Design Patterns

The Composite Design Pattern

The Composite Design Pattern

Treat Individual and Composite Objects the Same Way

The Composite Design Pattern: Compose objects into tree structures that represent whole-part hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly. A leaf has the same interface as a node.

Example: Directories contain entries, each of which could be a directory.

The Composite design pattern is a partitioning design pattern. The Composite design pattern allows a *group* of objects to be treated the same way as a *single instance* of an object. The intent of composite is to compose objects into structures that represent part-whole hierarchies. Composite lets objects treat individual objects and compositions uniformly. Java developers need the Composite pattern because they often must manipulate composites exactly the same way they manipulate primitive objects.

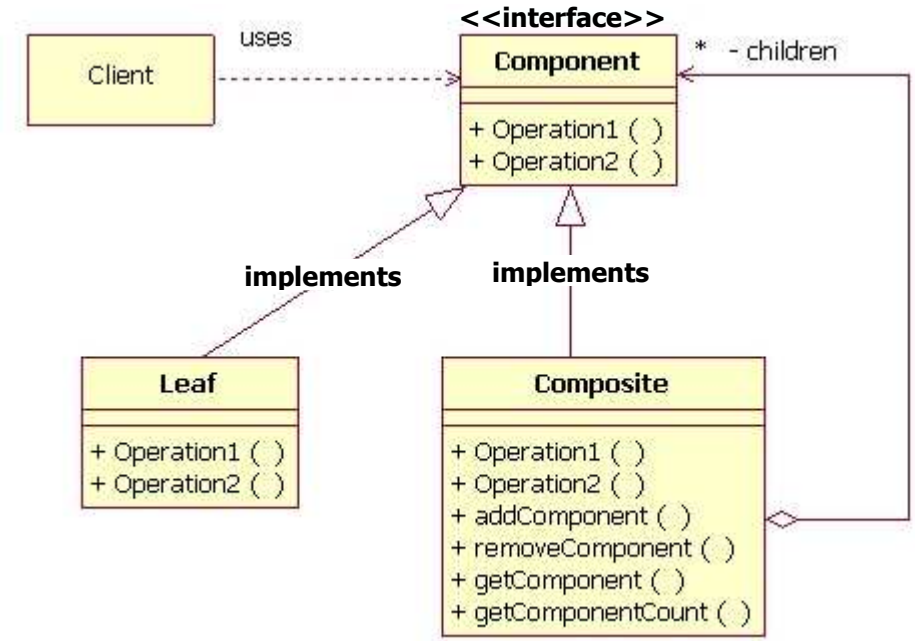
Compositions/Composite Objects: objects that contain other objects; for example, a drawing may be composed of graphic primitives, such as lines, circles, rectangles, text, and so on.

For example, graphic primitives such as lines or text must be drawn, moved, and resized. But we also want to perform the same operation on composites, such as drawings, that are composed of those primitives.

Ideally, we'd like to perform operations on both primitive objects and composites in exactly the same manner, without distinguishing between the two. If we must distinguish between primitive objects and composites to perform the same operations on those two types of objects, our code would become more complex and more difficult to implement, maintain, and extend.

The Composite Design Pattern

Implementing the Composite pattern is easy. Composite classes implement an interface that represents primitive objects. This figure shows a class diagram that illustrates the Composite pattern's structure:



The Composite Design Pattern

In the previous class diagram, *Component* represents an interface for primitive objects, and *Composite* represents a composite class.

For example, the Component class might represent an interface for graphic primitives, whereas the Composite class might represent a Drawing class. The Leaf class from the previous diagram represents a concrete primitive object; for example, a Line class or a Text class. The Operation1() and Operation2() methods represent domain-specific methods implemented by both the Component and Composite classes.

The Composite class maintains a *collection* of components. Typically, Composite class methods are implemented by iterating over that collection and invoking the appropriate method for each Component in the collection.
For example, a Drawing class might implement its draw() method like this:

```
// This method is a Composite method
public void draw()
{
   // Iterate over the components
   for(int i=0; i < getComponentCount(); ++i)
   {
      // Obtain a reference to the component and invoke its draw method
      Component component = getComponent(i);
      component.draw();
   }
}
```

The Composite Design Pattern

For every method in the Component interface, the Composite class implements a method with the same signature that iterates over the composite's components, as illustrated by the draw() method listed previously.

The Composite class implements the Component interface , so you can pass a Composite to a method that expects a Component; for example, consider the following method:

```
// This method is implemented in a class that's unrelated to the
// Component and Composite classes
public void repaint(Component component)
{
   // The component can be a composite, but since it extends
   // the Component class, this method need not
   //  distinguish between components and composites

   component.draw();
}
```

The preceding method is passed a Component—either a simple Component or a Composite—then it invokes that Component's draw() method. Because the Composite class extends Component, the repaint() method need not distinguish between Components and Composites—it simply invokes the draw() method for the Component (or Composite).

The Composite Design Pattern

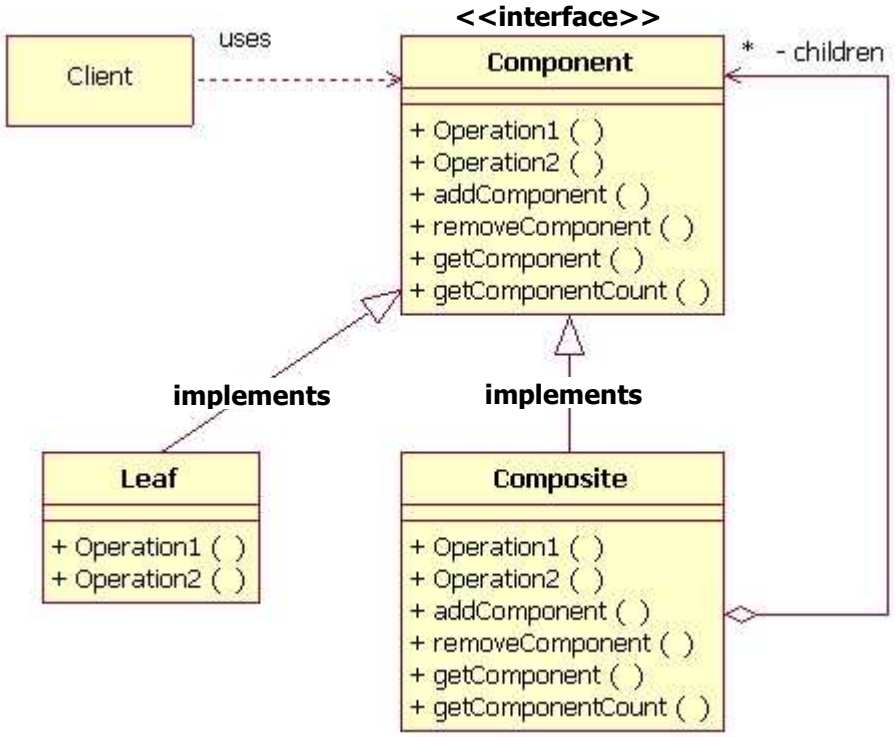
The Composite pattern class diagram does illustrate one problem with the pattern: you must distinguish between Components and Composites when you reference a Component, and you must invoke a Composite-specific method, such as addComponent(). You typically fulfill that requirement by adding a method, such as isComposite(), to the Component class. That method returns false for Components and true for Composites. Additionally, you must also cast the Component reference to a Composite instance, like this:

```
...
if(component.isComposite())
{
   Composite composite = (Composite)component;
   composite.addComponent(someComponentThatCouldBeAComposite);
}
...
```

Notice that the addComponent() method is passed a Component reference, which can be either a primitive Component or a Composite.

The Composite Design Pattern

The following figure shows an alternative Composite pattern implementation:



If you implement Composite pattern in this way, you don't have to distinguish between Components and Composites, and you don't have to cast a Component reference to a Composite instance. The code listed earlier reduces to a single line:

```
...
component.addComponent(someComponentThatCouldBeAComposite);
...
```

The Composite Design Pattern

But, if the Component reference in the preceding code fragment does not refer to a Composite, what should the addComponent() do?

That's a major point of contention with this Composite pattern implementation. Because primitive Components do not contain other Components, adding a Component to another Component makes no sense, so the Component.addComponent() method can either fail silently or throw an exception.

Typically, adding a Component to another primitive Component is considered an error, so throwing an exception is perhaps the best course of action.

So which Composite pattern implementation—the one in the first class diagram or the one in the second—works best? That's always a topic of great debate among Composite pattern implementers.

The technique shown in the first implementation avoids implementing methods in a class that don't make sense for that object type.

In the second implementation, you never need to distinguish between Components and Containers, and you never need to perform a cast.

The Composite Design Pattern

Lets create an example by considering a Cargo Ship.

Assume that cargo ships are loaded with both individual “packages” (any item packed and shipped individually) and “shipping containers” that contain multiple “packages” (and also possibly other “shipping containers”).

- Larger “packages” (machinery parts, transformers, vehicles, etc.) are not loaded into shipping containers - they are loaded individually into the Ship.
- Smaller “packages” are generally loaded into the “shipping containers” for ease of transport.
- Small “shipping containers” are sometimes packed into larger “shipping containers”.

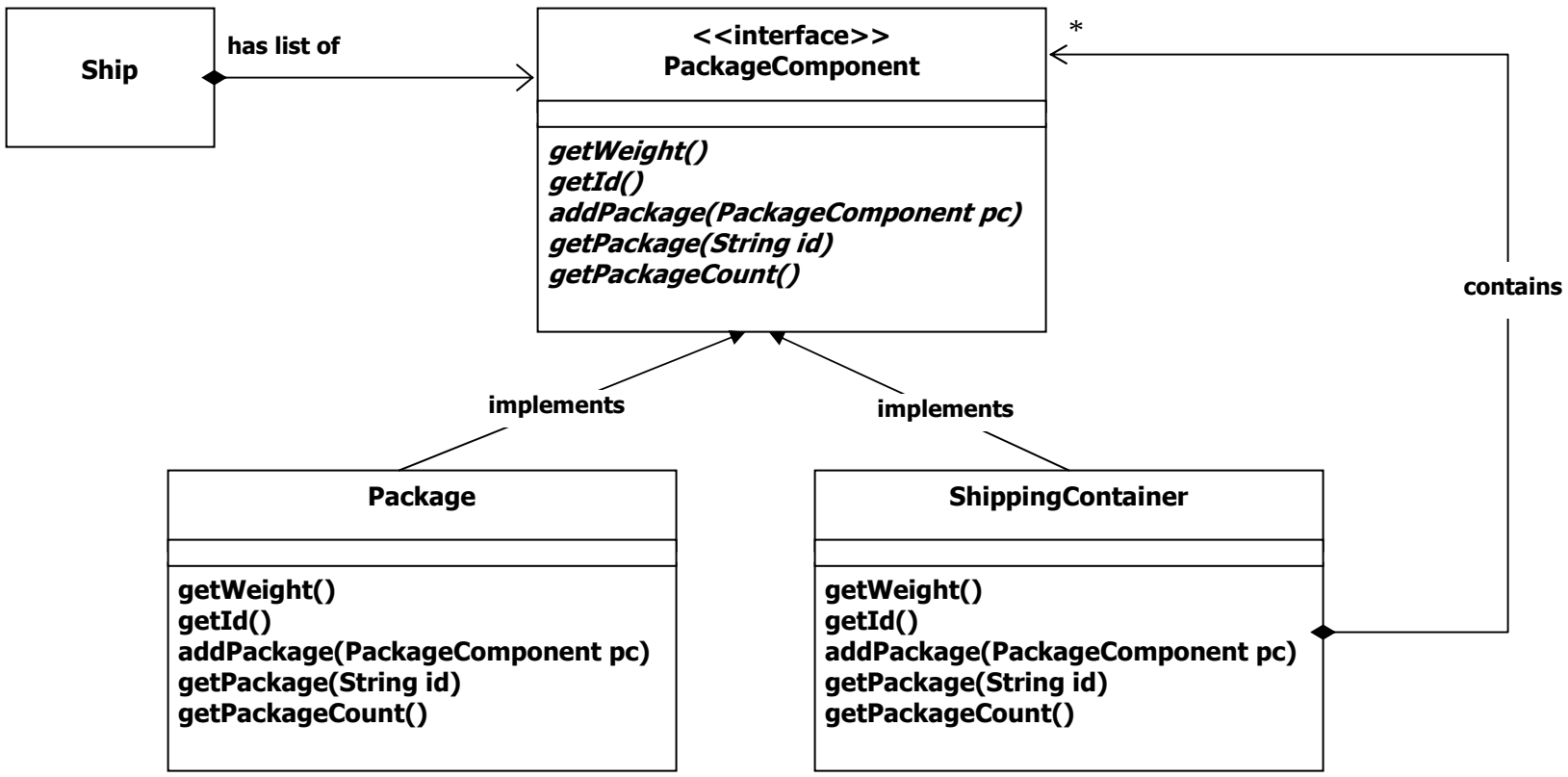
These “shipping containers” are then loaded onto the ship.

Assume the “cargo ship” needs to be able to report the total count and total weight of all loaded packages at any given moment. Also assume that we do not want to design our ship such that it needs to know the details of and differences between “packages” and “shipping containers”.

We can use the Composite Design Pattern to help us solve this design problem.

The Composite Design Pattern

The following UML class diagram represents a solution to this problem using the Composite Design Pattern:



The Composite Design Pattern

The following is an implementation example of the “Ship – Shipping Container” – Package problem based upon the previous UML diagram:

PackageComponent interface

```
public interface PackageComponent
{
    double getWeight();

    String getId();

    void addPackage(PackageComponent pc)
                                throws Exception;

    PackageComponent getPackage(String id);

    int getPackageCount();
}
```

The Composite Design Pattern

Continued: an implementation example of the "Ship – Shipping Container" – Package problem based upon the previous UML diagram:

Package Class

```
public class Package implements PackageComponent
{
    private String identifier = null;
    private double weight;

    public Package(String id, double w)
    {
        identifier = id;
        weight = w;
    }

    public double getWeight()
    {
        return weight;
    }

    public String getId()
    {
        return identifier;
    }

    public void addPackage(PackageComponent pc)
                                        throws Exception
    {
      throw new Exception(
         "A Package cannot have Packages added to it!");
    }

    public PackageComponent getPackage(String id)
    {
        return null;
    }

    public int getPackageCount()
    {
        return 1;
    }

}
```

The Composite Design Pattern

Continued: an implementation example of the "Ship – Shipping Container" – Package problem based upon the previous UML diagram:

ShippingContainer Class

```
public class ShippingContainer
                    implements PackageComponent
{

    private String identifier = null;
    public HashMap<String, PackageComponent> items
                  = new HashMap<String, PackageComponent>();


    public ShippingContainer(String id)
    {
        identifier = id;
    }


    public double getWeight()
    {
        double wgt = 0;
        for (PackageComponent pc : items.values()){
            wgt += pc.getWeight();
        }
        return wgt;
    }


    public String getId()
    {
        return identifier;
    }


    public void addPackage(PackageComponent p)
    {
        items.put(p.getId(), p);
    }

    public PackageComponent getPackage(String id)
    {
        if (items.containsKey(id))
        {
            return items.get(id);
        }

        PackageComponent p = null;
        for (PackageComponent pc : items.values())
        {
            p = pc.getPackage(id);
            if (p != null) return p;
        }

        return null;
    }

    public int getPackageCount()
    {
        int cnt = 0;
        for (PackageComponent pc : items.values())
        {
            cnt += pc.getPackageCount();
        }
        return cnt;
    }
}
```

The Composite Design Pattern

Continued: an implementation example of the "Ship – Shipping Container" – Package problem based upon the previous UML diagram:

Ship Class

```
public class Ship
{
    public HashMap<String, PackageComponent> cargo
               = new HashMap<String, PackageComponent>();

    public double getWeight()
    {
        double wgt = 0;
        for (PackageComponent pc : cargo.values())
        {
            wgt += pc.getWeight();
        }
        return wgt;
    }

    public void addPackage(PackageComponent p)
    {
        cargo.put(p.getId(), p);
    }

    public PackageComponent getPackage(String id)
    {
        if (cargo.containsKey(id))
        {
            return cargo.get(id);
        }

        PackageComponent p = null;
        for (PackageComponent pc : cargo.values())
        {
            p = pc.getPackage(id);
            if (p != null) return p;
        }

        return null;
    }

    public int getPackageCount()
    {
        int cnt = 0;
        for (PackageComponent pc : cargo.values())
        {
            cnt += pc.getPackageCount();
        }
        return cnt;
    }

}
```

The Composite Design Pattern

Continued: an implementation example of the "Ship – Shipping Container" – Package problem based upon the previous UML diagram:

A Sample "main"

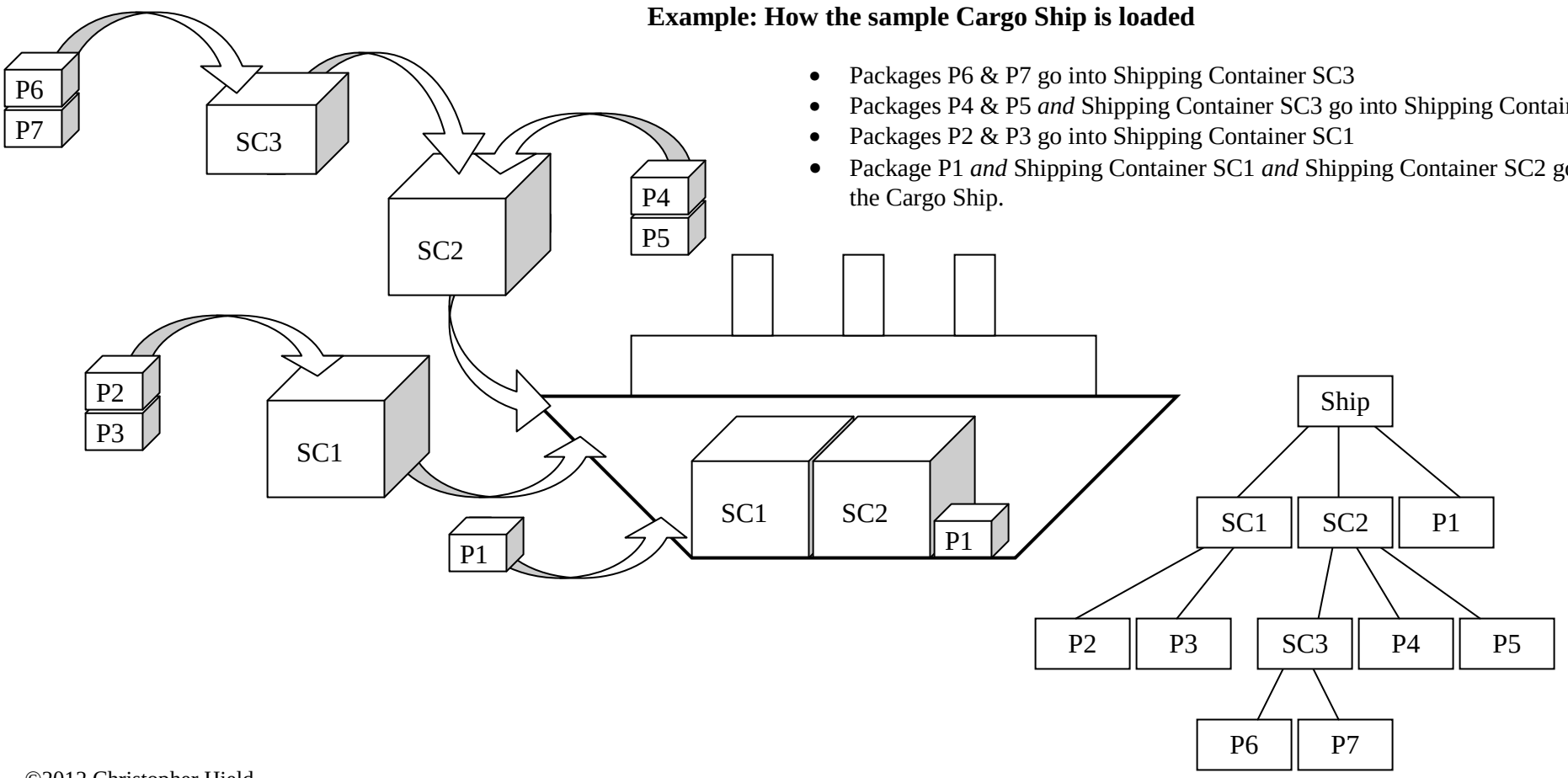
```
// This sample "main" builds a Cargo Ship filled with packages
// & shipping containers, and then with a single call can
// determine the weight & package cound, without worrying
// about which of its contents are which.

[…]
    public static void main(String args[])
    {
        // Build a Ship
        Ship theShip = new Ship();

        // Create some (empty) "shipping containers"
        PackageComponent sc1 = new ShippingContainer("SC1");
        PackageComponent sc2 = new ShippingContainer("SC2");
        PackageComponent sc3 = new ShippingContainer("SC3");

        // Create some individual "package" objects
        PackageComponent p1 = new Package("P1", 101.4);
        PackageComponent p2 = new Package("P2", 122.8);
        PackageComponent p3 = new Package("P3", 143.2);
        PackageComponent p4 = new Package("P4", 164.9);
        PackageComponent p5 = new Package("P5", 185.5);
        PackageComponent p6 = new Package("P6", 206.1);
        PackageComponent p7 = new Package("P7", 227.6);
```

A Sample "main" (cont.)

```
try
{
    // Fill ShippingContainer 3 with some packages
    sc3.addPackage(p6);
    sc3.addPackage(p7);

    // Put ShippingContainer 3 INTO ShippingContainer
    // 2, and add more packages to ShippingContainer 2
    sc2.addPackage(sc3);
    sc2.addPackage(p5);
    sc2.addPackage(p4);

    // Now put some packages into ShippingContainer 1
    // - but no ShippingContainers

    sc1.addPackage(p3);
    sc1.addPackage(p2);

    // Now put the 2 top-level ShippingContainers (1 &
    // 2) AND one individual package on the Ship
    theShip.addPackage(sc1);
    theShip.addPackage(sc2);
    theShip.addPackage(p1);
}
catch (Exception e)
{
    e.printStackTrace();
}
```

The Composite Design Pattern

Continued: an implementation example of the "Ship – Shipping Container" – Package problem based upon the previous UML diagram:



The Composite Design Pattern

Continued: an implementation example of the "Ship – Shipping Container" – Package problem based upon the previous UML diagram:

A Sample "main" (cont.)

```
        // Next - get the individual package count
        System.out.println("Num. Packages: " +
                                theShip.getPackageCount());

        // Now get the total weight of the Ship's contents
        System.out.println("Total Package Weight: " +
                                    theShip.getWeight());

    } // End main
```

When we call the Cargo Ship's "getPackageCount()" method, the ship is not concerned with exactly what is in it's cargo area (Containers or Packages) – all elements must implement the PackageComponent interface, therefore all can respond to the "getPackageCount()" request..

The Composite pattern put in place in this example assures us that regardless of the type of item in the cargo area (Containers or Packages), the ship can just propagate the "getPackageCount()" to each top-level item and simply return the sum of their responses:

- Cargo Ship package count is **7**
  - Package P1 returns it's package count = **1**
  - Shipping Container SC1 reports the sum of the package count of P2 & P3 = 1 + 1 = **2**
  - Shipping Container SC2 reports the sum of the package count of P4, P5, & SC3 = 1 + 1 + *2* = **4**
    - *Shipping Container SC3 reports the sum of the package count of P6 & P7 = 1 + 1 = 2*

The Composite Design Pattern

Continued: an implementation example of the "Ship – Shipping Container" – Package problem based upon the previous UML diagram:

A Sample "main" (cont.)

```
        // Next - get the individual package count
        System.out.println("Num. Packages: " +
                                theShip.getPackageCount());

        // Now get the total weight of the Ship's contents
        System.out.println("Total Package Weight: " +
                                        theShip.getWeight());

    } // End main
```

Likewise, when we call the Cargo Ship's "getWeight()" method, the ship is not concerned with exactly what is in it's cargo area (Containers or Packages) – all elements must implement the PackageComponent interface, therefore all can respond to the "getWeight ()" request..

The Composite pattern put in place in this example assures us that regardless of the type of item in the cargo area (Containers or Packages), the ship can just propagate the "getWeight ()" to each top-level item and simply return the sum of their responses:

- Cargo Ship weight is **1151.5**
  - Package P1 returns it's package count = **101.4**
  - Shipping Container SC1 reports the sum of the weight of P2 & P3 = 122.8 + 143.2 = **266.0**
  - Shipping Container SC2 reports the sum of the weight of P4, P5, & SC3 = 164.9 + 185.5 + *433.7* = **784.1**
    - *Shipping Container SC3 reports the sum of the weight of P6 & P7 = 206.1 + 227.6 = 433.7*

The Composite Design Pattern

Summary

When dealing with tree-structured data, programmers often have to discriminate between a leaf-node and a branch. This makes code more complex, and therefore, error prone. The solution is an interface that allows treating complex and primitive objects uniformly.

In object-oriented programming, a composite is an object (e.g., a shape) designed as a composition of one-or-more similar objects (other kinds of shapes/geometries), all exhibiting similar functionality. This is known as a "has-a" relationship between objects. The key concept is that you can manipulate a single instance of the object just as you would a group of them. The operations you can perform on all the composite objects often have a least common denominator relationship. For example, if defining a system to portray grouped shapes on a screen, it would be useful to define resizing a group of shapes to have the same effect (in some sense) as resizing a single shape.

Composite can be used when clients should ignore the difference between compositions of objects and individual objects. If programmers find that they are using multiple objects in the same way, and often have nearly identical code to handle each of them, then composite is a good choice; it is less complex in this situation to treat primitives and composites as homogeneous.