Design Patterns

The Flyweight Design Pattern

Flyweight
Design Pattern

The Problem

Online auction web sites facilitate the buying and selling of a large number of a wide variety of goods. Millions of collectibles, decor, appliances, computers, furnishings, equipment, domain names, vehicles, and other miscellaneous items are listed, bought, or sold daily.

Each item has multiple price values associated with it: the item's initial price, reserve price (if applicable), and the current auction price. If the applications that make up the auction system represent an individual price as a "Price" object (why you would represent a price this way is briefly covered in Appendix A of this document, "Precision Issues Representing Currency Using float or double Types"), then many millions of Price objects would be needed to represent all those values.

In addition to the Price objects needed to represent the items offered up for auction, the current bid prices of all users for all the items they are bidding on would also need to be represented in a similar fashion. This adds up to an extremely large number of Price objects in this system, which in total could take up many megabytes (or even gigabytes) of memory space. Just representing these price values would be a large drain on the resources of a system – not to mention the resources needed to represent the system as a whole.

An analysis of the price values held in those millions of price objects would indicate that there are not *actually* millions of unique individual price values represented. A large number of those price objects would be found to hold the same value.

Many of the items up for auction would coincidently have the same auction price as other items up for auction. Many users may have bids on items that coincidently have the same price as other user's bids on other items. Imagine how many auctions might have a current bid price of $100.00. Of all the millions of price values in use, many would contain duplicated price values.

Assume that analysis showed that of the millions of price objects, there were only about 10,000 individual *unique* price values. The system therefore is using *millions* of price objects to represent 10,000 unique values. On average then, each unique price value is duplicated hundreds of times throughout the auction system. This is certainly wasteful. One price object holding the value $100.00 is semantically identical to all the other price objects holding the value $100.00. The system, as we have described it, is using many hundreds of times the resources it needs to in order to represent price values.

When you need to create a large number of objects, each requires some amount of memory to store the object's state (data values). Even if the storage requirements for each individual object are small, the sheer number of objects may cause the overall memory usage to be high. Depending upon the scenario and the target environment, the memory usage may be so high that the program will perform poorly, or cannot execute at all.

The Solution - Flyweight

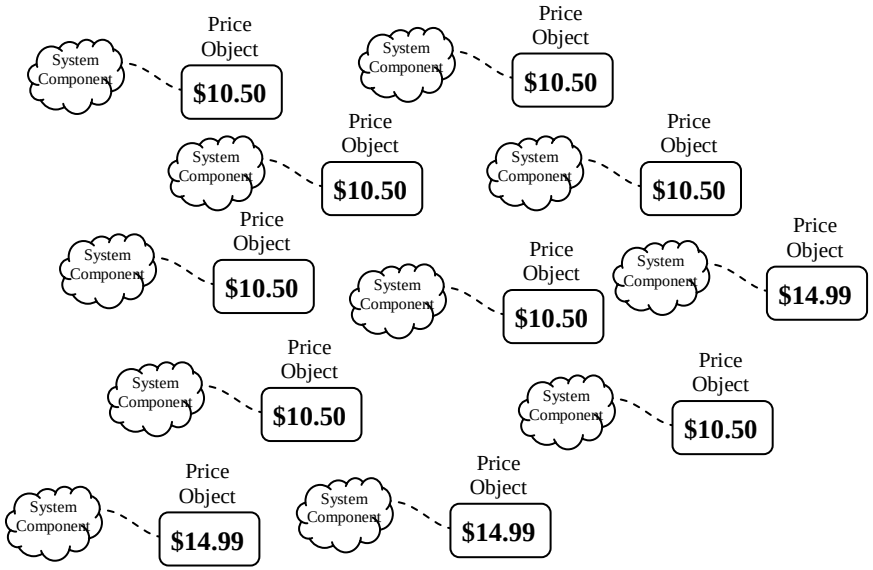
The Flyweight design pattern is used to reduce the memory and resource usage for complex applications containing many hundreds, thousands or millions of duplicate objects.

To do this, the flyweight pattern specifies a solution wherein only *one* object holding a specific value (or specific set of values) is ever created - rather than 2, or 10, or 10,000 objects each holding the same value. *Note that this is different from what is defined in the Singleton pattern. The Singleton pattern specifies a technique wherein only one object of a given class can ever be created. Here, we will have multiple objects of the same class type, but we will have only one object holding a given set of data (i.e., one state).*

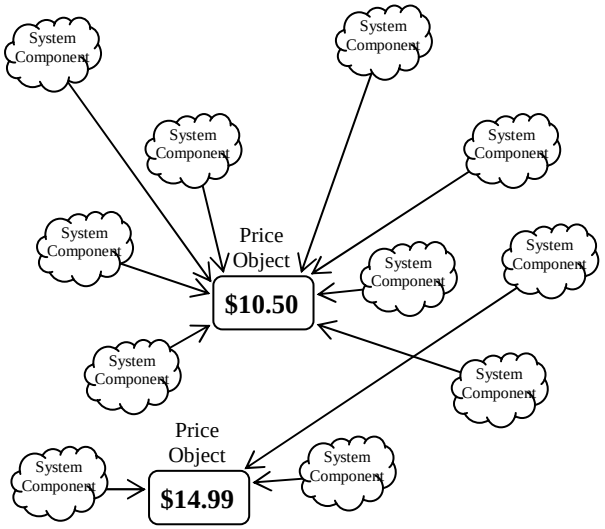
Flyweight
Design Pattern

We can take all of the objects that have the same state and replace them with a single *shared* object, thus reducing the number of objects down to the number of unique states you have.

Non-Flyweight representation of prices: Many individual Price objects owned by a variety of system components. Duplicate price values result in duplicate objects



Flyweight representation of prices: One Price object per price value. A variety of system components refer to the one Price object representing a desired value. Duplicate price values result in duplicate objects



Flyweight Technique

Instead of invoking a normal constructor, the Flyweight pattern requires that a "Factory" is used to create these Flyweight (shared) objects. That way, you can track the objects that have already been created, and only create a new object if the requested state is different from the objects you already have.

Note that Flyweights and their Factory are often located separately in their own java package. In this way, the Flyweight's constructor can be declared as package-visible: Available to the Factory, not available to code outside the package. This allows the Factory to invoke the Flyweight's constructor, but hides the Flyweight's constructor from the rest of the code-base.

When some part of the application requests the creation of one of these objects via the Factory, the Factory checks to see if an object containing the same values as the current request has already been created. If so, a reference to that previously created object is returned (no new object is created). If no object containing the same values as the current request has been created, a new object is created using those values and is returned to the requester. A reference to that new object is saved so that if another request to create an object with those same values is made, a reference to that newly created object will be returned.

It is important that Flyweight objects should be immutable *(unchangeable once created).* Since a single flyweight object is referred to by many system components, it must be sure to maintain the value it was created with. The methods of a flyweight should *never* modify the state of the object, but instead return a *new* object whose state reflects the operation performed.

Immutable objects are objects whose state (the object's data) cannot change after construction. Examples of immutable objects from the Java JDK include String and Integer. This is in contrast to a mutable object, which can be modified after it is created

Flyweight
Design Pattern

For example, assume the “add” method of a Price flyweight would add the current price object’s value to a price value passed into the “add” method as the parameter. Since the Price flyweight is immutable, this operation should *not* modify either the current price object’s value or the price object passed in as the parameter. Instead, the sum of the 2 Price flyweights should be returned as a separate object:

```
System.out.println(aPrice); // Displays the value “$10.00”
System.out.println(anotherPrice); // Displays the value “$1.99”
System.out.println(sumPrice); // Displays “null”

sumPrice = aPrice.add(anotherPrice); // Adds the prices – returns the sum

System.out.println(aPrice); // Still displays the value “$10.00”
System.out.println(anotherPrice); // Still displays the value “$1.99”
System.out.println(sumPrice); // Displays the value “$11.99”
```

When the above code executes, the values of “aPrice” and “anotherPrice” would remain unchanged. A different Price reference would be returned as the calculated sum, to be held in “sumPrice”.

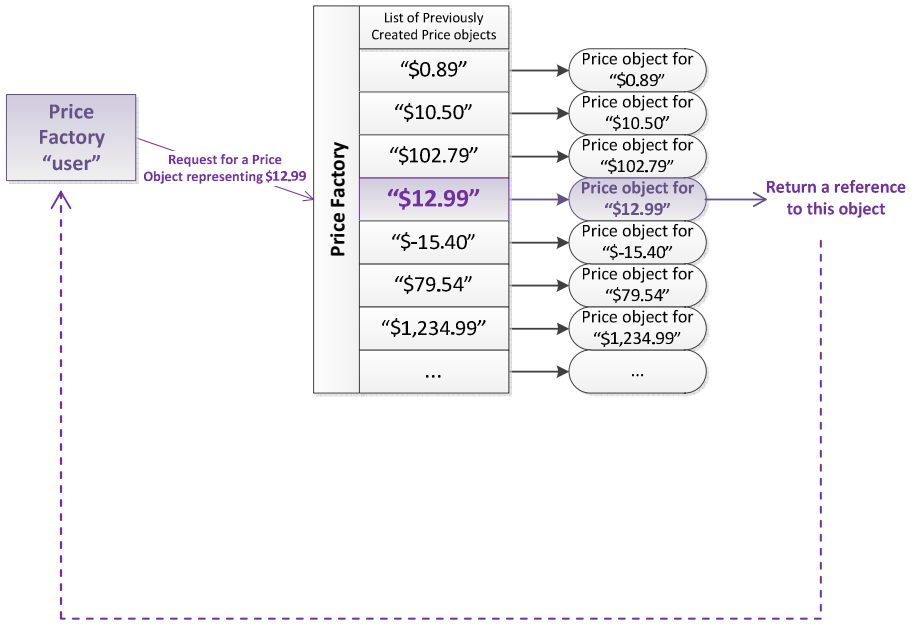
In our auction example, a “Price Factory” would be created to support the Flyweight Design Pattern that would maintain a list of all Price objects it has previously created. When a request is made to the Price Factory to create a Price object for a certain value, the Price Factory will *first* check to see if it has previously created a Price object with that value. If it *has* previously created a Price object with that value, it simply returns a reference to that previously created Price object. If it *has not* previously created a Price object with that value, it creates a *new* price object representing the requested value, puts that object in its list of previously created prices, and then returns the requested Price object to the caller.

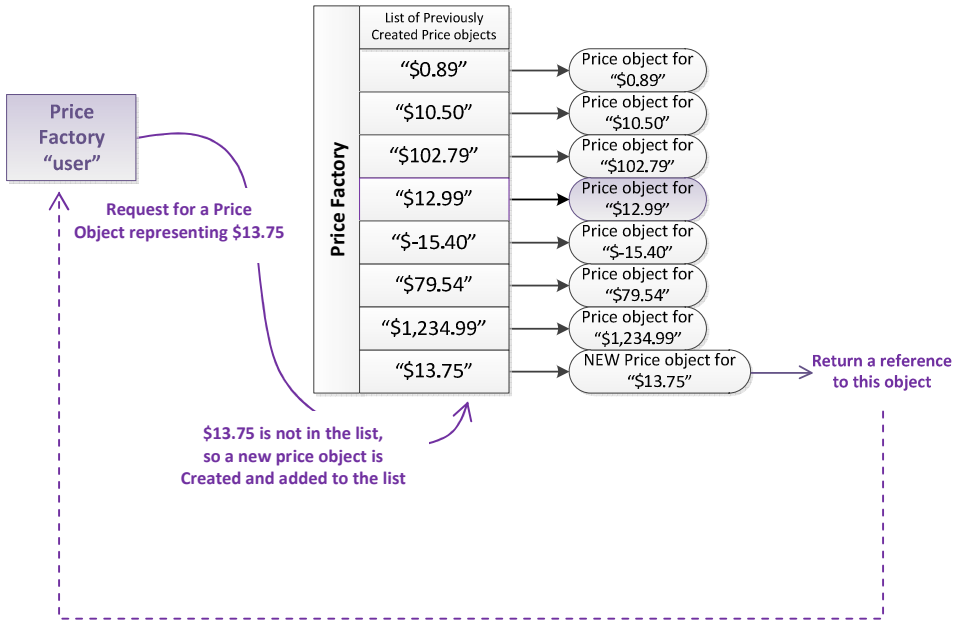
Flyweight
Design Pattern

Example

- If the Price Factory (shown in the diagram below) was asked to create a Price object for the value "$12.99", it would first check to see if it has previously created a Price object with that value. Since it *has* ("$12.99" is in the list of previously created Price objects), it would simply return a reference to that previously created Price object.

- If the PriceFactory was asked to create a Price object for the value "$13.75", it would first check to see if it has previously created a Price object with that value. Since it has not ("$13.75" is not in the list of previously created Price objects), it would create a new price object representing "$13.75", put that Price object in the list of previously created prices, and then return the Price object to the caller.





Lazy vs. Eager Initialization

In the previous Price Factory example, the Factory supporting our Price flyweights was creating the Price flyweights only when they were needed. This technique is called “Lazy Initialization”. Only objects that are specifically *needed* are created, and only *when* they are needed. Lazy initialization is often used together with the Factory design pattern. Lazy Initialization has several objectives:

- Limit object creation to only those that are needed
- Delay an object creation until it's absolutely necessary
- Store the object for future use, so you won't need to create it again

Lazy Initialization is the most common form of initialization used in conjunction with the Flyweight pattern. This technique works best if you do not know the exact values (states) of the objects that will need to be created in advance.

However, what if price-use-analysis showed that every day, without fail, you always used *every* price value between $0.01 and $10.00. Other values are also used, but not as regularly as these are. You might use $100.48 one day, but it might not be requested again for several days. Since you know in advance that used *every* price value between $0.01 and $10.00 would be needed, you *could* try an alternate initialization technique in your Price Factory called “Eager Initialization”.

Using Eager Initialization, the Price Factory would pre-create Price objects for all price values between $0.01 and $10.00. This way, when a request is made for one of these values, no object creation is needed at that time, the factory simply returns a reference to the appropriate pre-created Price object. If other non-regular values were requested, the Price Factory would perform as it would in the “Lazy Initialization” example – create the new price object, add it to the list of previously created prices, then return the Price object to the requesrer.

Examples of Flyweight Usage

In Java, String objects are managed as flyweights. Java puts all *fixed String literals* into a literal pool. For redundant literals, Java keeps only *one* copy in the pool. This usage of the flyweight can be demonstrated by the following code example:

```
String string1 = "This is a literal Java String";
String string2 = "This is a literal Java String";

if (string1.equals(string2)) {
    System.out.println("These 2 variables contain the same String");
} else {
    System.out.println("These 2 variables do not contain the same String");
}


if (string1 == string2) {
    System.out.println("These 2 variables ARE the same String");
} else {
    System.out.println("These 2 variables ARE NOT the same String");
}
```

When executes, this code generates the following:

```
These 2 variables contain the same String
These 2 variables ARE the same String
```

- The two String objects are *semantically* equal because their contents are the same. Content (semantic) equality is demonstrated by calling .equals() in the first "if" statement – which returns "true".
- The two objects are also located in the same location in memory (Flyweight in action!). The second "if" statement compares the references (addresses) of "string1" and "string2" which are shown to be equal, even though they were independently created.

Even though these two String objects are created separately, Java is actually storing the String once, and referencing it in multiple places. The Flyweight payback here can be significant: profiling reveals that String objects often represent anywhere from a third to a half of all objects created in a typical application. As the test demonstrates, the comparison can work—it can return true even if the two objects were created at separate times during execution of the Java application.

Another area that lends itself to usage of the Flyweight pattern is the use of Time objects. Imagine an appointment scheduling application for a medical group. There are many doctors, and many patients, and a long stretch of time in the future in which to make appointments. This application must support very large numbers of appointments, perhaps millions. Just as before, the large number of Time objects would be a significant drain of memory resources. The reality is however, that appointments during a business day typically start either on the hour, or at quarter hour intervals past the hour. If we were to use the Flyweight pattern so that only unique Time instances were created, we could reduce the maximum number of Time objects to 40 (four possible appointment times per hour during the 10 hours the office is open). The appointment application could still support odd times if needed (times off the quarter-hour), but in general use, the flyweight would dramatically minimize object creation and memory usage.

When Should the Flyweight Pattern Be Used?

There are a couple of conditions that should be met before attempting to convert your objects to flyweights. Your application must use a large number of objects of a given class type with the same state. This is the most important condition; it is not worth performing this optimization if you only expect to use a few copies of the object in question.

The other condition is that you must be left with a relatively small number of unique objects. Here again, it is not worth performing this optimization if it results in the reduction of only a small percentage of the objects that will be created.

Benefits of the Flyweight Pattern

The flyweight pattern can reduce your resource load by several orders of magnitude. It does not require huge changes to your code to get these savings. Once you have created the class that is to be your flyweight (i.e., Price, Time, etc.) and the Factory, the only change you must make to your code is to call a method of the Factory instead of instantiating objects directly.

If you are creating the flyweight for other programmers to use as an API, they need only slightly alter the way they call it to get the benefits. This is where the pattern really excels; if you make this optimization to your API once, it will be much more efficient for everyone else who uses it. When using this optimization for a library that is used over an entire site, your users may well notice a huge improvement in memory usage and speed.

Drawbacks of the Flyweight Pattern

This is only an optimization pattern. It does nothing other than improve the efficiency of your code under a strict set of conditions. It cannot and should not be used everywhere; it can actually make your code less efficient if used unnecessarily.

In order to optimize your code, this pattern adds complexity, which makes it harder to debug and maintain. It is harder to debug because there are now two places where an error could occur (the factory and the flyweight) where before there was only a single object to worry about.

These drawbacks are not “deal breakers” - they simply mean that this optimization should only be done when needed. Trade-offs must always be made between run-time efficiency and maintainability, but such trade-offs are the essence of engineering.

Flyweight
Design Pattern

Summary

The Flyweight Design pattern is an optimization pattern, used to improve performance and make your code more efficient, especially in its use of memory. Each unique instance of a Flyweight becomes a resource shared among many locations. A single flyweight object takes the place of many of the original objects.

It is important that Flyweight objects should be immutable. The methods of a flyweight should *never* modify the state of the object, but instead return a *new* object whose state reflects the operation performed.

For the flyweight object to be shared like this, an additional class must be added. A Factory class is needed to control how the Flyweight gets instantiated, and to limit the number of instances created to the absolute minimum. It should also store previously created instances and should to reuse them if a similar object is needed later.

When used improperly, the Flyweight pattern can make your code more complicated, harder to debug, and harder to maintain, with few performance benefits to make up for it. When used properly however, the flyweight pattern can improve performance and reduce needed resources significantly.

Appendix A: Precision Issues Representing Currency Using `float` or `double` Types

While floating-point data types are capable of representing extremely large positive and negative numbers and offer the equivalent of many decimal digits of precision, they are nonetheless inexact when it comes to representing decimal numbers. The reason for this stems from the fact that computers are binary, not decimal, machines. Internally, the hardware must represent a decimal number using a binary format, and most real decimal numbers cannot be exactly represented in a fixed-length binary format.

For example, the decimal value 9.48 cannot be represented exactly as a binary floating-point value; it must be represented as a close approximation – in this case, 9.479999542236328125. Fortunately, Java's built-in float-to-String conversion methods can identify such an approximation, and display the value as 9.48, instead of 9.479999542236328125, as can be seen in the below example:

```
float unitCost = 9.48f;
System.out.println("Value: " + unitCost);
```

Generates `Value: 9.48`

Now, suppose you wish to compute the total cost of 100 items having a unit cost of $9.48:

```
float unitCost = 9.48f;
float totalCost = unitCost * 100.0f;
System.out.println("Total Cost: $" + totalCost);
```

Generates `Total Cost:  $947.99994`

In this case, the effects of inexactness when using a floating-point data type to represent a monetary value are obvious. This example indicates that 9.48 * 100.0 is not 948.0 as you would expect, but rather 947.99994. If that computed value is then truncated to two decimal places, the result is $947.99, not $948.00. It would cost you a penny if you were preparing a customer invoice – every time this happens. The problem only gets worse as the computations become more complex.