Nicholas DeMarinis

Object Oriented Analysis & Design

19 February 2013

Deep Dive into Patterns: The Flyweight

The Flyweight pattern provides a method for sharing attributes and behavior between large numbers of objects. The Gang of Four book describes a flyweight as “a shared object that can be used in multiple contexts simultaneously,” meaning that a flyweight object can be shared among many objects across different implementations, or contexts, at the same time without the need for complex synchronization routines each time an object is used. To do this, a flyweight decouples the elements of an object that can be shared from those that are context-dependent. The information that can be shared—that is, data which holds true regardless of the flyweight’s context—is stored in the flyweight object; the Gang of Four describe this as the object’s *intrinsic state*. Conversely, the context-dependent data, or the *extrinsic state*, must be maintained by the flyweight’s context and passed to the flyweight when used (Gamma et al, 1994). Since each flyweight only stores its intrinsic state—which, by definition, only includes information that is valid for all contexts—each context can use the same copy of a flyweight. In this way, a system with many similar objects can share data among those objects by encapsulating the intrinsic data in flyweights. In this way, the shared nature of flyweights can reduce the memory footprint of objects with common data, reducing the overhead incurred by systems with large quantities of objects.

To demonstrate how flyweights can be shared among many objects, the GoF book and most other resources describe the canonical example of a rich document editor. Ideally, each character in the editor should be represented by its own object to allow for fine-grained control of the font, text style, as well as other attributes and behavior. However, representing every character as an object in memory would be prohibitive, even in a small document. To help solve this problem, every character in the editor has common and immutable attributes, most notably its character code (and possibly common operations like drawing the character in a window), which can be encapsulated into a flyweight and shared among other objects with the same attributes (Gamma et al, 1994). Using this implementation, each character object no longer needs to store its own character code; instead, it just needs to maintain a reference to the flyweight. This shifts the responsibility of maintaining the data from the character objects, which exist in great quantity, to a comparatively set of flyweights, thus reducing the overhead required to maintain the most numerous objects. The GoF book illustrates this shift of responsibility as shown in Figure 1 below.

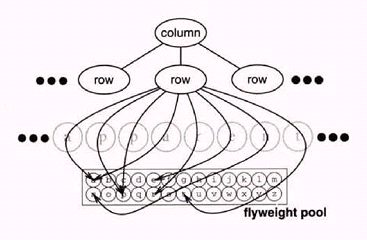


Figure : Flyweights maintaining the character codes for a document editor

Here, each character on a particular row of the editor references a flyweight, which maintains the character’s intrinsic state. While this reduces the overhead required to store each character, it requires some additional complexity to maintain the flyweight objects and support the characters’ extrinsic state—in this case, their style and font information. This can be resolved by adding a factory to handle creation and passing of the flyweight objects, and abstracting the extrinsic state each flyweight needs to operate, demonstrated in Figure 2.

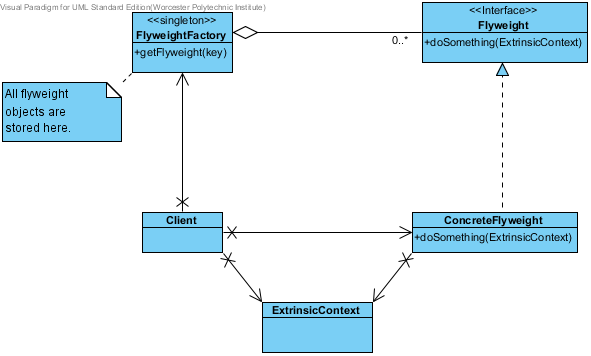


Figure : A Generic Flyweight Implementation

In this way, the client can obtain flyweight objects (characters, in this case) from the pool using the factory, which maintains the pool of flyweight objects. The factory typically handles this using a hash table indexed by a key provided by the client based on its intrinsic state (here, the character code). In this way, clients placing requests for flyweight objects with the same key will receive and therefore share the same object. If the client requests a flyweight not in the table (and thus not yet used by another object), the factory creates it. By using a factory to maintain the flyweight objects, the implementation can ensure that clients only use a single copy of each flyweight.

In addition to the intrinsic state provided by the flyweight, extrinsic state also affects the behavior of an object using a flyweight. Since flyweight objects themselves need to be the same in each context, the client must provide the extrinsic state needed for the operation when calling the flyweight in its own context. By encapsulating the client’s state and only passing it to the flyweight when needed, the flyweight itself remains context-independent while allowing it to perform context-dependent operations. In this case of the character example, the client (a particular row of the editor) might pass font or location information to the character flyweight to draw it on the screen.

Encapsulating state in this manner brings up two other patterns that utilize an encapsulated context in a similar matter: strategy and state. Both of these encapsulate intrinsically-shared information (a specific behavior), but use it based on a context provided by the client. In this sense, both strategy and state objects could be implemented by flyweights, allowing behavior-based information to be shared among many similar objects (Anderson, 2007). In this way, the object’s context only maintains its variable components, while delegating all of the responsibilities for its behavior (in the case of strategy and state) or shared storage to shared objects, which is definitely a powerful abstraction for large systems with many objects.

Works Cited

Anderson, Kenneth. *State and Flyweight Patterns.* University of Colorado, 29 November 2007. 18 February 2013. Retrieved from <http://www.cs.colorado.edu/~kena/classes/6448/f07/lectures/28/28-StateFlyweight.pdf>

Gamma, Erich; Helm, Richard; Johnson, Ralph; John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional, 1994, p. 198-206.

Sugrue, James. *Design Patterns Uncovered: The Flyweight Pattern*. DZone, 2010. 18 February 2013. Retrieved from <http://java.dzone.com/articles/design-patterns-flyweight>