Flyweight Design Pattern

Acu Raul Mihai

Intent of the DP

The Flyweight pattern is meant to provide a solution to the high memory usage when we have multiple similar objects, by sharing as many characteristics and features among them. It is a structural pattern and reduces the number of objects created while reducing memory usage.

Explanation

By using the Flyweight pattern, we actually try to reuse objects that already exist by storing them, or, if no matching object is found, creating a new one. A more detailed intent of this design pattern is to support creation of a large number of objects that have a part of their internal state similar, while other state parts can vary. Flyweight pattern saves memory by sharing flyweight objects among clients.

Mutable objects can be represented by Flyweights if the mutable parts can be separated out and made extrinsic. The disadvantage of Flyweight is that moving the state outside the object breaks **encapsulation**, and may be less efficient than keeping the state intrinsic. In these cases it may be necessary to decide whether performance or memory is more important.

http://gameprogrammingpatterns.com/flyweight.html
https://www.tutorialspoint.com/design_pattern/flyweight_pattern.htm
http://java.boot.by/scea5-guide/ch07s03.html

Related patterns

Factory and Singleton

Flyweights are usually created using a factory and the singleton is applied to that factory so that for each type or category of flyweights a single instance is returned.

State and Strategy

State objects can be shared. If State objects have no instance variables ,the state they represent is encoded entirely in their type, so they are essentially flyweights with no intrinsic state, only behavior.

Strategies increase the number of objects in an application. Any residual state is maintained by the context, which passes it in each request to the Strategy object. We can reduce this overhead by implementing strategies as stateless objects that contexts can share

Composite pattern

The Flyweight pattern is often combined with the composite pattern in order to represent a hierarchical structure as a graph with shared leaf nodes. A consequence of sharing is that flyweight leaf nodes cannot store a pointer to their parent. Rather, the parent pointer is passed to the flyweight as part of its extrinsic state. This has a major impact on how the objects in the hierarchy communicate with each other.

Common situations of use

Flyweight pattern comes in handy when our application struggles with the shortage of memory, runs slowly already and creating a huge number of instances and objects might just make it even harder to work with.

It is highly recommended to use flyweight when you have objects that need to be more lightweight, generally because you have too many of them.

For example, in graphics, if we`re trying to generate a forest, with instanced rendering, it’s not so much that they take up too much memory as it is they take too much time to push each separate tree over the bus to the GPU, but the basic idea is the same.

Can be mistaken with

In my opinion, Flyweight might be mistaken with the **Prototype** pattern.

In Prototype objects' creation go through cloning, it ease object's creation. By making a request for cloning we create new cloned object each time, which might be mistaken for flyweight objects, because they share the similar (quite the same, actually) states.

In Flyweight by making a request we try to reuse as much objects as possible by sharing them. New required object will be created if we don't find such one.

While in Prototype we could clone even one object, Flyweight pattern makes sense to use when in the application we use big number of objects.