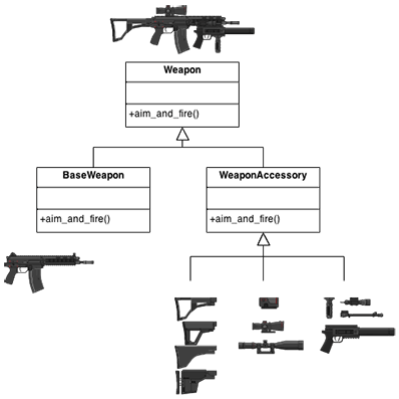
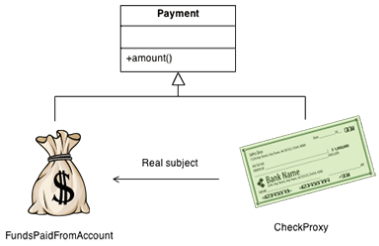
Structural patterns

In Software Engineering, Structural Design Patterns are Design Patterns that ease the design by identifying a simple way to realize relationships between entities.



- **Adapter**
  Match interfaces of different classes
- **Bridge**
  Separates an object's interface from its implementation
- **Composite**
  A tree structure of simple and composite objects
- **Decorator**
  Add responsibilities to objects dynamically
- **Facade**
  A single class that represents an entire subsystem
- **Flyweight**
  A fine-grained instance used for efficient sharing
- **Private Class Data**
  Restricts accessor/mutator access
- **Proxy**
  An object representing another object



Rules of thumb

1. **Adapter** makes things work after they're designed; **Bridge** makes them work before they are.

2. **Bridge** is designed up-front to let the abstraction and the implementation vary independently. **Adapter** is retrofitted to make unrelated classes work together.

3. **Adapter** provides a different interface to its subject. **Proxy** provides the same interface. **Decorator** provides an enhanced interface.

4. **Adapter** changes an object's interface, **Decorator** enhances an object's responsibilities. **Decorator** is thus more transparent to the client. As a consequence, **Decorator** supports recursive composition, which isn't possible with pure **Adapters**.

5. **Composite** and **Decorator** have similar structure diagrams, reflecting the fact that both rely on recursive composition to organize an open-ended number of objects.

6. **Composite** can be traversed with **Iterator**. **Visitor** can apply an operation over a **Composite**. **Composite** could use **Chain of responsibility** to let components access global properties through their parent. It could also use **Decorator** to override these properties on parts of the composition. It could use **Observer** to tie one object structure to another and **State** to let a component change its behavior as its state changes.

7. **Composite** can let you compose a **Mediator** out of smaller pieces through recursive composition.

8. **Decorator** lets you change the skin of an object. **Strategy** lets you change the guts.

9. **Decorator** is designed to let you add responsibilities to objects without subclassing. **Composite**'s focus is not on embellishment but on representation. These intents are distinct but complementary. Consequently, **Composite** and **Decorator** are often used in concert.

10. **Decorator** and **Proxy** have different purposes but similar structures. Both describe how to provide a level of indirection to another object, and the implementations keep a reference to the object to which they forward requests.

11. **Facade** defines a new interface, whereas **Adapter** reuses an old interface. Remember that **Adapter** makes two existing interfaces work together as opposed to defining an entirely new one.

12. **Facade** objects are often **Singleton** because only one **Facade** object is required.

13. **Mediator** is similar to **Facade** in that it abstracts functionality of existing classes. **Mediator** abstracts/centralizes arbitrary communication between colleague objects, it routinely "adds value", and it is known/referenced by the colleague objects. In contrast, **Facade** defines a simpler

interface to a subsystem, it doesn't add new functionality, and it is not known by the subsystem classes.

14. **Abstract Factory** can be used as an alternative to **Facade** to hide platform-specific classes.

15. Whereas **Flyweight** shows how to make lots of little objects, **Facade** shows how to make a single object represent an entire subsystem.

16. **Flyweight** is often combined with **Composite** to implement shared leaf nodes.

17. **Flyweight** explains when and how **State** objects can be shared.

Read next

This article is taken from our book **Design Patterns Explained Simply**.

All of the design patterns are compiled there. The book is written in clear, simple language that makes it easy to read and understand (just like this article).

We distribute it in PDF & EPUB formats so you can get it onto your iPad, Kindle, or other portable device immediately after your purchase.

Learn more

Code examples

READ NEXT

Adapter

RETURN

Design Patterns
AntiPatterns
Refactoring
UML

My account
Forum
Contact us
About us