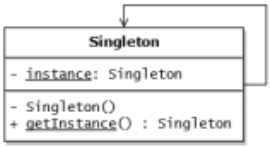
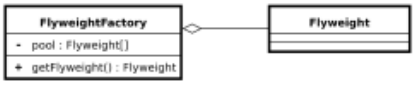
Design patterns:

Gang of Four Patterns

Original book describes 23 patterns – 5 Creational, 7 Structural, and 11 Behavioural.

Creational Patterns	Behavioural Patterns	Structural Patterns
Abstract Factory	Chain of Responsibility	Adapter
Builder	Command	Bridge
Factory Method	Interpreter	Composite
Prototype	Iterator	Decorator
Singleton	Mediator	Façade
	Memento	Flyweight
	Observer	Proxy
	State	
	Strategy	
	Template Method	
	Visitor	





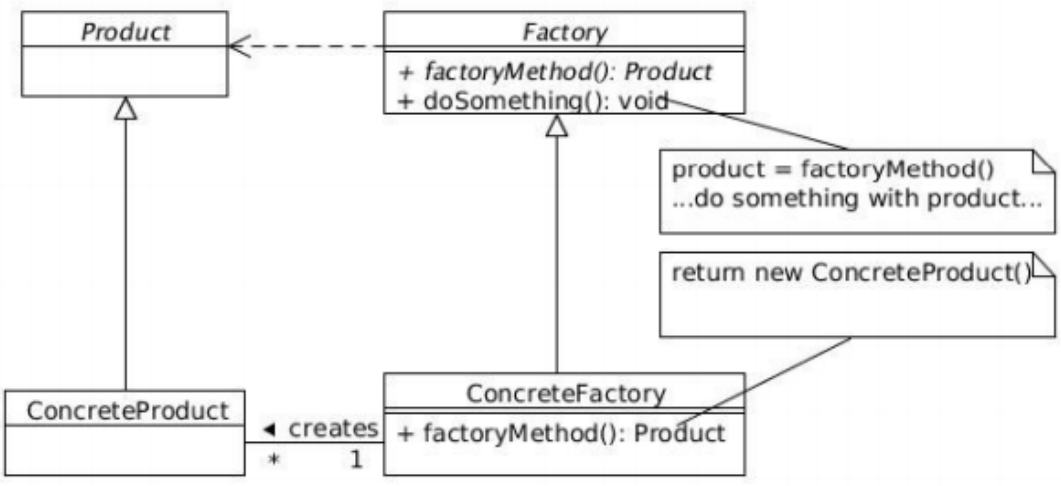
Creational Patterns

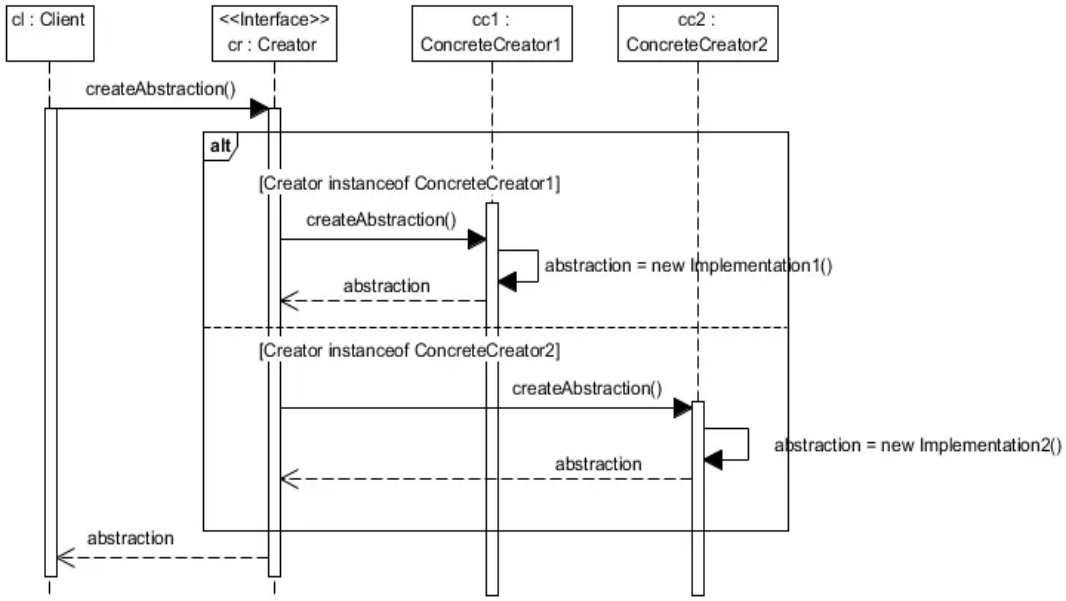
1. Factory Method

Intent: Define an interface for creating an object but let subclasses decide which class to instantiate. Factory method allows classes to differ instantiation to subclasses

Consequences: Creating objects in a class with a factory method is more flexible

• Connects parallel class hierarchies – Factory methods can be called from either creators or from parallel classes

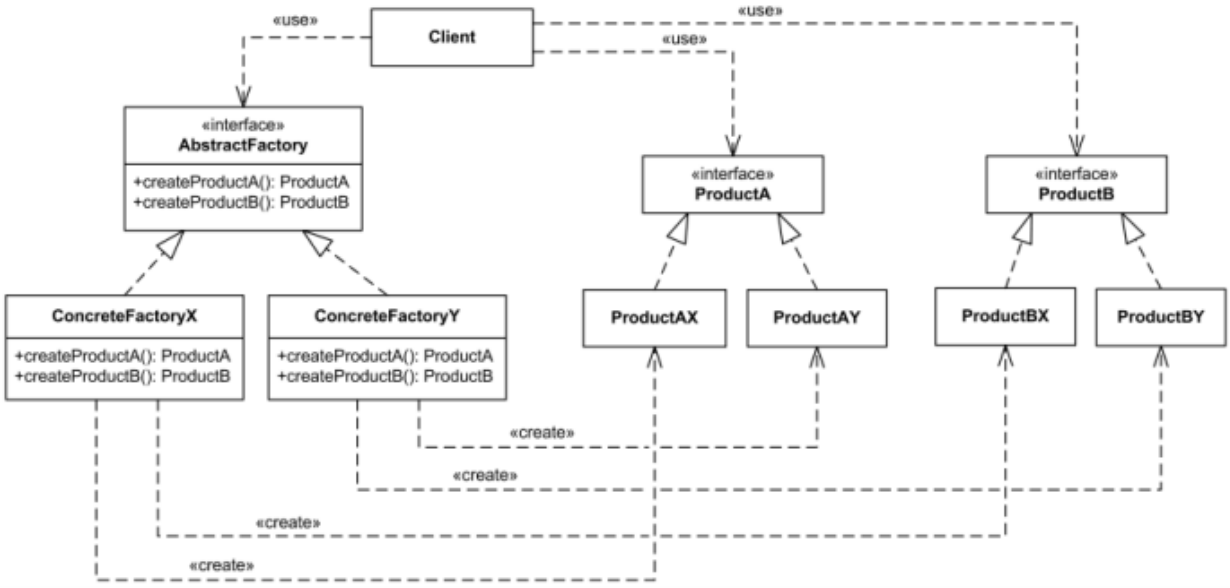




2. Abstract Factory

Intent: Provide an interface for creating families of related or dependent objects without specifying their concrete classes

Consequences: isolates concrete classes – You can remove the use of concrete class names in client code, meaning if you need to change a concrete implementation, the client never needs to know • It makes exchanging product families easy – Switching out abstract factories can result in a new functionality without much code change at all • It promotes consistency among products – as all products will follow a similar design, you end up with more consistent classes • Extending product families (new products in same family) is very difficult –because the abstract factory defines all the products it can create. Adding becomes a major change.

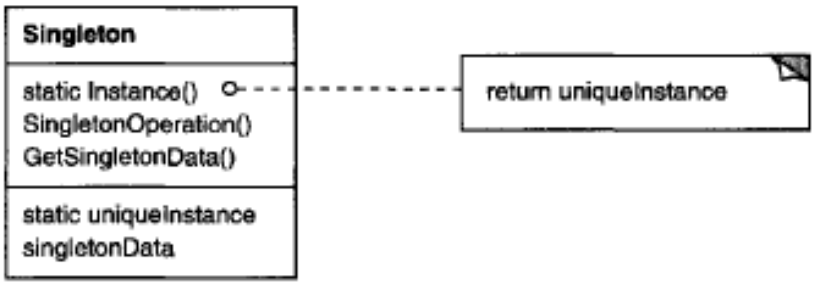


3. Singleton

Intent: Ensure a class only has 1 instance, provide a global point of access

Consequences: Strict control on access – since the singleton has only one point of entry, this can be easily monitored and controlled

• Reduced namespace – The pattern removes pollution of the global namespace by unnecessary repetition

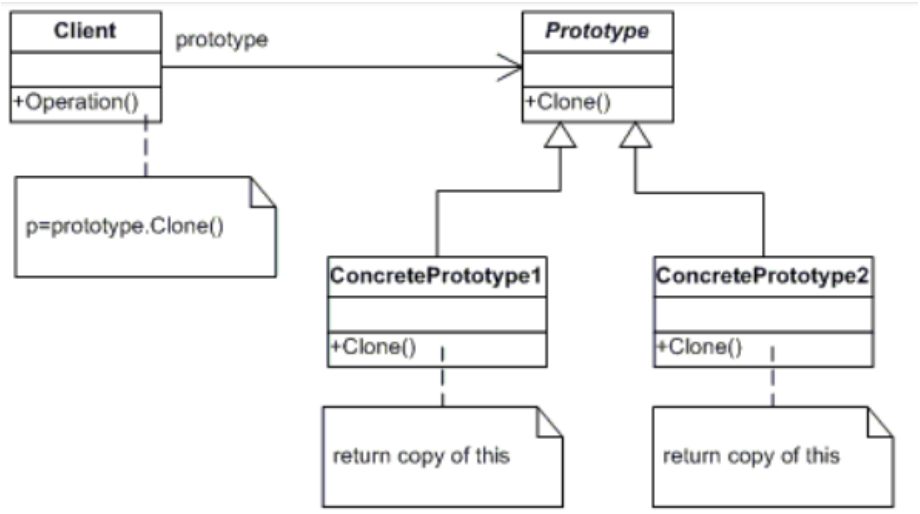


4. Prototype

Intent: • Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype

Consequences: Very similar to Abstract Factory and Builder pattern in terms of consequences • Adding and removing products at runtime – Prototypes let you add and remove concrete product classes simply by registering a prototype instance. This allows for clients to add or remove prototypes at runtime • Reduced subclassing – since you don't need to have a creator class for each concrete class, you reduce the overall complexity of code

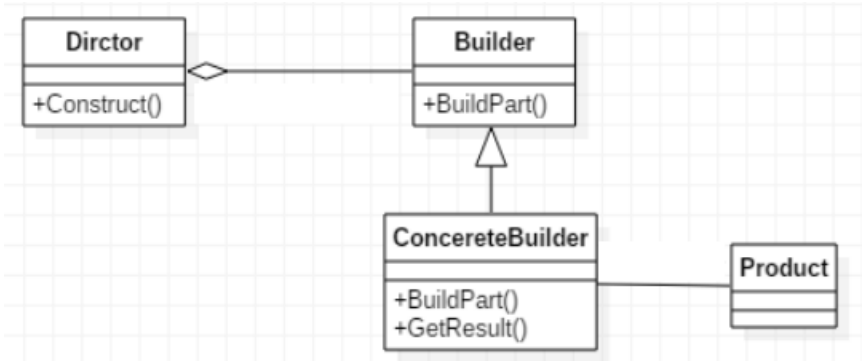
When to use prototype: when a system should be independent of how its products are created/composed/respresented

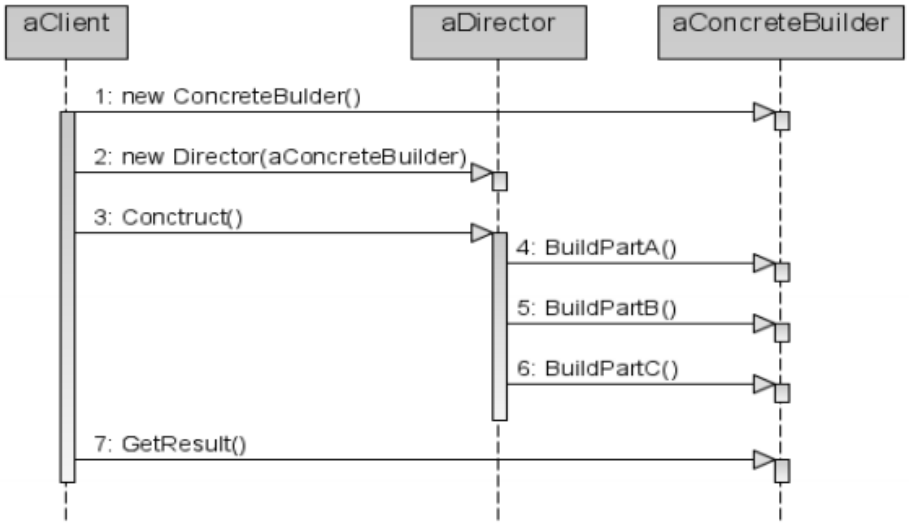


5. Builder

Intent: Separate the construction of a complex object from its representation so that the same construction process can create different representations

Consequences: You can vary the internal representation – Since your product is not in your consumer's hands until the final execution, all you need to do to modify the built representation is modify the builder (or add another builder) • It isolates construction from representation – This improves modularity within code greatly. • It gives you finer control over the construction process – since the builder does not release the product to the director until it is in the final build state, you actually have a lot finer control over what is happening





Behavioural Patterns

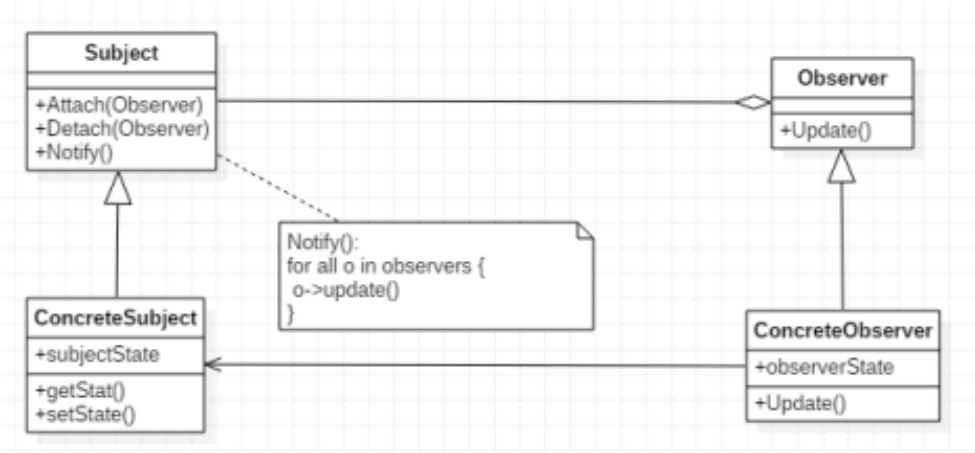
6. Observer

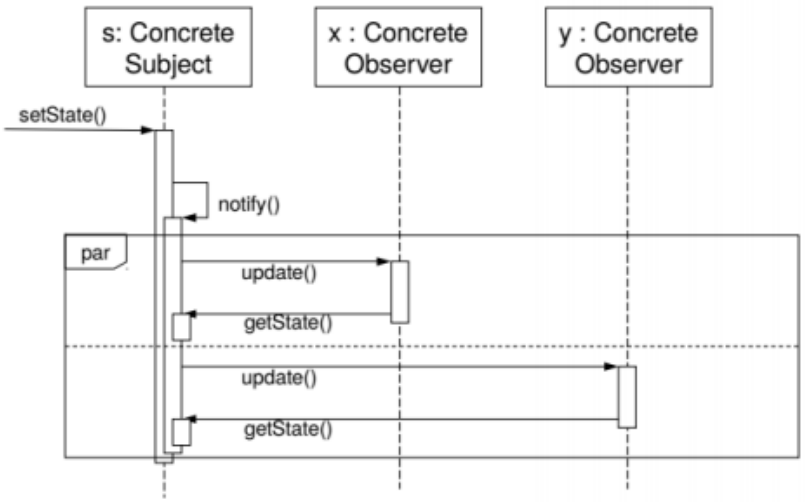
Intent: Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and update automatically

Consequences: There is only light, abstract coupling between the Subject and Observer. Since the Subject only knows it has a list of Observers whom conform to the simple Observer Interface

• Broadcast messaging is achieved, as the Subject does not need to specify the receiver of the message

• Not easy to trace the cost of an update, as everything is so loosely coupled at a design level, but is highly interdependent at an operation level. Each change to the Subject may cause large cascading changes throughout application, if not carefully controlled for

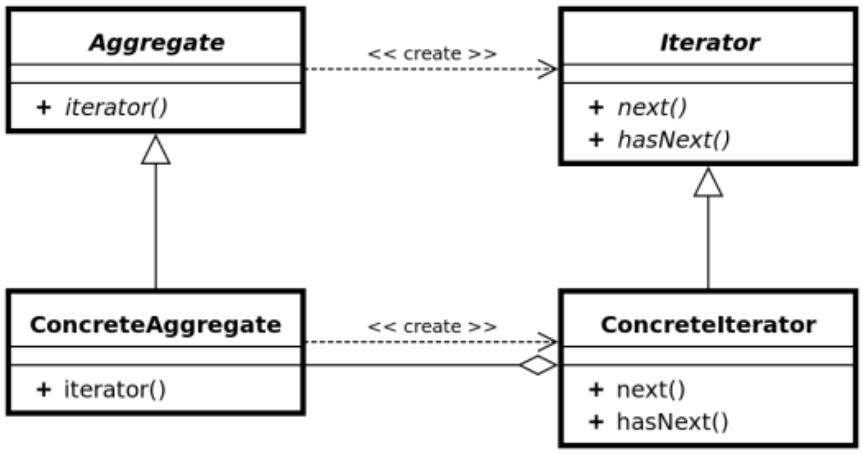




7. Iterator

Intent: Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation

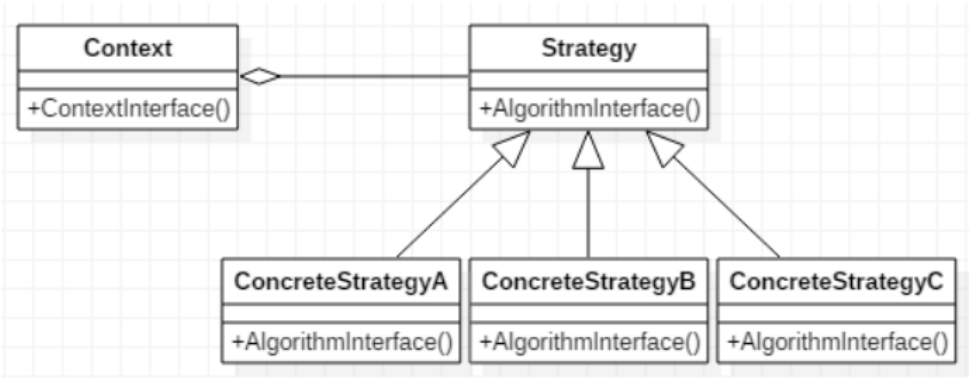
Consequences: Supports variations in the traversal of an aggregate. Complex aggregates may be traversed in many ways • Iterators simplify the aggregate interface by removing details about traversal • More than one traversal can be pending on an aggregate. Iterators only keep track of their own traversal state, and thus, multiple traversals can be occurring in unison



8. Strategy

Intent: Define a family of algorithms, encapsulating each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it

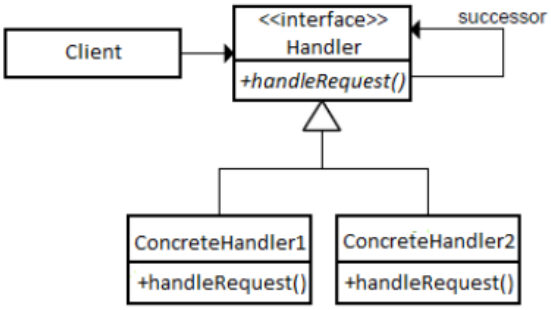
Consequences: • An alternative to sub-classing – Sometimes algorithmic differences are only difference between parent and children classes, Strategy removes the need to create specific children • Eliminate conditional statements – By delegating the choice to the composite class, remove the need for if blocks • Choice of implementation – Client can pick between multiple strategies that have different time or space trade-offs • Clients must be aware of different strategies – The pattern can potentially create issues if client doesn't have full awareness • Communication overhead between strategy and context – Context can create large overhead for simple algorithms that don't need the full set of information that other more complex algorithms need • Increased number of objects – As each strategy requires its own class, the number of objects can increase



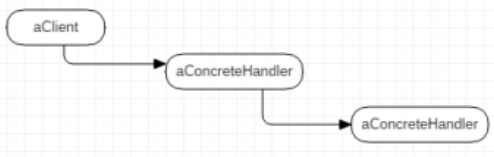
9. Chain of Responsibility

Intent: Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

Consequences: Reduced Coupling – The pattern frees an object from knowing which other object handles a request. • Added flexibility in assigning responsibility – Since the chain can be modified at runtime, you have control over how requests are handled, and can even change based on context • Receipt isn't guaranteed – Since a request has no explicitly defined receiver there is no guarantee it'll be handled. A request can fall off the end of the chain without being handled. This is usually a configuration issue



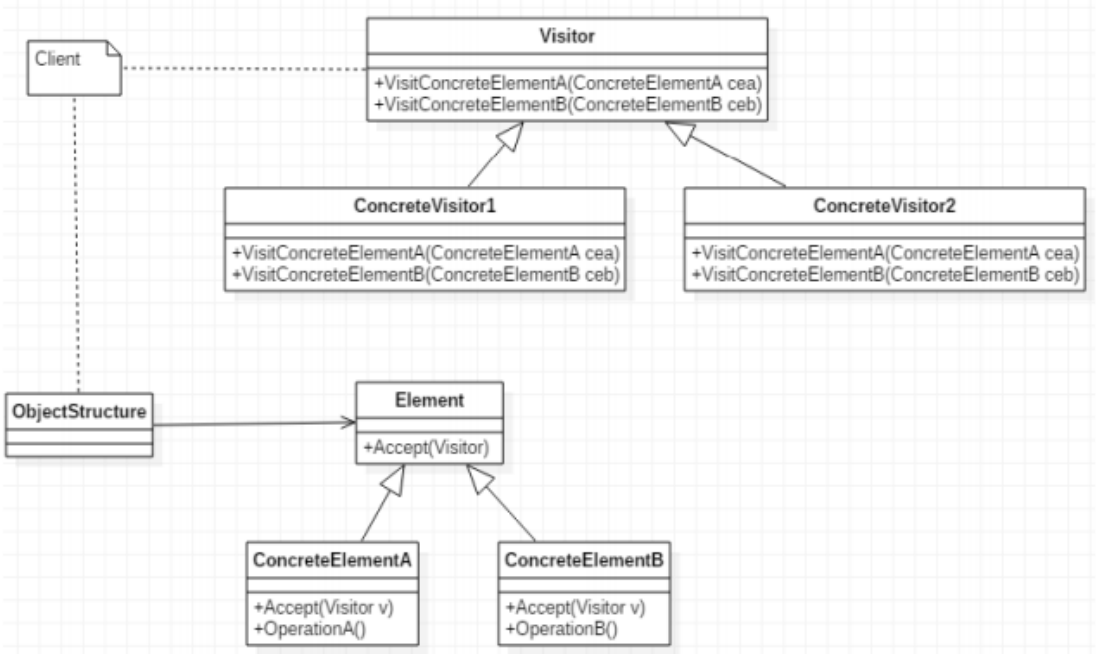
A typical object structure looks something like this:

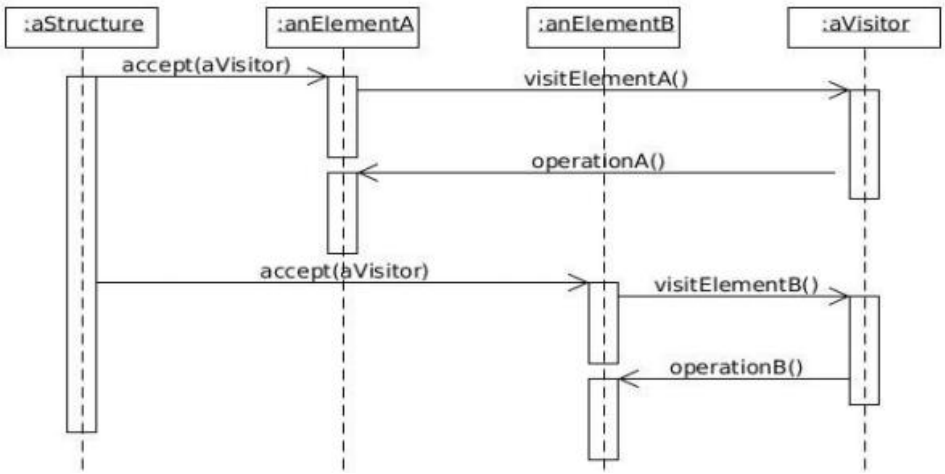


10. Visitor

Intent: Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates

Consequences: Visitor makes adding new operations easy – visitors can be easily created to be run across many objects • A visitor gathers related operations and separates unrelated ones – you end up with tighter coupling behaviourally, meaning if a behaviour needs modification, it's located in a single place • Adding new concrete objects is hard – since you need to make a new operation for this object in all visitor classes. You need to make the decision if you will be frequently changing your algorithms or your object structure • Visiting can occur across classes – Unlike Iterator, that requires each visited node to be of the same type, Visitors can visit classes which are not in the same hierarchy, as long as they have an accept method, and a subsequently appropriate Visit method defined • Accumulating state – visitors can take on the responsibility of accumulating the state of each element in the object structure. Without this, the state would have to be stored and passed independently • Breaking Encapsulation – One of the key consequences of visitor is that it requires you to build public methods which have access to state variables within the class directly. This can easily end up breaking the encapsulation of your code!

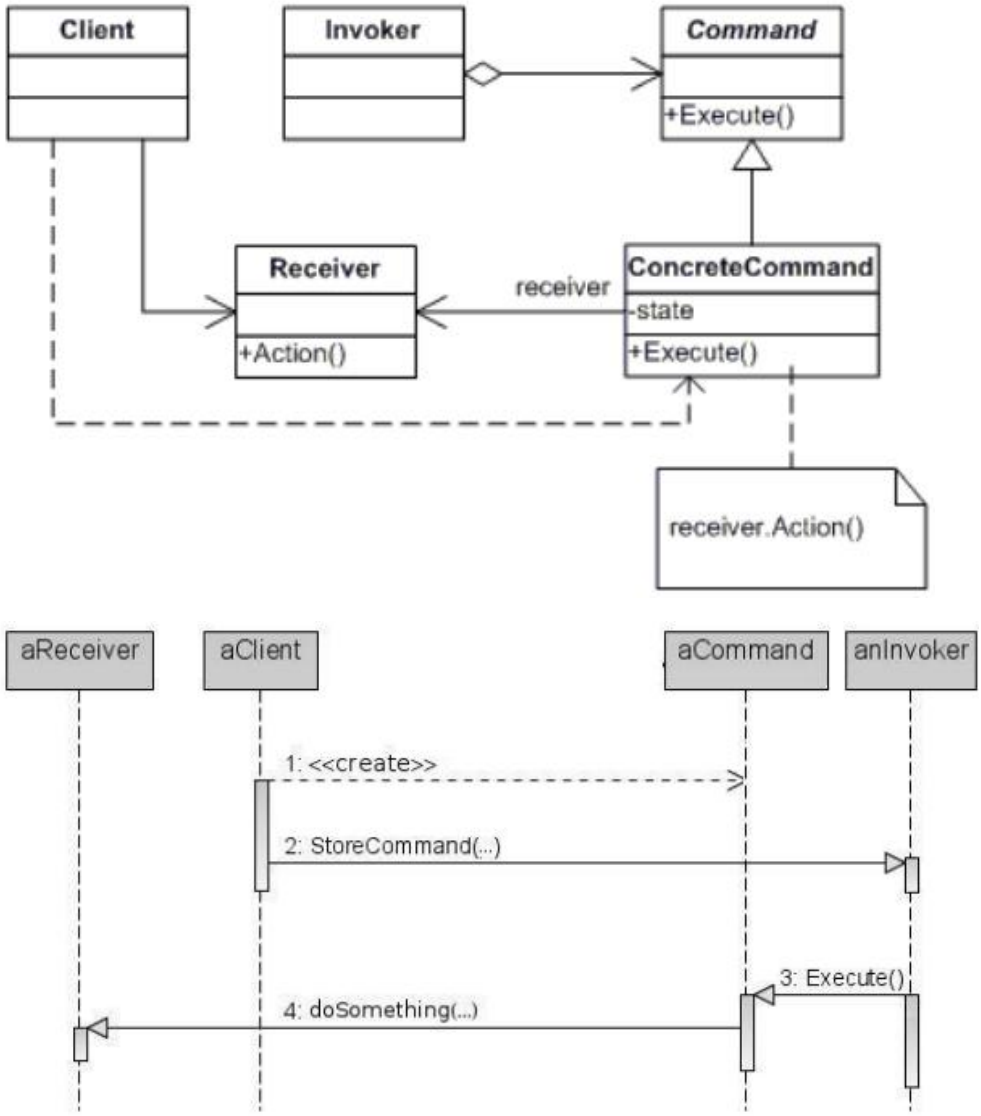




11. Command

Intent: • Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

Consequences: Command decouples the object that invokes the operation from that which performs it • Commands are first=class objects that can thus be manipulated like any other object • It's easy to add new commands because you can just extend • Composite commands can be created (following the composite pattern's behaviour)



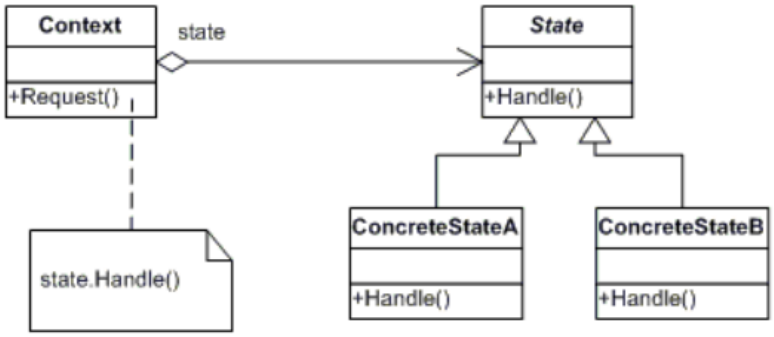
12. State

Intent: Allow an object to alter its behaviour when its internal state changes. The object will appear to change its class.

Consequences: It localizes state-specific behaviour – All state-specific code becomes encapsulated in an object, it means that introducing new states does not impact objects, and updating one state only requires changes to that state object and not the class that will be in said state • It makes state transition specific – Since the type of object that State is subclassed to will result in the state, it becomes very explicit when a transition has occurred • State objects can be shared – If a state has no specific instance variables associated with it, it can be shared between multiple objects, and essentially becomes a flyweight

State is an abstract class
Context will contain a state instance variable instantiated with the specific concrete state belonging to the current state of context.

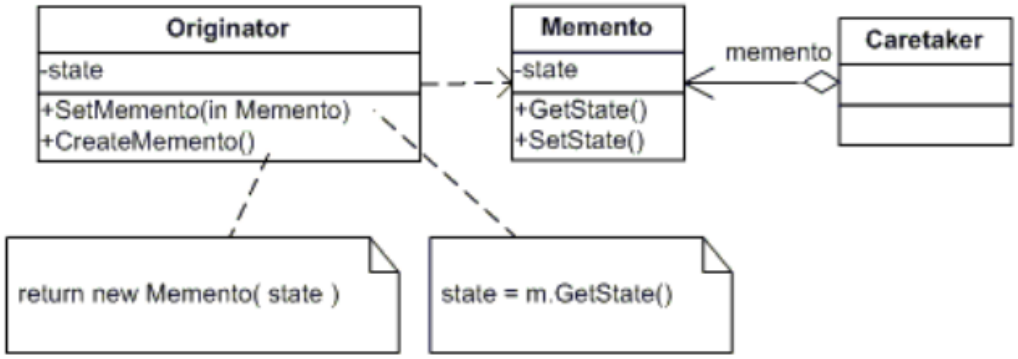
When context methods are called, if they are state specific, they will call upon the state object that is currently held resulting in the state-specific behaviour



13. Memento

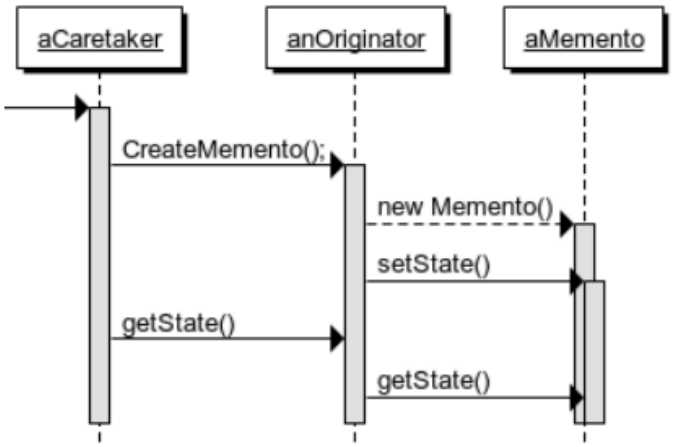
Intent: Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to that state later.

Consequences: Preserves encapsulation boundaries – By storing information external to the originator, but only storing what is necessary to maintain state, the memento preserves encapsulation • Simplifies originator – Without the memento, the originator would have to store several previous states, and require logic for how and when to access these pieces of information • Using Mementos can be expensive – If we follow a naïve approach to how the details are stored, the amount of space required, and processor use to store this information can become very large. • Defining narrow and wide interfaces – It can be difficult to ensure that only the originator has access to the memento's state in certain languages (NOTE: This isn't just about python – think about how best to do this in Java) • Hidden costs in caring for mementos – A caretaker is responsible for deleting mementos, however, the caretaker has no idea how much stat is in the memento. Thus, a caretaker which may seem lightweight can actually be much larger than is expected



Note: The caretaker may never actually call on getState, as the originator will not need to be reverted.

Mementos are passive, only the originator will assign or retrieve a state. The memento has no activities it completes.

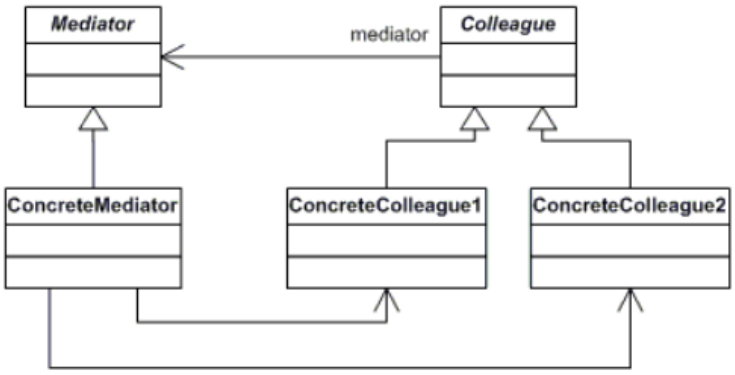


14. Mediator

Intent: Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interactions independently.

Consequences: It limits subclassing – the mediator localizes behaviour that traditionally would be distributed across many subclasses • Decouples classes – by pulling out interaction behaviour, classes stop being closely coupled with one another • It abstracts how objects communicate – objects can be focused on their tasks, and not need to worry about how they will update each other • Centralizes control – the mediator class in the centre takes care of all of the other classes

The Mediator object will store a list of Colleagues, and have functions for when specific colleagues have changed. This behaviour will be setup using an Observer pattern.



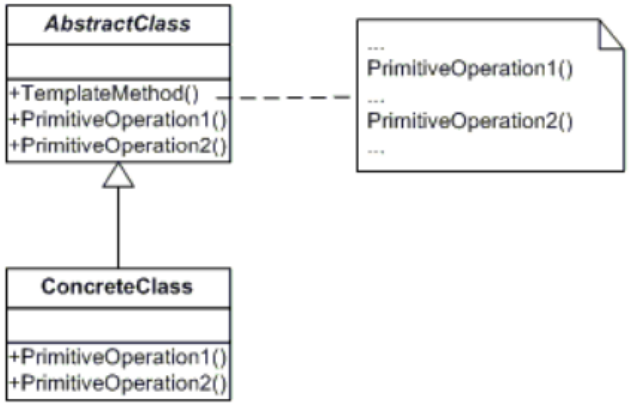
15. Template Method

Intent: Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure

Consequences: Template methods are common in class libraries and implement what is often called the "Hollywood principle*", as the parent class calls the methods of the subclass • Template methods can run in several forms – they can either override parent behaviour entirely, partially, or not at all

The Abstract class has a templated method, made up of several primitive operations, which are usually not defined at the Abstract class level.

Each concrete class implementing the abstract class will then implement these primitive operations. When TemplateMethod is called, it will incorporate the subclass specific versions of these pieces of the method.

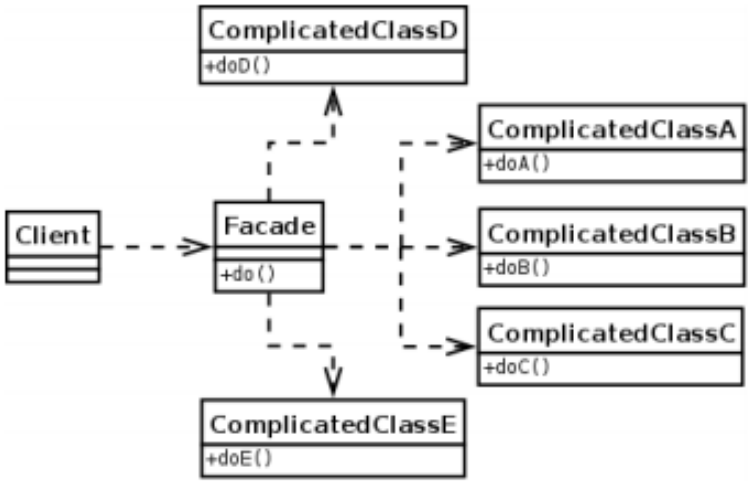


Structural Patterns

17. Facade

Intent: Provide a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystems easier to use.

Consequences: Clients are shielded from subsystem objects, reducing scope of what they need to understand • Promotes weak coupling, between subsystems and clients of that system • It doesn't explicitly prevent direct use of subclasses, but gives an alternative.



18. Adapter

Intent: Convert the interface of a class into another interface to meet a clients demand/expectation. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces

Consequences: Each of the two versions have different consequences which will be broken down on the following slide

Two forms:

Class Adapter

• Adapts Adaptee to Target by committing to a concrete Adaptee class. As a consequence, a class adapter won't work when we want to adapt a class and all its subclasses

• Allows for overriding the Adaptee's behaviour

Object Adapter

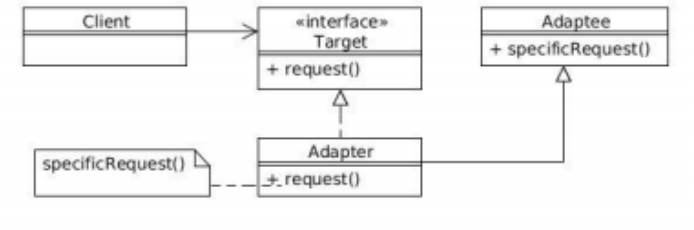
• Lets a single adapter work with many adaptees, for example, all subclasses of the adaptee as well as the adaptee itself

• Makes it difficult to override the methods of the adaptee

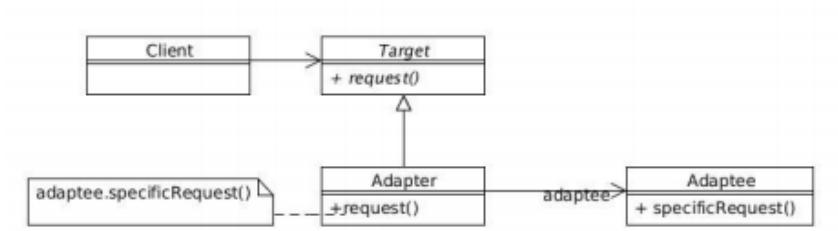
Consequences

• How much adapting should be done – It can be difficult where to draw the line for when this is an appropriate pattern and when it will end up ultimately creating more work • Pluggable adapters – a class is more reusable if you minimize the assumptions other classes make to use it. The term refers to building in adapters from the onset • Using two-way adapters – currently adapter does not allow for two-way adaptation, and it isn't transparent to the client what is happening. This can lead to suboptimal situations

Class Adapter



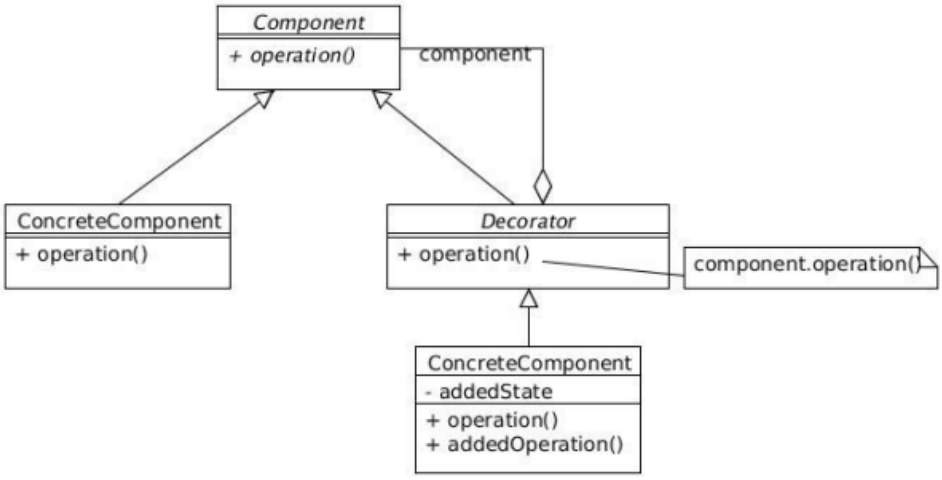
Object Adapter



19. Decorator

Intent: Attach additional responsibilities to an object dynamically. Decorators allow for a flexible alternative to subclassing when there is a need to extend functionality

Consequences: More flexibility than static inheritance – The decorator pattern provides a more flexible way to add responsibilities to objects • Avoids feature-laden classes high up in the hierarchy – decorator offers a pay-as-you-go approach to adding responsibilities to classes • Lots of little objects – these can be a mess to deal with. Can become very complex to debug



20. Composite

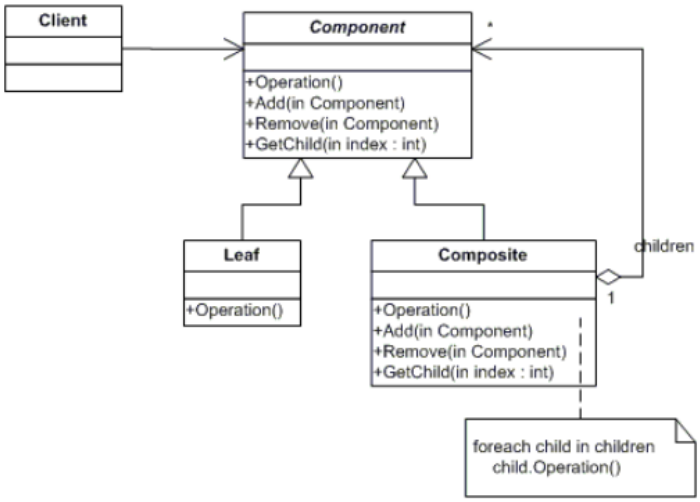
Intent: Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly

Consequences: Primitive objects (Leaf) and composites can be used interchangeably • Clients can treat all related objects the same way, and don't need to worry if they are dealing with a composite or a leaf. This means they don't need complex logic on their end. • Makes adding new components easy – newly defined composite or leaf subclasses work automatically with existing structures and client code. Clients don't have to be changed for new Component classes to work • You need to runtime check if things belong – you cannot rely on hierarchy structures to type check you have correct class – the solution becomes too general

Notes
The composite class is both a component of and an implementation of the component.

The Leaf is any class that has no children and is the building block of our composition. You usually will have many classes like Leaf

The client uses the component interface to manipulate the Leaf objects stored inside the composite.



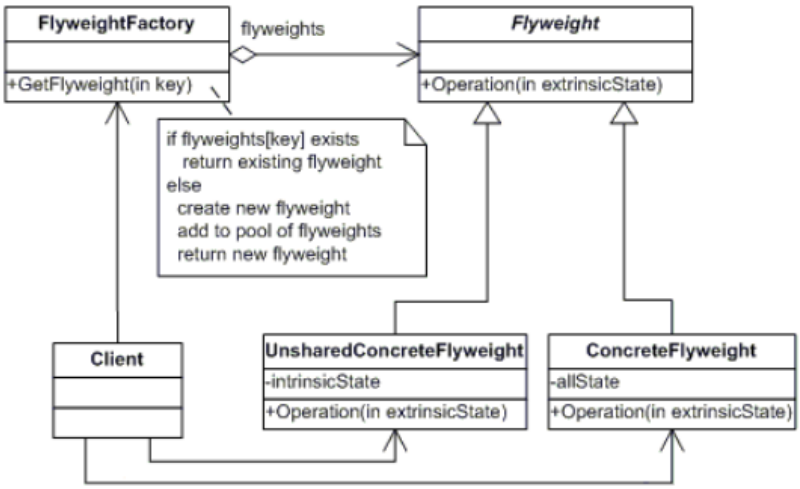
21. Flyweight

Intent: Use Sharing to support large numbers of fine-grained objects efficiently

Consequences: Flyweight is entirely about a cost of storage to cost of compute trade off. It is very expensive computationally to extrinsically store state for objects, but storing the state on every instance is going to cost even more storage • Reducing the number of instances of large objects is the goal of flyweight

ConcreteFlyweight:
Implements the flyweight interface and adds storage for intrinsic state, if any. A ConcreteFlyweight object must be shareable. The state must be independent of the context of the object.

UnsharedConcreteFlyweight:
Not all flyweight subclasses need to be shared. They flyweight interface enables sharing; it doesn't enforce it.

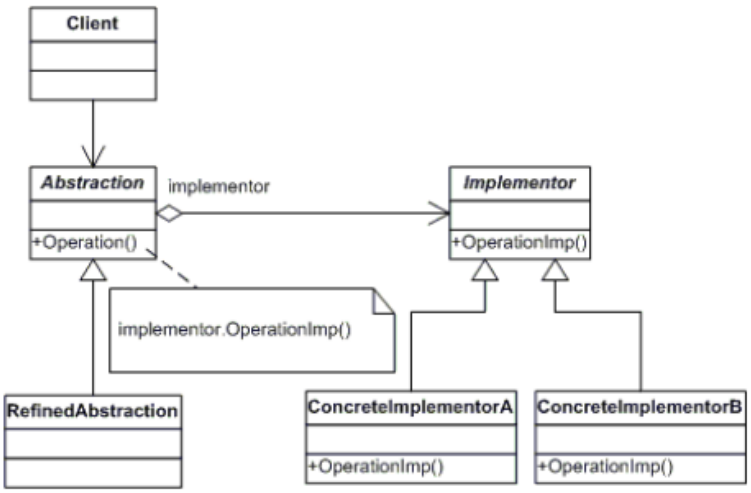


When to use flyweight: lots of objects, states can be made extrinsic, high storage costs

22. Bridge

Intent: Decouple an abstraction from its implementation so that the two can vary independently

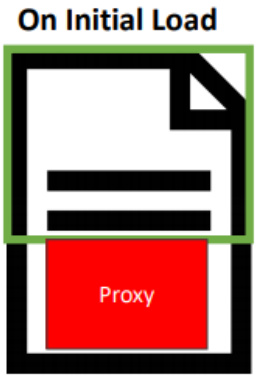
Consequences: • Decoupling of interface and implementation – An implementation is not permanently bound to it's interface. This allows us to even go as far as to switch implementations of an object at runtime • Improved extensibility – You can extend the abstraction and implementer hierarchies independently • Hiding implementation details from clients – You can shield implementation details from clients

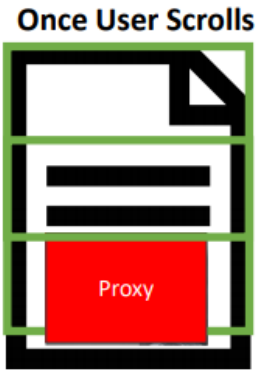


23. Proxy

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

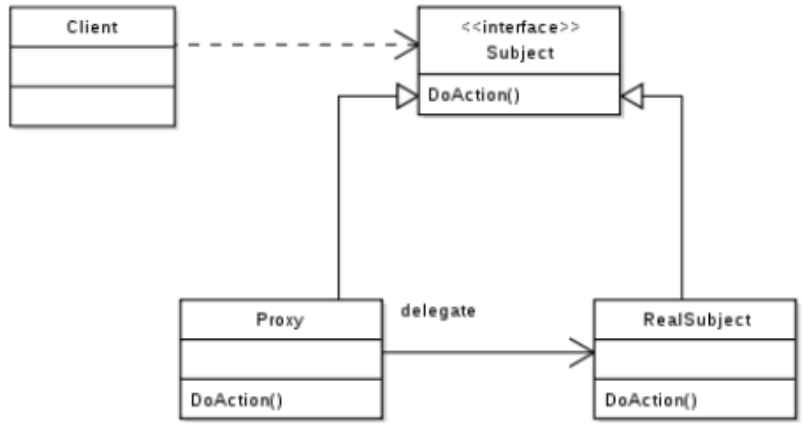
Consequences: There are several forms of proxies • Remote proxy – Can be used to hide the fact that an object resides at a different address space • Virtual proxy – can perform optimizations such as creating objects on demand, when needed • Protection proxy – Allow for additional housekeeping tasks to occur when an object is accessed

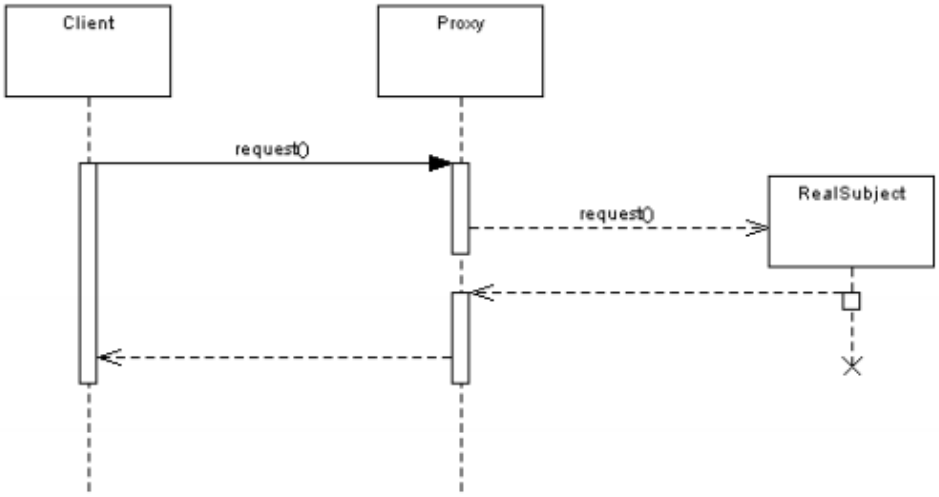




When a user loads a word document, which contains a large image, the document loads a proxy of the image, which will only fully be loaded dynamically once the user scrolls to the point where it is on screen.

If the proxy object was not in place, the rest of the formatting of the document couldn't occur, as the image itself creates constraints on how the text in the document formats.


Client
<<interface>>
Subject
DoAction()
Proxy
delegate
RealSubject
DoAction()
DoAction()


Client
Proxy
request()
request()
RealSubject