**Structural Patterns**

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.

Example: class form of the Adapter

**Structural object patterns** describe ways to compose objects, rather than composing interfaces or implementations, to realize new functionality.

The added flexibility of object composition comes from the ability to change the composition at run-time, which is impossible with static class composition.

Example: object form of the Adapter, Composite, Proxy, Flyweight, Decorator

# Adapter

Convert the interface of a class into another interface clients expect.

Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

Often the adapter is responsible for functionality the adapted class doesn't provide.

An adapter often becomes necessary when you discover that two incompatible classes should work together, generally to avoid replicating code.

The Adapter pattern makes things work after they're designed.

An adapter makes two existing interfaces work together as opposed to defining an entirely new one.

## Adapter (class)

A class adapter uses multiple inheritance to adapt interfaces.

Use one inheritance branch to inherit the interface and another branch to inherit the implementation.

In C++, inherit the interface publicly and inherit the implementation privately.

## Adapter (object)

The object adapter uses object composition to combine classes with different interfaces.

The object adapter requires a little more effort to write, but it's more flexible.

The object adapter version will work equally well with subclasses of adaptee —the client simply passes an instance of a subclass of adaptee to the adapter constructor.

# Bridge

# Composite

Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

Composite 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.

Graphics applications like drawing editors and schematic capture systems let users build complex diagrams out of simple components. A simple implementation could define classes for graphical primitives such as Text and Line s plus other classes that act as containers for these primitives.

# Decorator

# Facade

# Flyweight

# Proxy

Provide a surrogate or placeholder for another object to control access to it.

One reason for controlling access to an object is to defer the full cost of its creation and initialization until we actually need to use it.

In the Proxy pattern, the subject defines the key functionality, and the proxy provides (or refuses) access to it.

Consider a document editor that can embed graphical objects in a document. Some graphical objects, like large raster images, can be expensive to create. But opening a document should be fast, so we should avoid creating all the expensive objects at once when the document is opened. These constraints would suggest creating each expensive object on demand, which in this case occurs when an image becomes visible.

# References

Book: Design Patterns Elements of Reusable Object-Oriented Software