

Flyweight Pattern

Some programs require a large number of objects that have some shared state among them. Consider for example a game of war, where there is a large number of soldier objects; a soldier object maintain the graphical representation of a soldier, soldier behavior such as motion, and firing weapons, in addition soldier's health and location on the war terrain. Creating a large number of soldier objects is a necessity however it would incur a huge memory cost. Note that although the representation and behavior of a soldier is the same their health and location can vary greatly.

Motivation

Some programs require a large number of objects that have some shared state among them. Consider for example a game of war, where there is a large number of soldier objects; a soldier object maintain the graphical representation of a soldier, soldier behavior such as motion, and firing weapons, in addition soldier's health and location on the war terrain. Creating a large number of soldier objects is a necessity however it would incur a huge memory cost. Note that although the representation and behavior of a soldier is the same their health and location can vary greatly.

Intent

- The intent of this pattern is to use sharing to support a large number of objects that have part of their internal state in common where the other part of state can vary.

Implementation

The figure below shows a UML class diagram for the Flyweight Pattern:

- Flyweight - Declares an interface through which flyweights can receive and act on extrinsic state.
 - ConcreteFlyweight - Implements the Flyweight interface and stores intrinsic state. A ConcreteFlyweight object must be sharable. The Concrete flyweight object must maintain state that is intrinsic to it, and must be able to manipulate state that is extrinsic. In the war game example graphical representation is an intrinsic state, where location and health states are extrinsic. Soldier moves, the motion behavior manipulates the external state (location) to create a new location.
 - FlyweightFactory - The factory creates and manages flyweight objects. In addition the factory ensures sharing of the flyweight objects. The factory maintains a pool of different flyweight objects and returns an object from the pool if it is already created, adds one to the pool and returns it in case it is new.
- In the war example a Soldier Flyweight factory can create two types of flyweights : a Soldier flyweight, as well as a Colonel Flyweight. When the Client asks the Factory for a soldier, the factory checks to see if there is a soldier in the pool, if there is, it is returned to the client, if there is no soldier in pool, a soldier is created, added to pool, and returned to the client, the next time a client asks for a soldier, the soldier created previously is returned, no new soldier is created.
- Client - A client maintains references to flyweights in addition to computing and maintaining extrinsic state

A client needs a flyweight object; it calls the factory to get the flyweight object. The factory checks a pool of flyweights to determine if a flyweight object of the requested type is in the pool, if there is, the reference to that object is returned. If there is no object of the required type, the factory creates a flyweight of the requested type, adds it to the pool, and returns a reference to the flyweight. The flyweight maintains intrinsic state (state that is shared among the large number

of objects that we have created the flyweight for) and provides methods to manipulate external state (State that vary from object to object and is not common among the objects we have created the flyweight for).

Applicability & Examples

The flyweight pattern applies to a program using a huge number of objects that have part of their internal state in common where the other part of state can vary. The pattern is used when the larger part of the object's state can be made extrinsic (external to that object).

Example - The war game.

The war game instantiates 5 Soldier clients, each client maintains its internal state which is extrinsic to the soldier flyweight. And Although 5 clients have been instantiated only one flyweight Soldier has been used.

Source: [Click here to see java source code](#)

Specific problems and implementation

Text Editors

Object oriented text editors need to create Character Objects to represent each character that is in the document. A Character object maintains information about what is the character, what is its font, what is the size of the character, as well as character location inside the document. A document typically consists of extremely large number of character objects which requires large memory. Note that the number of characters in general (Digits, Letters, Other special characters) is known and is fixed, and the fonts that can be applied to each character are also known, thus by creating a Letter flyweight that maintains Character Type (letter, digit, etc...), as well as font, and by creating a Letter Client object that only maintains each character's location inside the document, we have reduced the editor's memory requirements drastically.

Consequences

Flyweight pattern saves memory by sharing flyweight objects among clients. The amount of memory saved generally depends on the number of flyweight categories saved (for example a soldier category and a lieutenant category as discussed earlier).

Related Patterns

Factory and Singleton patterns - 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 Patterns - State and Strategy objects are usually implemented as Flyweights.

Known Uses

Games with graphics as discussed with the War Game Example

Text Editors as discussed in the Text Editors example.