TEC | Tecnológico de Costa Rica

Escuela de Ingeniería en Computación
Área de Ingeniería en Computadores
Algoritmos y Estructuras de Datos II

Tarea Extra-Clase #3: Algoritmos de Encriptación

Elaborado por:
Bradly Valverde Fernández
Gustavo Hernández Granera
Greivin Salas Oconitrillo
Josué Chaves Araya
Elías Arce Méndez

Profesor:
Pedro Gutiérrez García

Grupo:
01

07 de octubre, 2018

Introduction

Gamma et al. 1994 says that structural patterns are concerned with how classes and objects are composed to form larger structures. Structural class patterns use inheritance to compose interfaces or implementations. Here will be explained seven of the most important structural patterns that are used today which are adapter, bridge, composite, decorator, facade, flyweight and proxy.

GoFPatterns (no date) in its structural patterns web course says that good toolbox of structural patterns allows you to solve many thorny problems you are likely to encounter. They show you how to glue different pieces of a system together in a flexible and extensible fashion. Structural patterns help you guarantee that when one of the parts changes, the entire structure does not need to change. They also show you how to recast pieces that do not fit (but that you need to use) into pieces that do fit.

Structural patterns include some quite sophisticated and powerful techniques. However, they are based on just a few basic techniques, primarily:

1. Delegation: The pattern is one in which a given object provides an interface to a set of operations. However, the actual work for those operations is performed by one or more other objects.
2. Object composition: Other objects are stored as pointers or references inside the object that provides the interface to clients. Object composition is a powerful yet often overlooked tool in the OOP programmer's toolbox. Structural patterns show you how to take advantage of it.

Structural patterns have many beneficial effects including:

1. Increased efficiency (Decorator, Flyweight, Proxy).
2. Enhanced reusability (Adapter, Bridge, Decorator).
3. Separating implementation from interface (Adapter, Bridge, Façade, Proxy).
4. Reducing complexity by providing cleaner, simpler interfaces to a system that are easier for client programmers to understand (Adapter, Bridge, Composite, Façade, Proxy).

Bridge

Bridge is a design pattern that help when in a software we want to implement two or more different types of systems, for example: We have a videogame that was released for Windows this has a Online Multiplayer and Solo Story Mode, now the programmers want to implements this to MAC OS and Linux, they need to create a implementation for Online Multiplayer and Story Mode and then Sony and Nintendo want that game in their platforms, for this the programmers of the game have to add 8 kinds of implements to the code, that´s very inefficient, lucky to us Bridge come to help, the main function is to separate Implementation from the abstraction.

Bridge is known as Handle/Body pattern too. This pattern is use when:

- Want to make a implementation to some platforms.
- Map an orthogonal class hierarchies.
- There are a proliferation of implementation classes from a complicated abstraction.

Example code:
https://github.com/delias2798/Structural_Patterns/blob/master/Bridge.cpp

Proxy

Proxy is a design pattern that make a mediotary class between the client and the real object to manipulate many functions laki access that no needed to change the original (real) class.

There are many kind of proxies:

- Remote: represent an Object in other direction.
- Virtual: creates a expensive Object.
- Protection: manipulate the access of the Object.
- Smart: do many things such a: Count the number of references yo manage the memory, see if the object is locked to ensure that no other reference can change it.

Example code:
https://github.com/delias2798/Structural_Patterns/blob/master/Proxy.cpp

Decorator

The decorator pattern is about create transparent objects from a original object and the result transparent object bring new method and responsibility. That is useful because you can make your personal object adding differents characteristics in any moment you want, and maybe you are thinking in inheritance throw adding the methods or characteristics in the main class who create the original object and that can be right in some cases but in so much contexts isn't possible because adding the methods to the main class would make it inflexible.

So, In which context you use that? The question can be answered with example: Imagine that you have a textview and you want a scrollbar, but you can't add a scrollbar in the main class because no every textview need a scrollbar and reason of that the decorator pattern was created, you can make a decorator scrollbar class and use the textview class as base, then if you want to add more responsibilities like a toolbar or a border you just have to do this in a recursive way. Do that is viable because the transparency.

Example code:

https://github.com/delias2798/Structural_Patterns/blob/master/Decorator.cpp

Flyweight

Flyweight is about share objects to reduce cost of implementation and space, this is useful when you have a lots of objects, when storage costs are high or when you have many groups of objects may be replaced by relatively few shared objects.

The shared objects of flyweight pattern work in situation when you have a differents objects who present own characteristics and the problem is: you have a lots of objects who present own characteristics, so in that situation you share a object (independent object) and stored an information that's independent of the flyweight's context, that concept is called intrinsic. For other side we have the concept extrinsic state who depends on and varies with the flyweights context and therefore can't be shared.

An example of this pattern is the document editor, the editor use objects to represent characters, rows and columns; and that is inefficient because it create a hundreds of character objects that result expensive for memory and may incur

unacceptable run-time overhead. The solution appear in flyweight pattern when created only one type of object and share objects to allow their use at fine granularities without prohibitive cost.
Example code:
https://github.com/delias2798/Structural_Patterns/blob/master/Flyweight.cpp

Façade

This pattern was made to provide a point of entry to a system, this system usually has a lot of components and can be a little of a mess inside, so Façade creates an interface to interact with it. It kind of reminds us to an object oriented principle, and that is encapsulation, façade hides all complicated things, and glues out all the functionality, in a single, easy to use interface, like we said before.

We use façade when we want an interface to a complex subsystem, when we want to promote independence and portability on a system, and to layer a system into subsystems each of one with a façade to communicate.

For example, imagine a car, let’s start it, you don’t go to the engine right? you get in the front seat and turn the key, if there is no problem, the engine starts, now let’s say you have to charge your phone or your laptop, you don’t rip your device apart and take the battery off, you just connect the cable to the charging port.
Example code:
https://github.com/delias2798/Structural_Patterns/blob/master/EjemploFacade.cpp

Adapter

Adapter, it’s a very useful one, it’s comparable to façade in the way that with adapter, we create interfaces, but what adapter really does, is to convert the interface of a class into another one that the client expects, also it makes different classes with incompatible interfaces, work together creating new interfaces that let them use all of their functionality, or let’s say you have an old system, and you need to use it in this super new one, you can code an adapter to make the interface of that

old system work with the new one, without the need to change any of those internally.
Example code:
https://github.com/delias2798/Structural_Patterns/blob/master/EjemploAdapter.cpp

Composite

This design pattern allows us, through a recursive composition strategy, to construct complex objects based on simpler ones that can exist by themselves as a system or also by other complex objects.

By the form of this pattern you can create objects as complex as you wish, you only have to create new complex objects that contain all or some of the previous objects and so on, creating a hierarchy of objects.

An example to understand it better is a book, a book is formed by pages, so the simplest object is the pages and the book would be the complex object. Next, if you want to create a more complex object, this could be a bookshelf, and in this way you can create more and more complex objects starting from the previous ones.
Example code:
https://github.com/delias2798/Structural_Patterns/tree/master/composite

Discussion

You may have noticed similarities between the structural patterns, especially in their participants and collaborations. This is so probably because structural patterns rely on the same small set of language mechanisms for structuring code and objects: single and multiple inheritance for class-based patterns, and object composition for object patterns. Adapter vs Bridge: The Adapter and Bridge patterns have some common attributes. Both promote flexibility by providing a level of indirection to another object. Both involve forwarding requests to this object from an interface other than its own. The key difference between these patterns lies in their intents.Adapter focuses on resolving incompatibilities between two existing interfaces. It doesn't focus on how those interfaces are implemented,nor does it consider how they might evolve independently. It's a way of making two independently designed classes work together without reimplementing one or the other. Bridge, on the other hand, bridges an abstraction and its (potentially numerous) implementations. It provides a stable interface to clients even as it lets you vary the classes that implement it. It also accommodates new implementations as the system evolves. Composite vs Decorator vs Proxy: Composite (183) and Decorator (196) have similar structure diagrams, reflecting the fact that both rely on recursive composition to organize an open-ended number of objects.This commonality might tempt you to think of a decorator object as a degenerate composite, but that misses the point of the Decorator pattern. The similarity ends at recursive composition, again because of differing intents. Decorator is designed to let you add responsibilities to objects without subclassing. It avoids the explosion of subclasses that can arise from trying to cover every combination of responsibilities statically. Composite has a different intent. It focuses on structuring classes so that many related objects can be treated uniformly, and multiple objects can be treated as one. Its focus is not on embellishment but on representation. Another pattern with a structure similar to Decorator's is Proxy. Both patterns describe how to provide a level of indirection to an object, and the implementations of both the proxy and decorator object keep a reference to another object to which they forward requests. Unlike Decorator, the Proxy pattern is not concerned with attaching or detaching properties dynamically, and it's not designed for recursive composition. In the Proxy pattern, the subject defines the key functionality, andthe proxy provides (or refuses) access to it. In Decorator, the component provides only part of the functionality, and one or more decorators furnish the rest (Gamma et al. 1994).

References

Gamma, E. et al. (2002). Design Patterns: : elements of reusable object-oriented software. Madrid: Addison Wesley.

GoFPatterns (no date). Structural Patterns. Recovered from: https://www.gofpatterns.com/design-patterns/module5/intro-structural-designPatterns.php