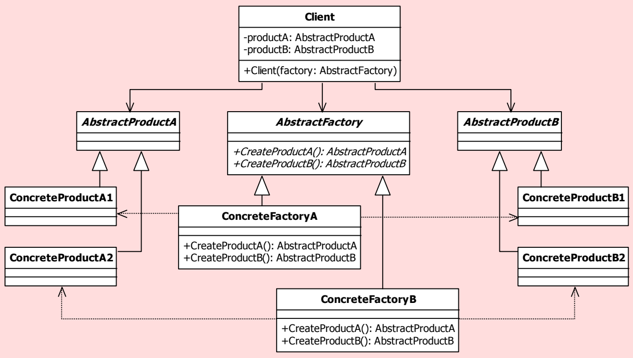
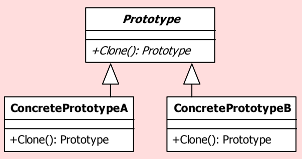
Creational Design Patterns

Abstract Factory


Client
-productA: AbstractProductA
-productB: AbstractProductB
+Client(factory: AbstractFactory)
AbstractProductA
AbstractFactory
+CreateProductA(): AbstractProductA
+CreateProductB(): AbstractProductB
AbstractProductB
ConcreteProductA1
ConcreteFactoryA
+CreateProductA(): AbstractProductA
+CreateProductB(): AbstractProductB
ConcreteProductB1
ConcreteProductA2
ConcreteProductB2
ConcreteFactoryB
+CreateProductA(): AbstractProductA
+CreateProductB(): AbstractProductB

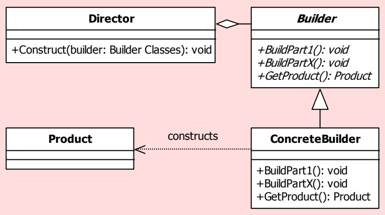
The abstract factory pattern is used to provide a client with a set of related or dependant objects. The "family" of objects created by the factory are determined at run-time.

Prototype



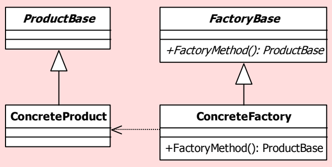
The prototype pattern is used to instantiate a new object by copying all of the properties of an existing object, creating an independent clone. This practise is particularly useful when the construction of a new object is inefficient.

Builder



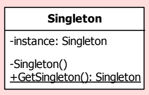
The builder pattern is used to create complex objects with constituent parts that must be created in the same order or using a specific algorithm. An external class controls the construction algorithm.

Factory Method



The factory pattern is used to replace class constructors, abstracting the process of object generation so that the type of object instantiated can be determined at run-time.

Singleton

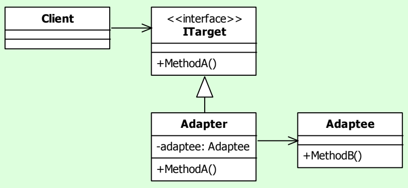


The singleton pattern ensures that only one object of a particular class is ever created. All further references to objects of the singleton class refer to the same underlying instance.

http://www.blackwasp.co.uk

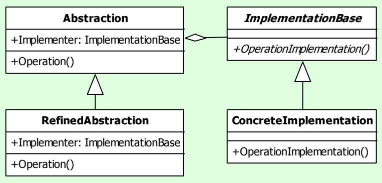
Structural Design Patterns

Adapter



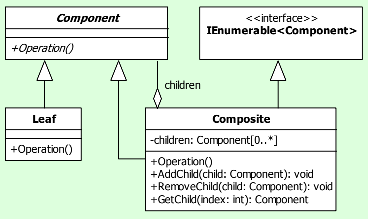
The adapter pattern is used to provide a link between two otherwise incompatible types by wrapping the "adaptee" with a class that supports the interface required by the client.

Bridge


Abstraction
+Implementer: ImplementationBase
+Operation()
ImplementationBase
+OperationImplementation()
RefinedAbstraction
+Implementer: ImplementationBase
+Operation()
ConcreteImplementation
+OperationImplementation()

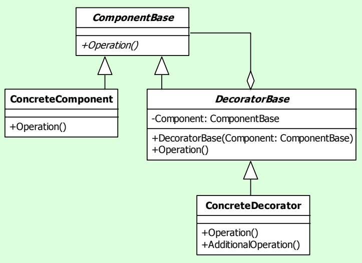
The bridge pattern is used to separate the abstract elements of a class from the implementation details, providing the means to replace the implementation details without modifying the abstraction.

Composite


Component
+Operation()
<<interface>>
IEnumerable<Component>
children
Leaf
+Operation()
Composite
-children: Component[0..*]
+Operation()
+AddChild(child: Component): void
+RemoveChild(child: Component): void
+GetChild(index: int): Component

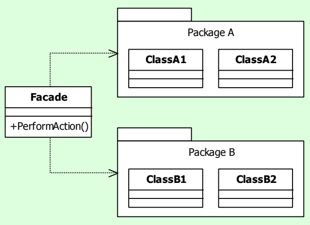
The composite pattern is used to create hierarchical, recursive tree structures of related objects where any element of the structure may be accessed and utilised in a standard manner.

Decorator



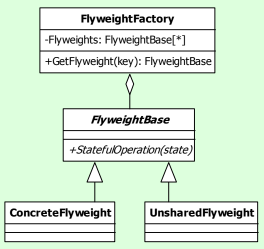
The decorator pattern is used to extend or alter the functionality of objects at run-time by wrapping them in an object of a decorator class. This provides a flexible alternative to using inheritance to modify behaviour.

Facade



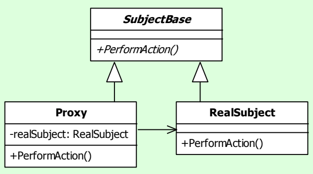
The facade pattern is used to define a simplified interface to a more complex subsystem.

Flyweight


FlyweightFactory
-Flyweights: FlyweightBase[*]
+GetFlyweight(key): FlyweightBase
FlyweightBase
+StatefulOperation(state)
ConcreteFlyweight
UnsharedFlyweight

The flyweight pattern is used to reduce the memory and resource usage for complex models containing many hundreds, thousands or hundreds of thousands of similar objects.

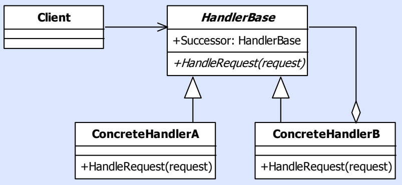
Proxy



The proxy pattern is used to provide a surrogate or placeholder object, which references an underlying object. The proxy provides the same public interface as the underlying subject class, adding a level of indirection by accepting requests from a client object and passing these to the real subject object as necessary.

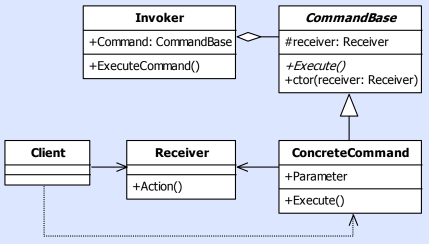
Behavioural Design Patterns

Chain of Responsibility



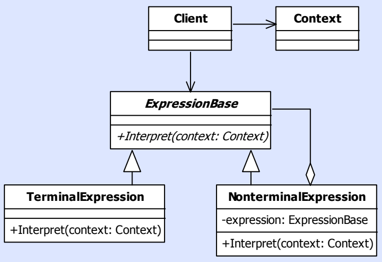
The chain of responsibility pattern is used to process varied requests, each of which may be dealt with by a different handler.

Command



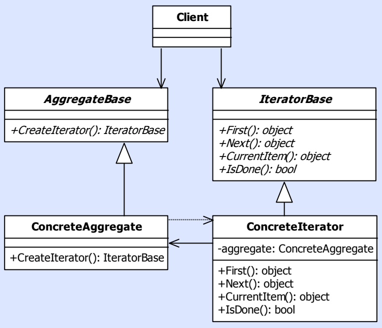
The command pattern is used to express a request, including the call to be made and all of its required parameters, in a command object. The command may then be executed immediately or held for later use.

Interpreter


Client
Context
ExpressionBase
+Interpret(context: Context)
TerminalExpression
+Interpret(context: Context)
NonterminalExpression
-expression: ExpressionBase
+Interpret(context: Context)

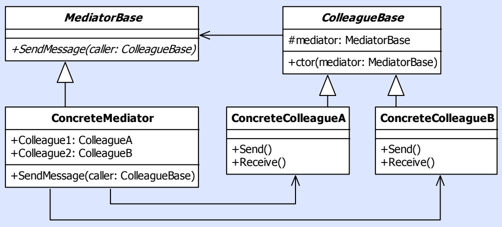
The interpreter pattern is used to define the grammar for instructions that form part of a language or notation, whilst allowing the grammar to be easily extended.

Iterator



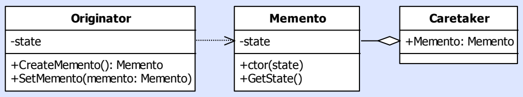
The iterator pattern is used to provide a standard interface for traversing a collection of items in an aggregate object without the need to understand its underlying structure.

Mediator



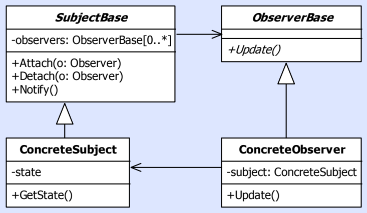
The mediator pattern is used to reduce coupling between classes that communicate with each other. Instead of classes communicating directly, and thus requiring knowledge of their implementation, the classes send messages via a mediator object.

Memento



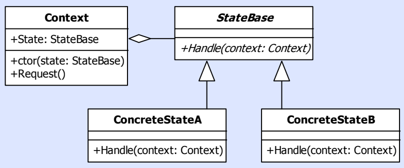
The memento pattern is used to capture the current state of an object and store it in such a manner that it can be restored at a later time without breaking the rules of encapsulation.

Observer



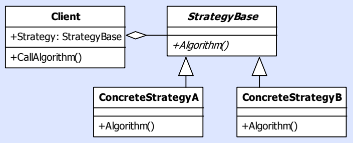
The observer pattern is used to allow an object to publish changes to its state. Other objects subscribe to be immediately notified of any changes.

State



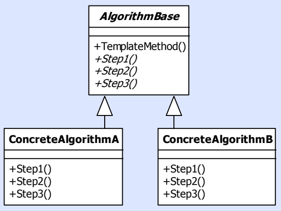
The state pattern is used to alter the behaviour of an object as its internal state changes. The pattern allows the class for an object to apparently change at run-time.

Strategy



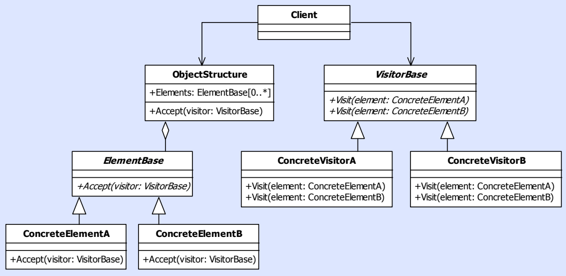
The strategy pattern is used to create an interchangeable family of algorithms from which the required process is chosen at run-time.

Template Method



The template method pattern is used to define the basic steps of an algorithm and allow the implementation of the individual steps to be changed.

Visitor


Client
ObjectStructure
+Elements: ElementBase[0..*]
+Accept(visitor: VisitorBase)
VisitorBase
+Visit(element: ConcreteElementA)
+Visit(element: ConcreteElementB)
ElementBase
+Accept(visitor: VisitorBase)
ConcreteVisitorA
+Visit(element: ConcreteElementA)
+Visit(element: ConcreteElementB)
ConcreteVisitorB
+Visit(element: ConcreteElementA)
+Visit(element: ConcreteElementB)
ConcreteElementA
+Accept(visitor: VisitorBase)
ConcreteElementB
+Accept(visitor: VisitorBase)

The visitor pattern is used to separate a relatively complex set of structured data classes from the functionality that may be performed upon the data that they hold.