Composite

The Composite pattern organizes objects into a tree data structure where leaf nodes represent individual objects and interior nodes represent compositions of individual objects and/or other compositions. What makes this organization a design pattern rather than a standard tree data structure is the fact that objects and compositions share the same interface. The Composite pattern lets clients treat individual objects and compositions of objects uniformly.

Composite Pattern

The Composite pattern lets clients treat individual objects and compositions of objects uniformly.

Introduction

When sending gifts to family and friends around a holiday, you can save on shipping costs by consolidating packages destined for the same general area.

For example, let's say your aunt and uncle live in San Francisco and your parents live in Minneapolis a few blocks away from your sister and her family who live in St Paul:

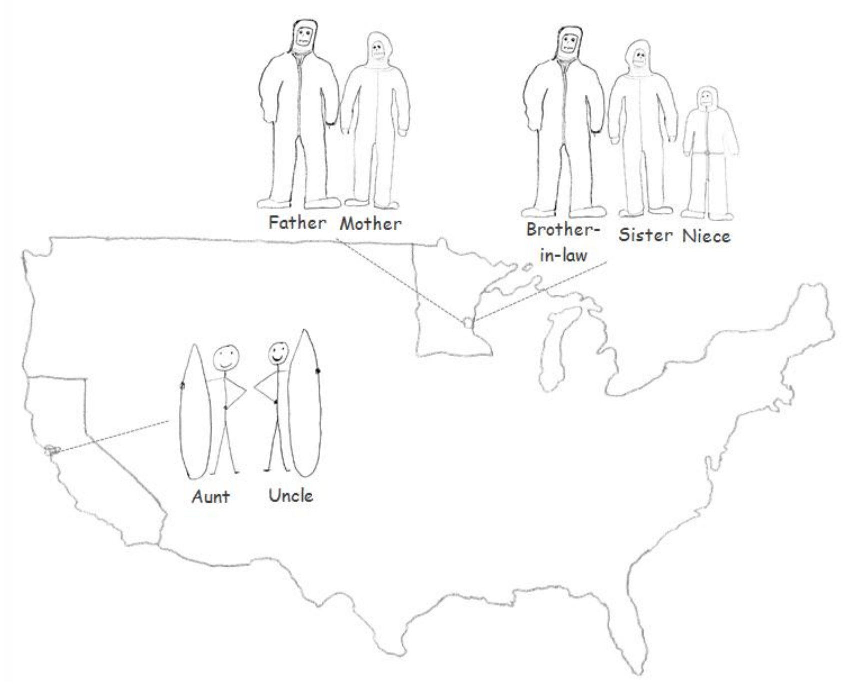
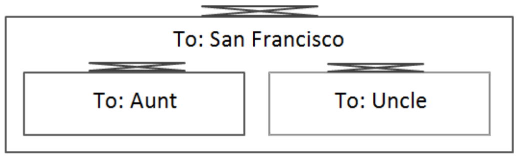
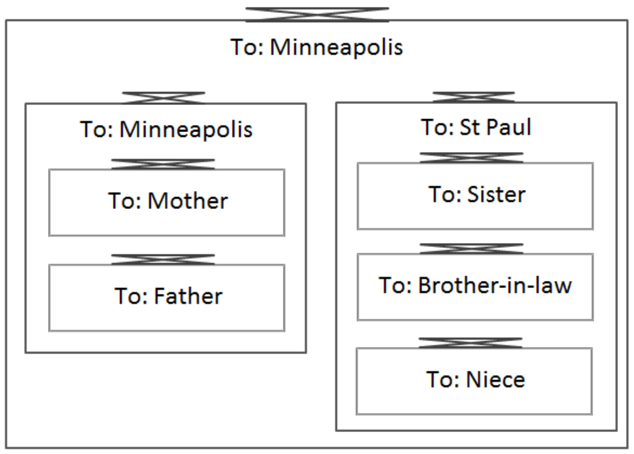


Figure x. Final destinations for holiday gifts

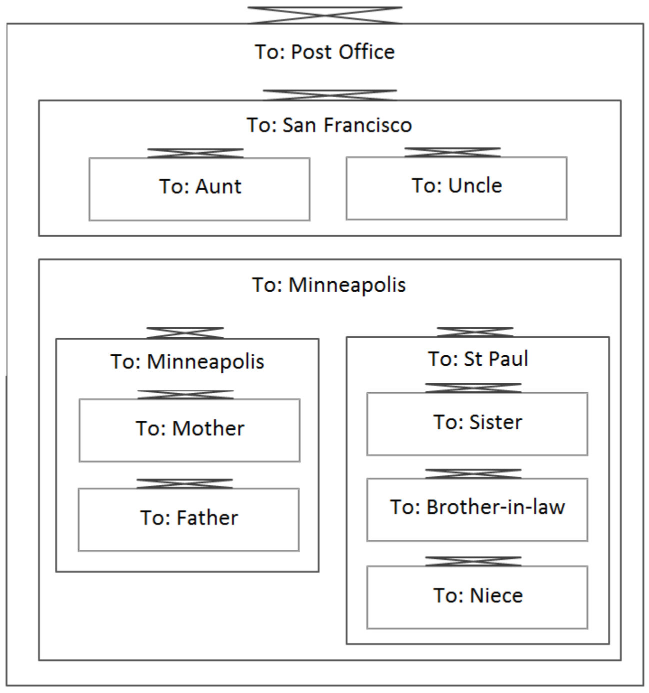
You might buy your aunt and uncle separate gifts, but you would almost certainly ship them together in one package:



Likewise, since your parents, sister and her family live so close to each other in Minnesota, you can ship all of their gifts in one package as well. For convenience, you might also group the gifts destined for your sister and her family in a separate package:



Just to keep the analogy going, you also might combine everything into one big package for transport to the post office:



In this design, there are individually wrapped items, such as a sweater or necktie, and packages that contain individually wrapped items and/or other packages. The beauty of the design is that from the client's perspective (postman, family member, etc.) wrapped items and wrapped packages have the same interface: a "to:" label. Clients can work at a higher level of abstraction. They don't have to distinguish between individually wrapped items and packages. They can treat both uniformly.

This is the essence of the Composite design pattern. Objects are composed hierarchically in a tree data structure. An element (or node) of the hierarchy is either an individual object or a composition of individual objects and/or other compositions. All elements in the hierarchy share the same interface.

In object-oriented terms, clients deal with the abstract interface of a parcel which can be an individually wrapped item or a package of parcels. Since packages conform to the parcel interface, packages may contain other packages recursively.

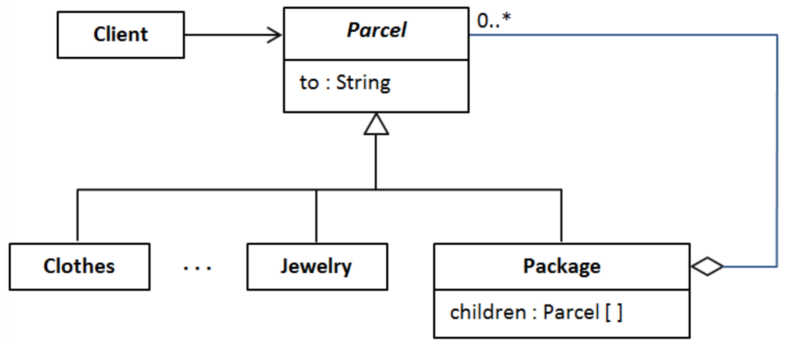
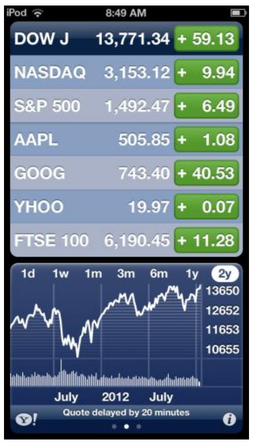


Figure x. A parcel is an individual item or package containing individual items and/or other packages

The Composite pattern is widely used in user interface design to simplify the construction of complex layouts such as the following:



DOW J	13,771.34	+ 59.13
NASDAQ	3,153.12	+ 9.94
S&P 500	1,492.47	+ 6.49
AAPL	505.85	+ 1.08
GOOG	743.40	+ 40.53
YHOO	19.97	+ 0.07
FTSE 100	6,190.45	+ 11.28

If you aren’t familiar with UI design, the task of designing the above user interface might seem more than a little daunting. However, the work can be made more manageable by applying one of the most time-honored concepts in software development: divide and conquer.

With divide and conquer a large problem is split into smaller sub problems. The sub problems are solved and combined into a solution for the problem as a whole. For example, to create the screen above you might work on the top and bottom portions separately. Assuming you could design these individually, it would be trivial to place one above the other in a vertical layout.

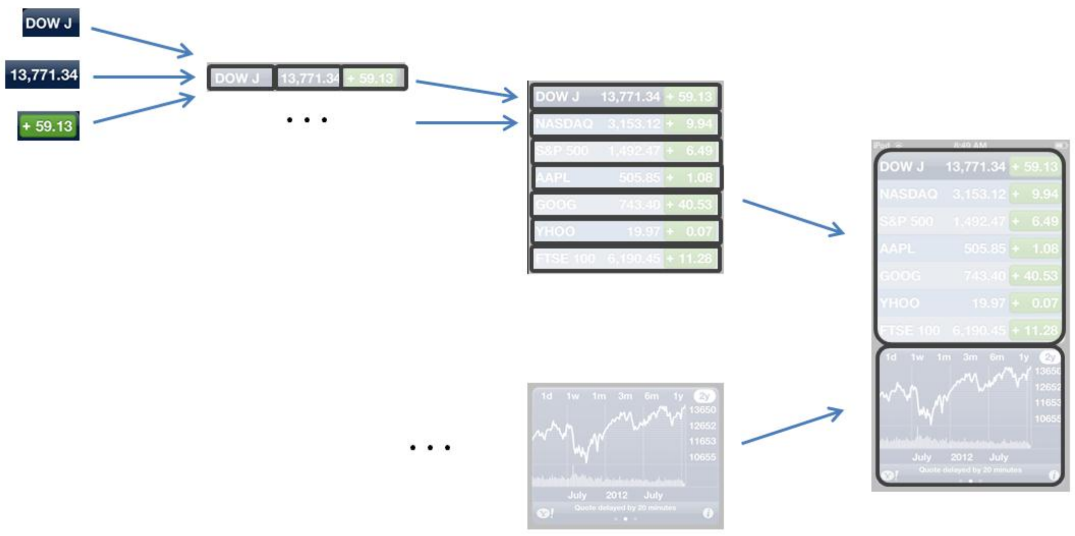


Figure x. Divide and conquer in user interface design

Likewise, the top section of the screen showing the stock quotes would be easy to construct if you already had the elements representing the rows. You would simply stack the rows one after another in a vertical layout. A row is easy to create. It is simply two labels and a button in a horizontal layout. Labels and buttons are primitive UI components so there is nothing more that needs to be done for them.

A user interface of arbitrary complexity can be built hierarchically and recursively by combining primitive UI components into composites and building up ever larger composites of composites. This approach to user interface design is made possible by the Composite design pattern.

Layout managers are designed to work with abstract components (see figure x). Components on the screen are either concrete primitive components such as buttons and labels or containers of components. Because containers implement the abstract interface `Component`, it is possible to have containers within containers ad infinitum.

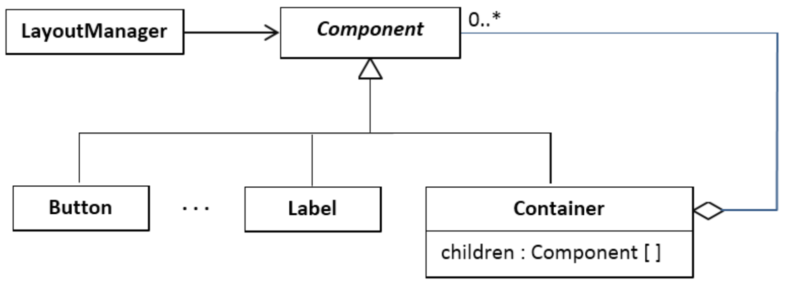


Figure x. Composite pattern in UI design

Intent

The Composite pattern organizes objects into a tree data structure where leaf nodes represent individual objects and interior nodes represent compositions of individual objects and/or other compositions.

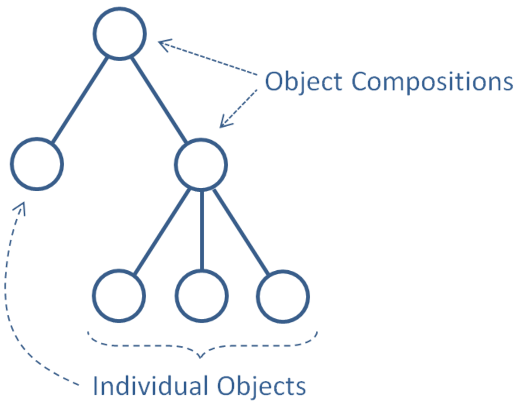
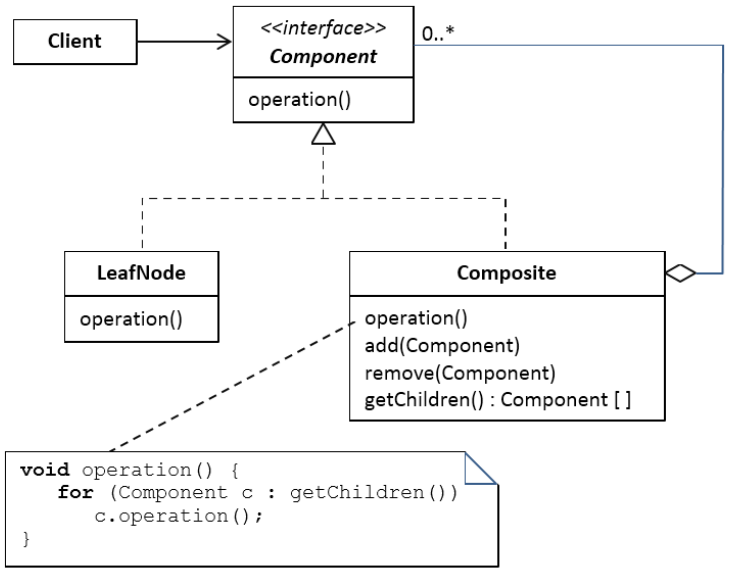


Figure x. The Composite pattern organizes objects into a hierarchical (tree) data structure

What makes this organization a design pattern rather than a standard tree data structure is the fact that objects and compositions share the same interface. The Composite pattern lets clients treat individual objects and compositions of objects uniformly.

Solution

Figure x shows the static structure for the Composite design pattern.



```
void operation() {
   for (Component c : getChildren())
      c.operation();
}
```

Figure x. Static structure for Composite pattern

The four main components of the Composite pattern are:

1. **Component** –`Component` defines the interface common to individual objects and compositions. In the design above, `Component` is defined as an interface. If there is default behavior common to leaf nodes and/or compositions, `Component` can be defined as an abstract class with default implementation.

2. **LeafNode** – Leaf nodes represent primitive elements in the composition. There are one or more leaf node types. For example, when the Composite pattern is used to manage the layout of UI components, leaf nodes are primitive UI components such as `Button`, `TextField`, `Label`, etc.

3. **Composite** – `Composite` defines behavior for a collection of components. `Composite` implements the operations defined in the `Component` interface. The standard implementation for an operation is to forward the request to all child components.

4. **Client** – Clients interact with components in the Composite design pattern through the `Component` interface. Clients are unaware if they are dealing with primitive components or a composition of components.

Sample Code

The sample code in this section shows an implementation for the non-technical example of the Composite pattern introduced at the beginning of the chapter. Clients interact with an abstraction representing a parcel. A parcel is either an individual item such as clothing or jewelry or a package of parcels. Since packages can be treated as parcels, it's possible to have packages within packages.

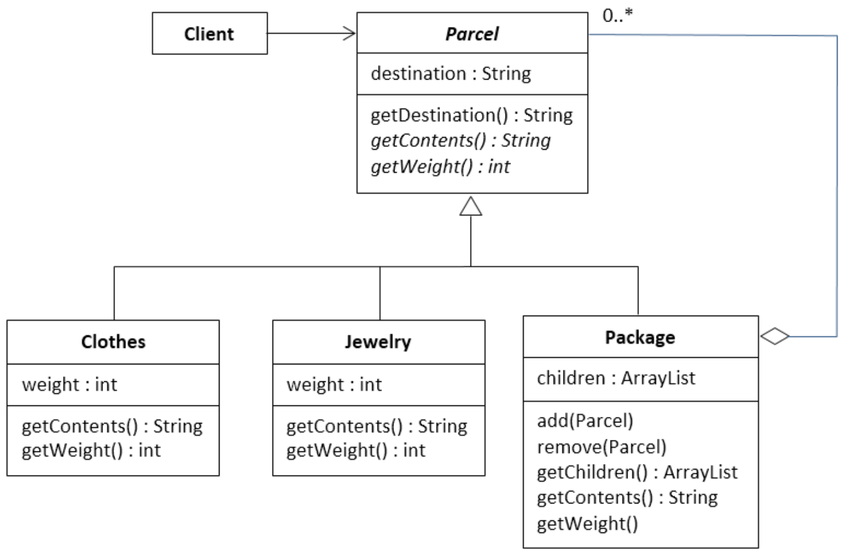


Figure x. Class diagram for sample code

The contents and weight of an individual item are defined in the class representing the individual item. The contents and weight of a package are computed from the children of the package. For example, the `getWeight()` method of `Package` is defined as:

```java
public int getWeight() {
   int weight = 0;
   for(Parcel p : children)
      weight = weight + p.getWeight();
   return weight;
}
```

Here is the complete program:

```java
import java.util.ArrayList;

public class Client {

   public static void main(String[] args) {
      Parcel mother = new Clothes("Mother",6);
      Parcel father = new Clothes("Father",8);
      Parcel aunt = new Jewelry("Aunt",4);
      Parcel uncle = new Clothes("Uncle",10);
      Parcel sister = new Clothes("Sister",6);
      Parcel brother_in_law = new Clothes("Bother-in-law",10);
      Parcel niece = new Jewelry("Niece",2);

      Package sanFrancisco = new Package("San Francisco");
      sanFrancisco.add(aunt);
      sanFrancisco.add(uncle);

      Package parents = new Package("Parents");
      parents.add(mother);
      parents.add(father);

      Package sisterAndFamily = new Package("Sister and family");
      sisterAndFamily.add(sister);
      sisterAndFamily.add(brother_in_law);
      sisterAndFamily.add(niece);

      Package minneapolis = new Package("Minneapolis");
      minneapolis.add(parents);
      minneapolis.add(sisterAndFamily);

      Package postOfficePackage = new Package("Post Office");
      postOfficePackage.add(sanFrancisco);
      postOfficePackage.add(minneapolis);

      process(aunt);
      System.out.println();
      process(postOfficePackage);
   }

   private static void process(Parcel parcel) {
      if (parcel.getWeight() > 150) {
         System.out.println("Weight limit exceeded.");
         System.out.println("Choose another shipping option.");
```

```
            return;
         }
         System.out.println("To: " + parcel.getDestintion());
         System.out.println("Contents: " + parcel.getContents());
         System.out.println("Weight: " + parcel.getWeight() + " kilograms");

      }
   }

   abstract class Parcel {
      private String destination;

      public Parcel(String destination) {
         this.destination = destination;
      }

      public String getDestintion() {
         return destination;
      }

      public abstract String getContents();

      public abstract int getWeight();
   }

   class Clothes extends Parcel {
      private int weight; // in kilograms

      public Clothes(String destination, int weight) {
         super(destination);
         this.weight = weight;
      }

      @Override
      public String getContents() {
         return getClass().getName();
      }

      @Override
      public int getWeight() {
         return weight;
      }
   }

   class Jewelry extends Parcel {
      private int weight;

      public Jewelry(String destination, int weight) {
         super(destination);
         this.weight = weight;
      }

      @Override
      public String getContents() {
         return getClass().getName();
      }
      @Override
      public int getWeight() {
         return weight;
      }
   }

   class Package extends Parcel {
      private ArrayList<Parcel> children = new ArrayList<Parcel>();

      public Package(String destination) {
         super(destination);
      }

      public void add(Parcel p) {
         children.add(p);
      }

      @Override
      public String getContents() {
         StringBuilder contents = new StringBuilder("[");
         // Place comma between elements:
         // item, item, item
         String separator = "";
         for (Parcel p : children) {
            contents.append(separator);
            contents.append(p.getContents());
            separator = ", ";
         }
         contents.append("]");
         return contents.toString();
      }

      @Override
      public int getWeight() {
         int weight = 0;
         for(Parcel p : children)
            weight = weight + p.getWeight();
         return weight;
      }
   }
```

Output:

```
To: Aunt
Contents: Jewelry
Weight: 4 kilograms

To: Post Office
Contents: [[Jewelry, Clothes], [[Clothes, Clothes], [Clothes, Clothes,
Jewelry]]]
Weight: 46 kilograms
```

TBD: Consider showing refactored class diagram with common attributes and behavior of Clothes and Jewelry extracted into a superclass.

Example #2

TBD: Consider adding a real-world inspired example that includes mutable operations on component. One option is to show how the Command and Composite pattern can be combined to allow a sequence of commands to be treated as a single command.

Example #3

TBD: Consider adding an example that requires parent references. Error/exception propagation?

Discussion

TBD: Consider discussing pros and cons of declaring child management operations in Component (root of class hierarchy) vs. Container (composite class). The tradeoffs are transparency vs. safety.

Related Patterns

Chain of Responsibility. Iterator.