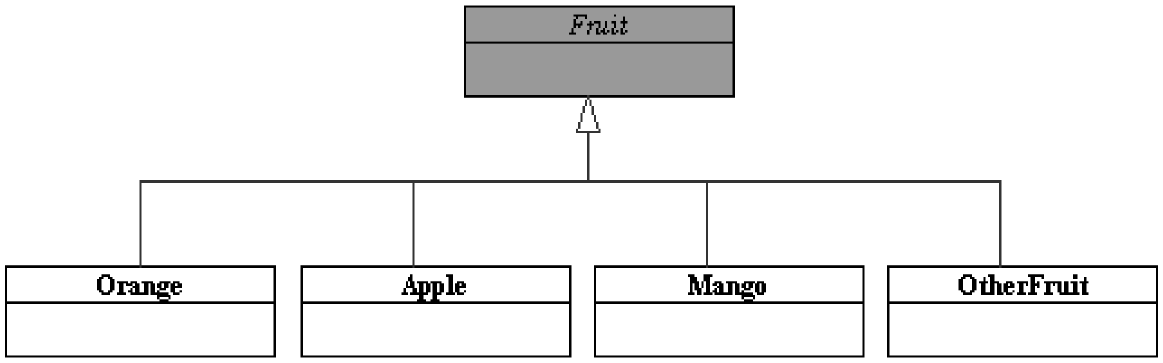


Figure 11.2: `Fruit` represents the union or abstraction of many types of specific things

   In the above UML class diagram, we see that oranges, apples, mangos and other concrete fruit classes all inherit from an abstract fruit class.
2. Keyboards, mice, microphones, network ports and speakers are all input/output ("I/O") devices. An I/O device cannot exist without being a specific, concrete type of device, such as a keyboard or a speaker. Thus the concept of an "I/O device" is an abstraction of the union of all keyboards, printer ports, microphones, etc. The notion of an I/O device contains high-level issues about moving data into and out of a computer, but not any low-level concrete information about how it is done such as via a parallel or serial bit stream or how the data is formatted. "I/O device" refers to all possible types of input/output device.

[2]http://www.exciton.cs.rice.edu/JavaResources/Oop/polymorph.htm

Union of different I/O devices

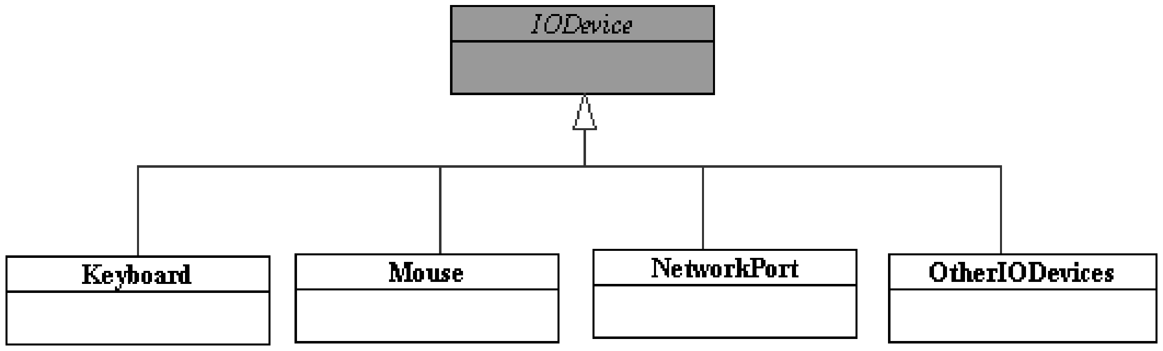


Figure 11.3: `IODevice` represents the union of many different types of specific devices

As before, the UML class diagram shows us that the keyboard, mouse, network port and other concrete I/O devices inherit from an abstract I/O device class.

11.3 Relationship to Abstraction and Variant/Invariant Principles

11.3.1 Abstraction Layers

One of the most important things that the union pattern does for an OO design is to define layers of abstraction. The abstract superclass represents a higher level of abstraction than the concrete subclasses. The union pattern thus defines two distinct abstraction levels. At any given moment, a program will that uses the classes involved in the union pattern will be running at either the lower, more concrete abstraction level or the higher, more abstract level.

Good OO design pays careful attention to maintaining a consistent abstraction level in any given section of code. Changing abstraction levels in the middle of a process, particularly the decreasing of abstraction, effectively nullifies the power and flexibility of having made the abstraction in the first place.

WARNING!

Simply having an abstract superclass does not automatically imply the existence of the union pattern!

Sometimes abstract superclasses are used to encapsulate and/or centralize shared or common code used in the subclasses. This gathering of code from the subclasses and relocating it to a superclass, often refered to as "**hoisting**", does not guarantee that the superclass is an abstraction of the subclasses. For instance, it would be convenient to put a field in "fruit" that tells us how many sections it has, because so many fruit, such as oranges, grapefruit, etc, form in sections. But the very fact that some fruit, such as mangos are not describable in terms of sections, means that to include such a field in "fruit" would compromise it as an abstraction of all fruit. When performing hoisting, one must be very careful to understand the difference between "universalizing" (trying to do everything) and "abstracting" (capturing the essence of the problem).

11.3.2 Variant vs. Invariant

One of the most fundamental OO design decisions is where exactly to put the dividing line between the **invariant**, unchanging aspects of the problem and the **variant**, changing aspects. The union design pattern provides a clear representation of one type of variant-invariant separation.

The abstract superclass is a representation of the essence that common to all possible concrete subclasses. It thus represents the invariant aspects of all elements in the union.

The concrete subclasses, while all equivalent at a higher, more abstract level, are different from each other in some manner. They thus represent the variant aspects of the problem.

Code that works at the abstraction level of the superclass is thus invariant code. It is capable of working with any instance of the concrete subclasses because it only deals with the invariant behavior of the superclass. The actual total behavior of the system is the invariant behavior of the abstract superclass plus the variant behaviors provided polymorphically by the instance of the concrete subclass being used.

Chapter 12

Visitor Design Pattern[1]

12.1 1. Decoupling Algorithms from Data Structures

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

[1]This content is available online at <http://cnx.org/content/m16707/1.1/>.

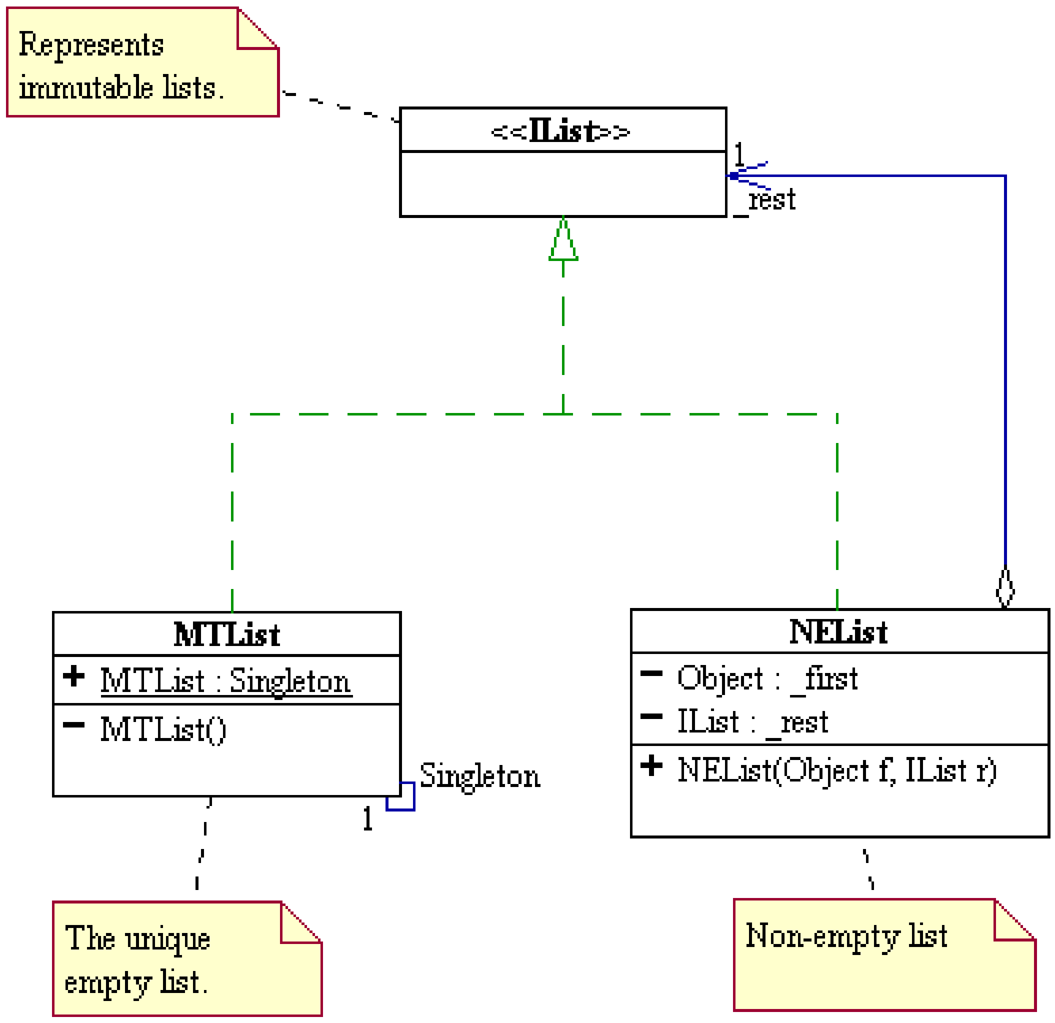


Figure 12.1

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!

12.2 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?

12.2.1 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.)

12.2.2 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.

```
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!
}
```

continued on next page

```
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 12.1

12.2.3 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.

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

```
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 12.2

12.2.4 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.

```
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);
}
}
```

```
// 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 12.3

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 perforns 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.

12.3 3. The Visitor Pattern

The visitor pattern[2] 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 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

[2]http://www.exciton.cs.rice.edu/JavaResources/DesignPatterns/VisitorPattern.htm

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!

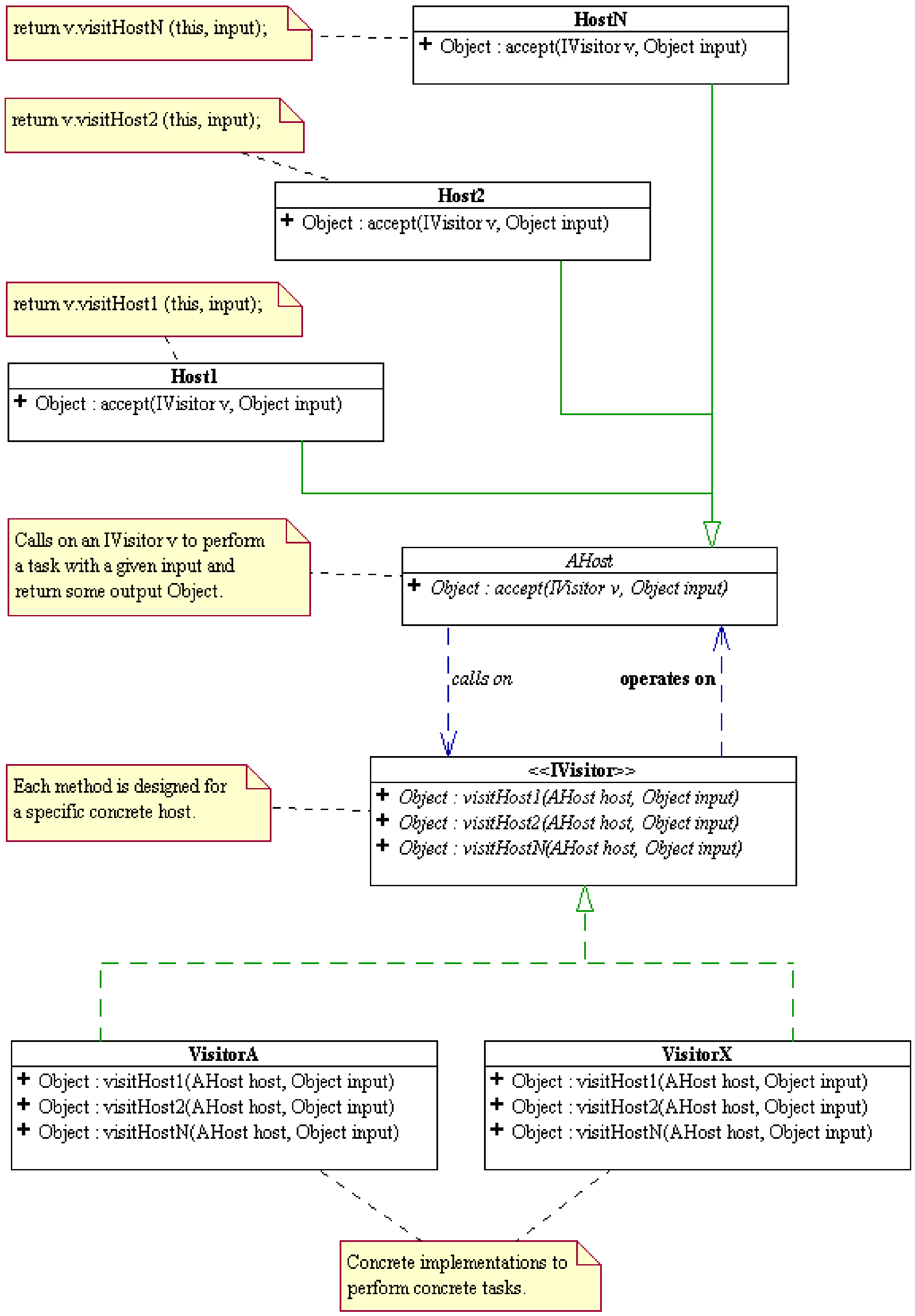

return v.visitHostN (this, input);
HostN
+ Object : accept(IVisitor v, Object input)
return v.visitHost2 (this, input);
Host2
+ Object : accept(IVisitor v, Object input)
return v.visitHost1 (this, input);
Host1
+ Object : accept(IVisitor v, Object input)
Calls on an IVisitor v to perform a task with a given input and return some output Object.
AHost
+ Object : accept(IVisitor v, Object input)
calls on
operates on
Each method is designed for a specific concrete host.
<<IVisitor>>
+ Object : visitHost1(AHost host, Object input)
+ Object : visitHost2(AHost host, Object input)
+ Object : visitHostN(AHost host, Object input)
VisitorA
+ Object : visitHost1(AHost host, Object input)
+ Object : visitHost2(AHost host, Object input)
+ Object : visitHostN(AHost host, Object input)
VisitorX
+ Object : visitHost1(AHost host, Object input)
+ Object : visitHost2(AHost host, Object input)
+ Object : visitHostN(AHost host, Object input)
Concrete implementations to perform concrete tasks.

Figure 12.2

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.

12.4 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.[3] Click here to see the code[4] .

[3]http://www.owlnet.rice.edu/~comp201/08-spring/lectures/visitor/listFW/doc/
[4]http://www.owlnet.rice.edu/~comp201/08-spring/lectures/lec17/listFW.zip

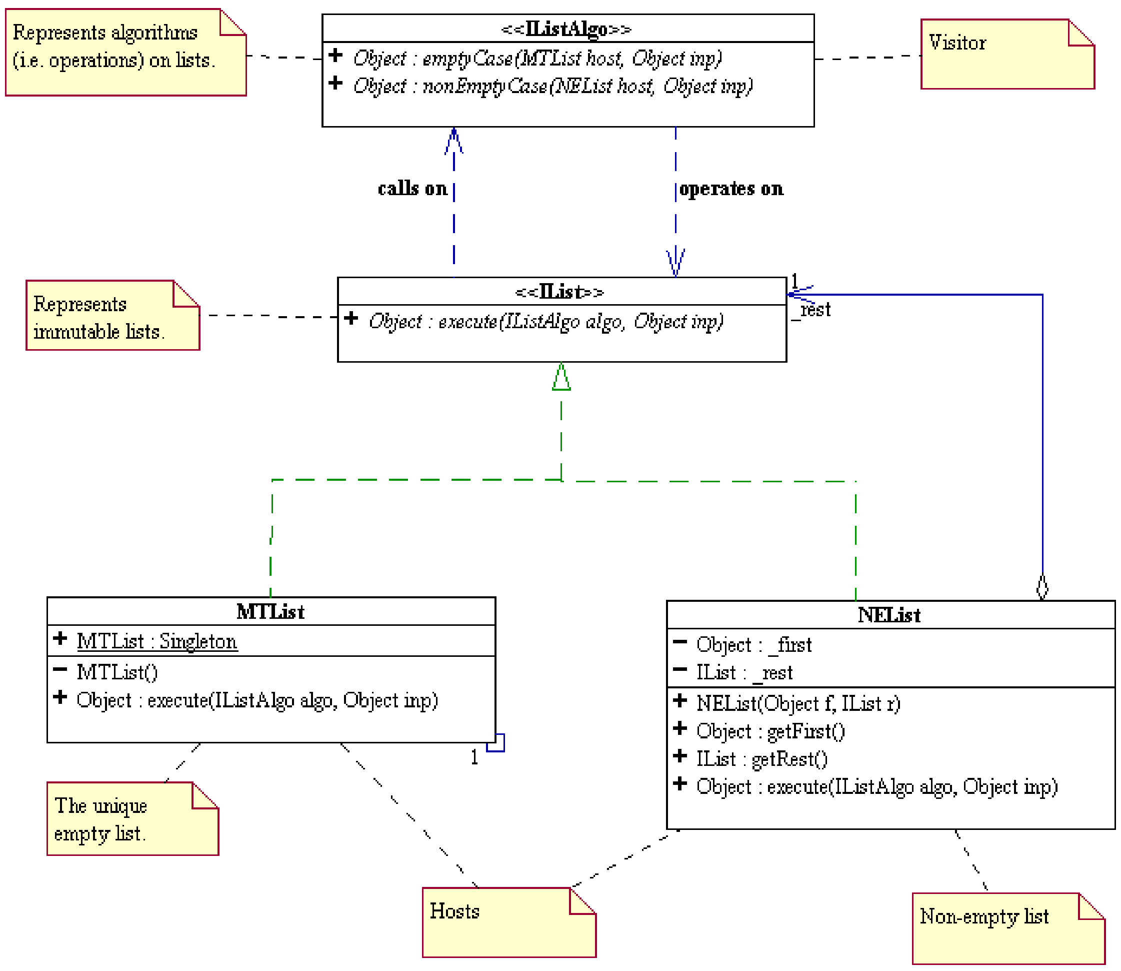

Represents algorithms (i.e. operations) on lists.
<<IListAlgo>>
+ Object : emptyCase(MTList host, Object inp)
+ Object : nonEmptyCase(NEList host, Object inp)
Visitor
calls on
operates on
Represents immutable lists.
<<IList>>
+ Object : execute(IListAlgo algo, Object inp)
1
_rest
MTList
+ MTList : Singleton
− MTList()
+ Object : execute(IListAlgo algo, Object inp)
1
NEList
− Object : _first
− IList : _rest
+ NEList(Object f, IList r)
+ Object : getFirst()
+ IList : getRest()
+ Object : execute(IListAlgo algo, Object inp)
The unique empty list.
Hosts
Non-empty list

Figure 12.3

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.

12.5 5. List Visitor Examples

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

/**
 * 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 12.4

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. * @stereotype visitor */ public class ToStringAlgo implements IListAlgo { public static final ToStringAlgo Singleton = new ToStringAlgo(); private ToStringAlgo() { }	

/**
 * Returns "()".
 */
public Object emptyCase(MTList host, Object... inp) {
return "()";
}
	/**
	 * Passes "(" + first to the rest of IList and asks for he
	 */
	public Object nonEmptyCase(NEList host, Object... ~inp) {
	return host.getRest().execute(ToStringHelper.Singleton, "
	}
	}

Table 12.5

/** * 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() { }	
/** ~* 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 ap * and recurse! */ public Object nonEmptyCase(NEList host, Object... acc) { return host.getRest().execute(this, acc[0] + ", " + host.g } }

Table 12.6

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 give
     * @param f the first element of this NEList.
     * @param r != null, the rest of this NEList.
     */
    public NEList(Object f, IList r) {
_first = f;
_rest = r;
    }

    /**
     * Returns the first data element of this NEList.
     * This method is marked as final to prevent all subcl
     * from overriding it.
     * Finalizing a method also allows the compiler to gen
     * more efficient calling code.
     */
    public final Object getFirst() {
    return _first;
    }

    /**
     * Returns the first data element of this NEList.
     * 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 IList getRest() {
     return _rest;
    }

    /**
     * Calls the nonEmptyCase method of the IListAlgo para
     * passing to the method itself as the host parameter
     * given input as the input parameter.
     * This method is marked as final to prevent all subcl
     * overriding it. Finalizing a method also allows the
     * to generate more efficient calling code.
     */
    public final Object execute(IListAlgo algo, Object...
    return algo.nonEmptyCase(this, inp);
```

Table 12.7

Download the above code here[5] .

[5] http://www.owlnet.rice.edu/∼comp201/08-spring/lectures/lec17/listFW.zip

Chapter 13

in Actions

13.1 Design Patterns for Sorting[1]

The following discussion is based on the the SIGCSE 2001 paper by Nguyen and Wong, "Design Patterns for Sorting"[2].

Merritt's Thesis

In 1985, Susan Merritt proposed that all comparison-based sorting could be viewed as "Divide and Conquer" algorithms.[3] 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 13.1: 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.

[1]This content is available online at <http://cnx.org/content/m17309/1.4/>.

[2]D. Nguyen and S. Wong, "Design Patterns for Sorting," SIGCSE Bulletin 33:1, March 2001, 263-267

[3]S. Merritt, "An Inverted Taxonomy of Sorting Algorithms," Comm. of the ACM, Jan. 1985, Volume 28, Number 1, pp. 96-99

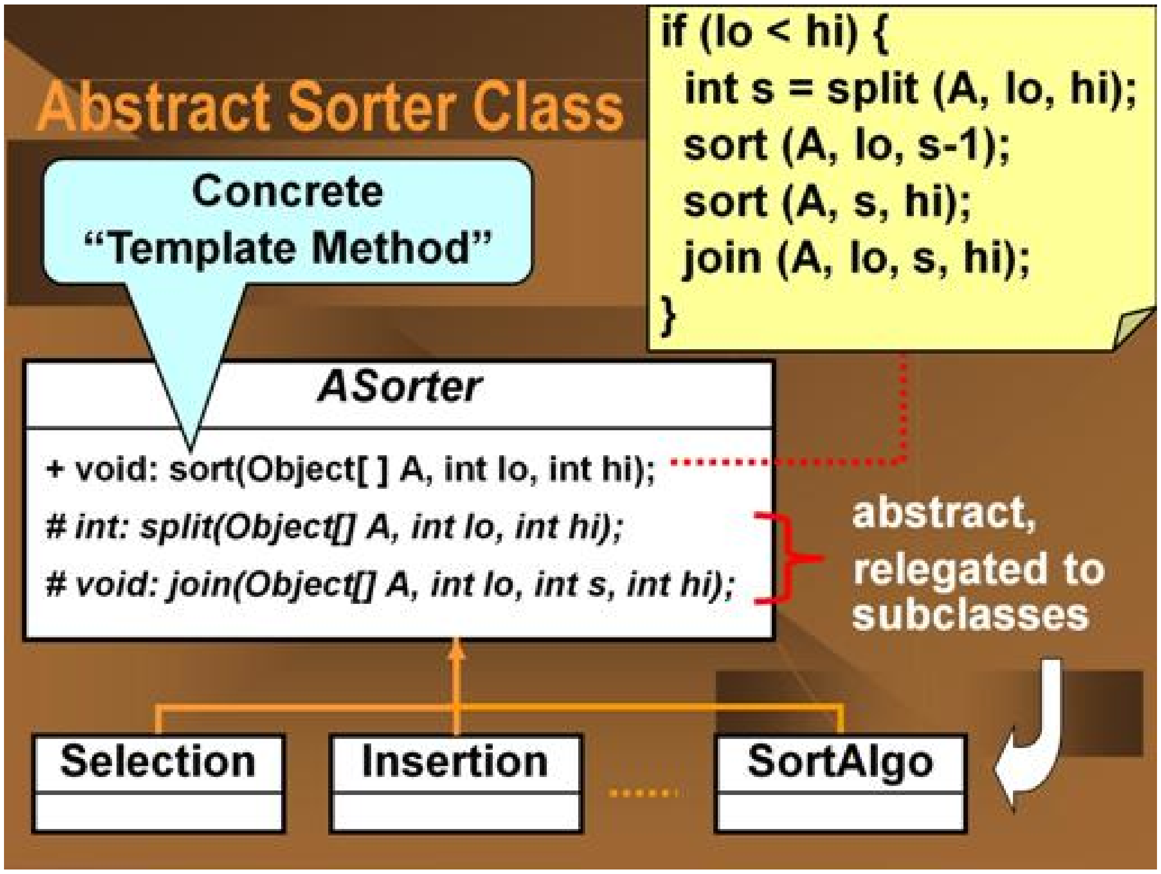
Merritt's thesis is potentially a very powerful method for studying and understanding sorting. In addition, Merritt's abstract characterization of sorting exhibits much object-oriented (OO) flavor and can be described in terms of OO concepts.

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.

Abstract Sorter Class



```
if (lo < hi) {
  int s = split (A, lo, hi);
  sort (A, lo, s-1);
  sort (A, s, hi);
  join (A, lo, s, hi);
}
```

Abstract Sorter Class
Concrete "Template Method"
ASorter
+ void: sort(Object[ ] A, int lo, int hi);
# int: split(Object[] A, int lo, int hi);
# void: join(Object[] A, int lo, int s, int hi);
abstract, relegated to subclasses
Selection
Insertion
SortAlgo

Figure 13.2: 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: **ASorter class**

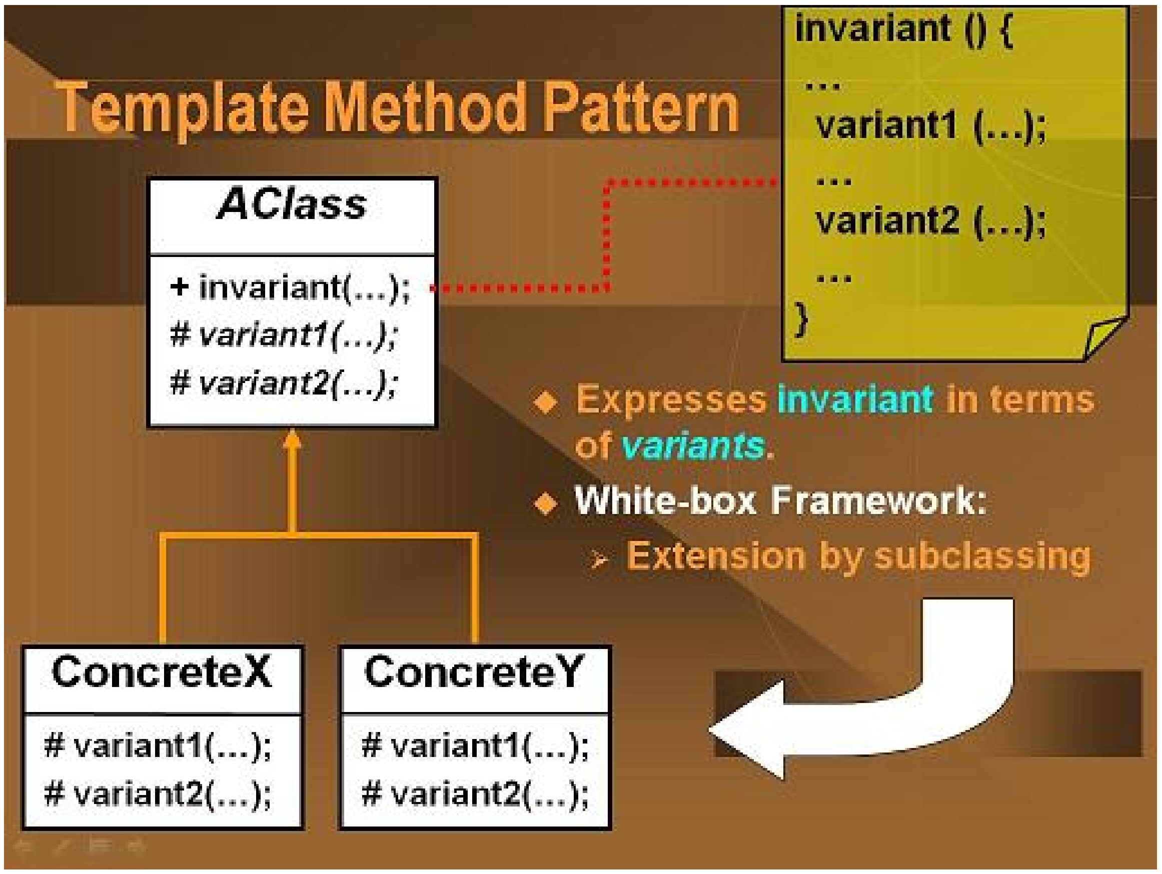
```
package sorter;

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;
 }
}
Java code  for ASorter, the abstract superclass for all concrete sorters and the implementation of Merr
```

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



```
invariant () {
 ...
  variant1 (...);
  ...
  variant2 (...);
  ...
}
```

- Expresses invariant in terms of *variants*.
- White-box Framework:
  - Extension by subclassing

Figure 13.3: 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**.

13.1.1 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 13.1: 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 13.4: 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 13.5: 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.

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)
 {
 }
}
```

`Java implementation of the SelectionSorter class.  The split method splits off the extrema (minimum, he`

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 13.2: 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 13.6: 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 13.7: 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.

Here is the full code for the insertion sorter: **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;
 }
}
```

`Java implementation of the selection sorter.  The split method simply splits off the right-most element`

Exercise 13.1 ***(Solution on p. 67.)***

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[4] . However, it can be done is a very straightforward manner. There are a number of viable solutions. Hint: think about the State Design Pattern (Chapter 9).

For more examples, please see download the demo code[5]. Please note that the ShakerSort code is disabled due to its use as a student exercise.

[4]http://en.wikipedia.org/wiki/Cocktail_sort
[5]See the file at <http://cnx.org/content/m17309/latest/Sorter.zip>

Solutions to Exercises in Chapter 13

Solution to Exercise 13.1 (p. 66)

The solution is left to the student but is available from the authors if proof of non-student status is provided.

Chapter 14

(Untitled)

Glossary

H

hoisting

The act of removing equivalent methods or fields common to all subclasses and moving them to their superclass. While this saves one from repeating code over and over again in the subclasses, the act of hoisting does not guarantee that the hoisted fields or methods are indeed true invariants of the subclasses. That is hoisting does not guarantee that the hoisted items are part of the abstraction of the subclasses. This is because hoisting assumes that all possible subclasses are known at the time the hoisting takes place, which is not always true.

I

invariant

Refers to anything that is the same across a set of classes. This includes attributes, concrete behaviors (methods) and abstract behaviors. The invariant properties constitute the abstraction of a set of classes.

P

polymorphism

The ability for a subclass instance to be used wherever its superclass (or implemented interface) is required. This is because all subclasses are abstractly equivalent to their superclasses. However, the use of different subclass instances in the same situation will produce differing resultant program behaviors because each subclass will behave in their own, differing concrete manners, even though those concrete behaviors are all abstractly equivalent. For more information, see the Wikipedia article on polymorphism[1] and this web page on polymorphism and abstraction[2] .

V

variant

Refers to anything that is **not** the same across a set of classes. This includes attributes and concrete behaviors (methods). The variant properties create the unique definition of a class.

[1]http://en.wikipedia.org/wiki/Polymorphism_in_object-oriented_programming
[2]http://www.exciton.cs.rice.edu/JavaResources/Oop/polymorph.htm

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

Design Patterns
just exploring