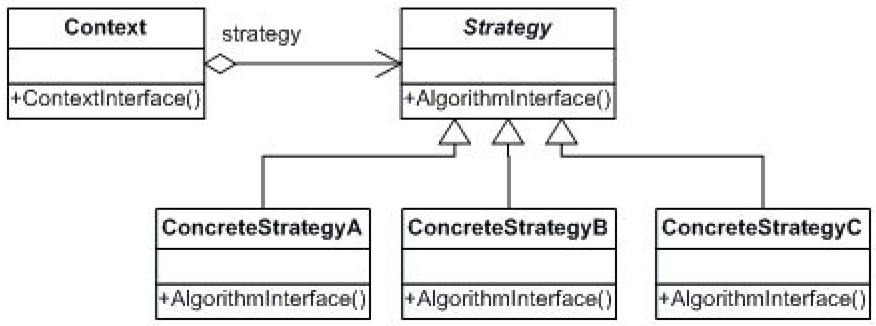
Welcome to Design Patterns: an introduction

Design Principle: Identify the aspects of your application that vary and separate them from what stays the same.
Take what varies and "encapsulate" it so it won't affect the rest of your code. The result? Fewer unintended consequences from code changes and more flexibility in your systems!
Design Principle: Program to an interface, not an implementation.
The point of the last principle is to exploit polymorphism by programming to a supertype so that the actual runtime object isn't locked into the code, so this **interface** may be **an interface** or an **abstract class** because we want to program to a supertype.
With encapsulating(separating) what varies from what stays the same and program those what vary as interfaces you are delegating functionality to these interfaces.
Design Principle: Favor composition over inheritance.
Composition: instead of inheriting a class, you are encapsulating its behavior and reference it in your class. This allows you to change behavior at runtime and also allows you to encapsulate a family of algorithms in their own set of classes.
The Strategy Pattern: defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Remember ⇒ knowing concepts like abstraction, inheritance, and polymorphism do not make you a good object-oriented designer. A design guru thinks about how to create flexible designs that are maintainable and that can cope with change.



Keeping your Objects in the know: the Observer Pattern

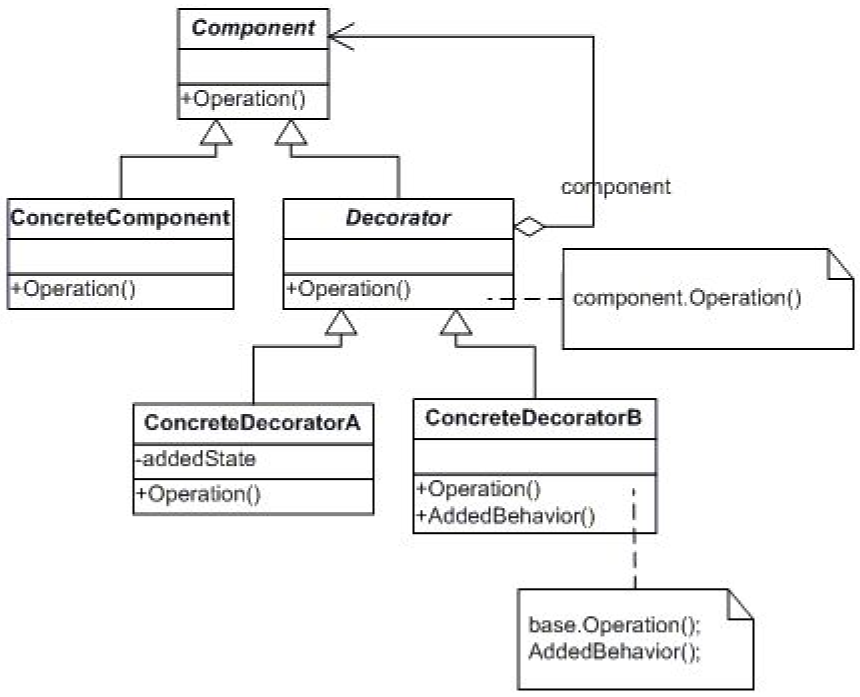
The Observer Pattern: defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically.
When two objects are loosely coupled, they can interact, but have very little knowledge of each other. The Observer Pattern provides an object design where subjects and observers are loosely coupled.
Design Principle: Strive for loosely coupled designs between objects that interact.
Loosely coupled designs allow us to build flexible OO systems that can handle change because they minimize the interdependency between objects.
Subjects, or as we also know them, Observables, update Observers using a common interface.
Observers are loosely coupled in that the Observable knows nothing about them, other than that they implement the Observer Interface.
You can push or pull data from the Observable when using the pattern (pull is considered more “correct”).
Don’t depend on a specific order of notification for your Observers.

Decorating Objects: the Decorator Pattern

With inheritance, the behavior is set statically at compile time. In addition, all classes must inherit the same behavior. If however, I can extend an object’s behavior through composition, so I can do this dynamically at runtime.
OCP: Classes should be closed for modification, but open for extension.
Applying OCP everywhere is wasteful, unnecessary, and can lead to complex, hard to understand the code.
Decorator Pattern: attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

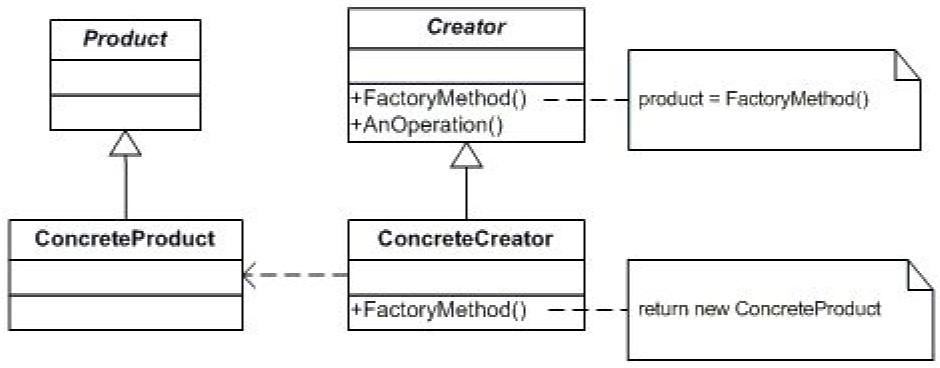
Decorators must have the same type as the objects they are going to decorate. We do this by inheriting the abstract base class to achieve type matching not to get the behavior. The extended behavior comes in through composition of decorators with the base components as well as other decorators.

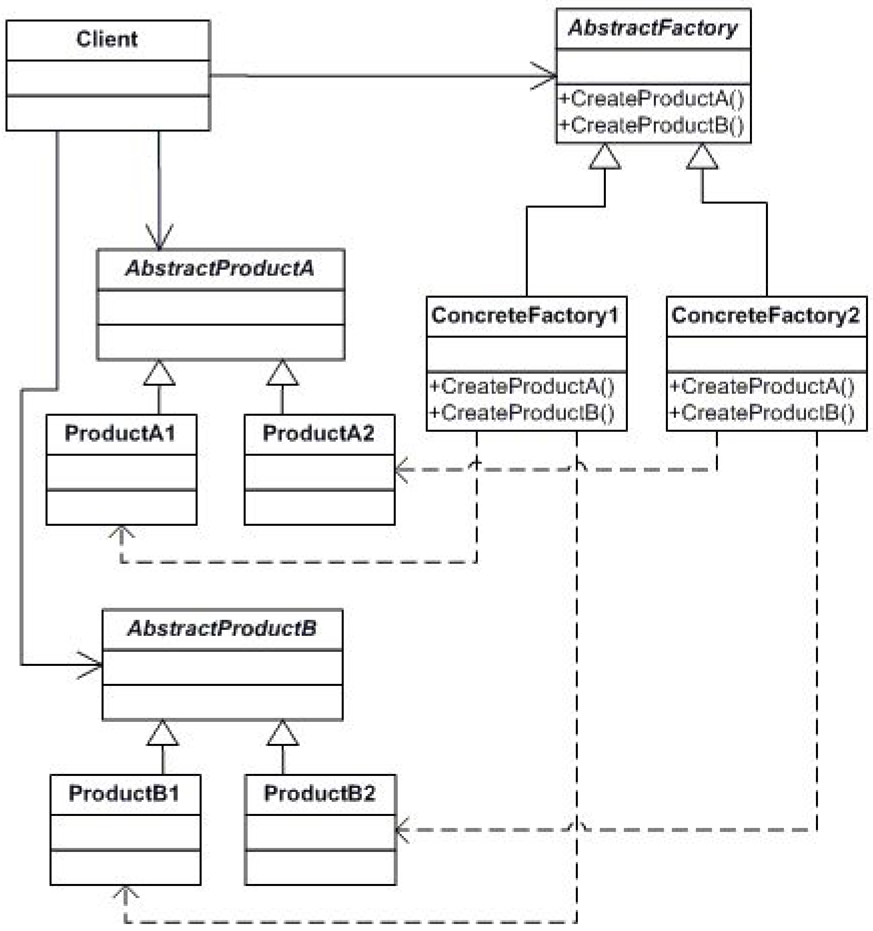
ß Inheritance is one form of extension, but not necessarily the best way to achieve flexibility in our designs.
ß In our designs we should allow the behavior to be extended without the need to modify existing code.
ß Composition and delegation can often be used to add new behaviors at runtime.
ß The Decorator Pattern provides an alternative to subclassing for extending behavior.
ß The Decorator Pattern involves a set of decorator classes that are used to wrap concrete components.
ß Decorator classes mirror the type of the components they decorate. (In fact, they are the same type as the components they decorate, either through inheritance or interface implementation.)
ß Decorators change the behavior of their components by adding new functionality before and/or after (or even in place of) method calls to the component.
ß You can wrap a component with any number of decorators.
ß Decorators are typically transparent to the client of the component; that is unless the client is relying on the component's concrete type.
ß Decorators can result in many small objects in our design, and overuse can be complex.



Baking with OO goodness: the Factory Pattern

When you see “new”, think “concrete”.
A factory method handles object creation and encapsulates it in a subclass. This decouples the client code in the superclass from the object creation code in the subclass.
All factory patterns encapsulate object creation. The Factory Method Pattern encapsulates object creation by letting subclasses decide what objects to create.
The Factory Method Pattern: defines an interface for creating an object, but lets subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
Encapsulate the code that creates objects. When you have code that instantiates concrete classes, this is an area of frequent change.
Design Principle: Depend upon abstractions. Do not depend upon concrete classes.
Factory Method Design Pattern achieves DIP.
The following guidelines can help you avoid OO designs that violate DIP:
ß No variable should hold a reference to a concrete class. → create a factory to get around this.
ß No class should derive from a concrete class. → derive from an interface or an abstract class.
ß No method should override an implemented method of any of its base classes. → methods in base classes should be shared by all subclasses so don’t override them.
The Abstract Factory Pattern: provides an interface for creating families of related or dependent objects without specifying their concrete classes.
Factory Method pattern creates objects through inheritance(you need to extend a class and override a factory method.) where **Abstract Factory** pattern create objects through object composition(you provide an abstract type for creating a family of products. Subclasses of this type define how those products are produced. To use the factory, you instantiate one and pass it into some code that is written against the abstract type).
Abstract Factory pattern interface needs to be updated if new products are added.
ß All factories encapsulate object creation.
ß Simple Factory, while not a bona fide design pattern, is a simple way to decouple your clients from concrete classes.
ß Factory Method relies on inheritance: object creation is delegated to subclasses which implement the factory method to create objects.
ß Abstract Factory relies on object composition: object creation is implemented in methods exposed in the factory interface.
ß All factory patterns promote loose coupling by reducing the dependency of your application on concrete classes.
ß The intent of Factory Method is to allow a class to defer instantiation to its subclasses.
ß The intent of Abstract Factory is to create families of related objects without having to depend on their concrete classes.
ß The Dependency Inversion Principle guides us to avoid dependencies on concrete types and to strive for abstractions.
ß Factories are a powerful technique for coding to abstractions, not concrete classes

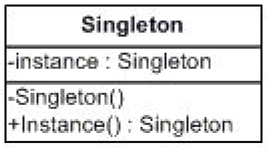




One of a Kind Objects: the Singleton Pattern

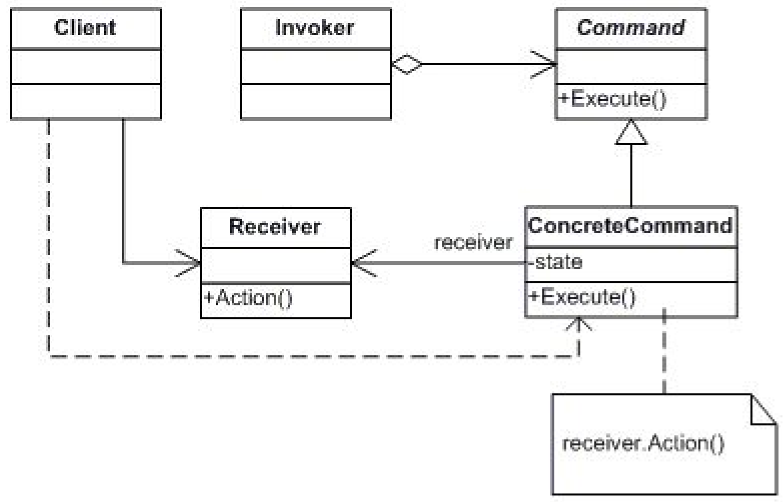
The Singleton Pattern ensures a class has only one instance and provides a global point of access to it.
We're taking a class and letting it manage a single instance of itself. We're also preventing any other class from creating a new instance on its own. To get an instance, you've got to go through the class itself.
ß We're also providing a global access point to the instance: whenever you need an instance,

just query the class and it will hand you back the single instance. We can implement this so that the Singleton is created in a lazy manner, which is especially important for resource-intensive objects.
ß The Singleton Pattern ensures you have at most one instance of a class in your application.
ß The Singleton Pattern also provides a global access point to that instance.
ß Java's implementation of the Singleton Pattern makes use of a private constructor, a static method combined with a static variable.
ß Examine your performance and resource constraints and carefully choose an appropriate Singleton implementation for multithreaded applications (and we should consider all applications multithreaded!).



Encapsulating Invocation: the Command Pattern

The Command Pattern: encapsulates a request as an object, thereby letting you parameterize other objects with different requests, queue or log requests, and support undoable operations.
ß The Command Pattern decouples an object, making a request from the one that knows how to perform it.
ß A Command object is at the center of this decoupling and encapsulates a receiver with an action (or set of actions).
ß An invoker makes a request of a Command object by calling its execute() method, which invokes those actions on the receiver.
ß Invokers can be parameterized with Commands, even dynamically at runtime.
ß Commands may support undo by implementing an undo method that restores the object to its previous state before the execute() method was last called.
ß Macro Commands are a simple extension of Command that allows multiple commands to be invoked. Likewise, Macro Commands can easily support undo().
ß In practice, it is not uncommon for "smart" Command objects to implement the request themselves rather than delegating to a receiver.
ß Commands may also be used to implement logging and transactional systems.

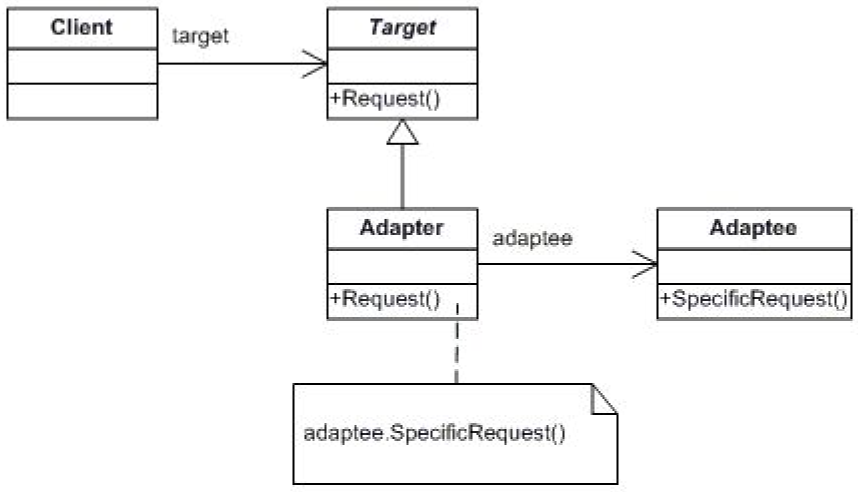


Being Adaptive: the Adapter and Facade Patterns

The Adapter Pattern converts the interface of a class into another interface the clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
Object Adapter uses **composition** to adapt the adaptee where **class adapter** uses **multiple inheritance** to adapt the adaptee which is not possible in C# and Java.

The Facade Pattern provides a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
A facade not only simplifies an interface, it decouples a client from a subsystem of components.
Facades and **adapters** may wrap multiple classes, but a facade's intent is to **simplify**, while an adapter is to **convert** the interface to something different.
Facades don't "**encapsulate**" the subsystem classes; they merely provide a simplified interface to their functionality. The subsystem classes still remain available for direct use by clients that need to use more specific interfaces.
A facade can add functionality in addition to making use of the subsystem.
The Adapter Pattern changes the interface of one or more classes into one interface that a client is expecting. While most textbook examples show the adapter adapting one class, you may need to adapt many classes to provide the interface a client is coded to. Likewise, a Facade may provide a simplified interface to a single class with a very complex interface. The difference between the two is not in terms of how many classes they "wrap," it is in their **intent**.
The intent of the **Adapter Pattern** is to **alter an interface** so that it matches one a client is expecting. The intent of the **Facade Pattern** is to **provide a simplified interface** to a subsystem.
Design Principle: Principle of Least Knowledge - talk only to your immediate friends.
For any object; any method in that object, the principle tells us that we should only invoke methods that belong to:
ß The object itself

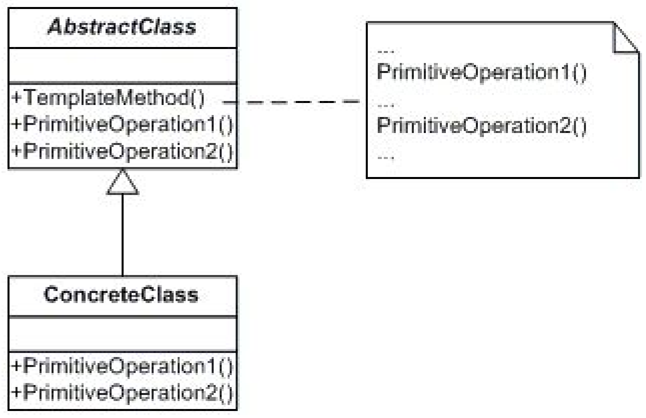
ß Objects passed in as a parameter to the method
ß Any object the method creates or instantiates
ß Any components of the object → components mean objects referenced in this object
----
ß When you need to use an existing class and its interface is not the one you need, use an adapter.
ß When you need to simplify and unify a large interface or complex set of interfaces, use a facade.
ß An adapter changes an interface into one a client expects.
ß A facade decouples a client from a complex subsystem.
ß Implementing an adapter may require little work or a great deal of work depending on the size and complexity of the target interface.
ß Implementing a facade requires that we compose the facade with its subsystem and use delegation to perform the work of the facade.
ß There are two forms of the Adapter Pattern: object and class adapters. Class adapters require multiple inheritance.
ß You can implement more than one facade for a subsystem.
ß An adapter wraps an object to change its interface, a decorator wraps an object to add new behaviors and responsibilities, and a facade "wraps" a set of objects to simplify.



Encapsulating Algorithms: the Template Method Pattern

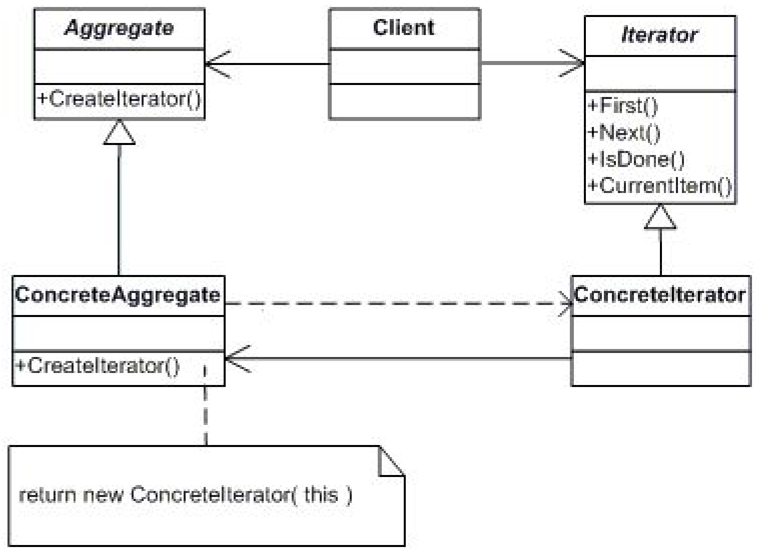
The Template Method Pattern: defines the skeleton of an algorithm in a method, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
The Template Method defines the steps of an algorithm and allows subclasses to provide the implementation for one or more steps.

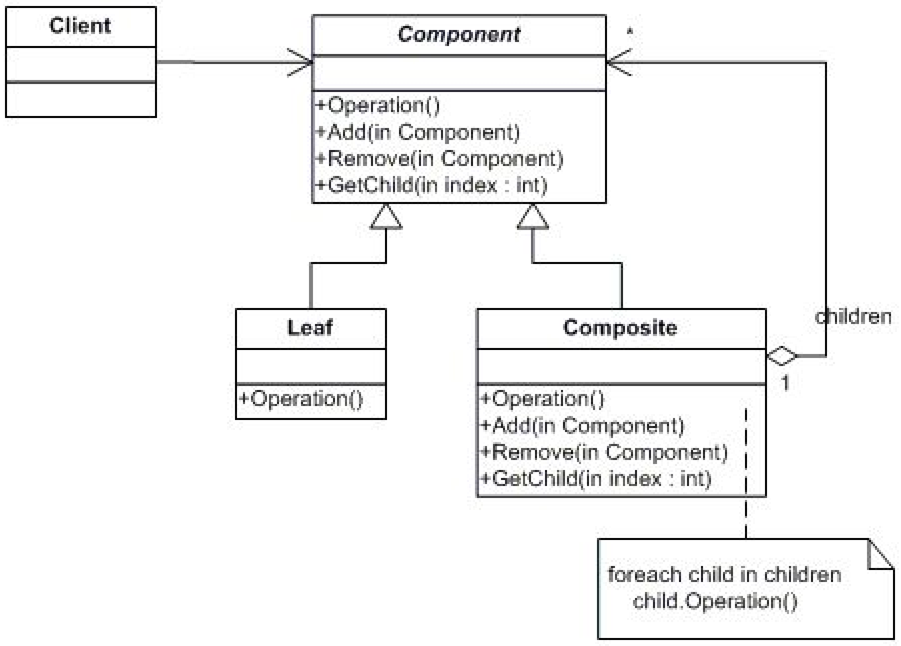
The template method helps us remove code duplication by abstracting the algorithm in the base class.
Use abstract methods when your subclass MUST provide an implementation of the method or step in the algorithm.
Use hooks when that part of the algorithm is optional. With hooks, a subclass may choose to implement that hook, but it doesn't have to.
The Hollywood Principle: Don't call us, we'll call you.
The Hollywood principle is implemented in the template method pattern because the abstract base class calls child classes when needed but child classes can't call the base class.
The Dependency Inversion Principle teaches us to avoid the use of concrete classes and instead work as much as possible with abstractions. The Hollywood Principle is a technique for building frameworks or components so that lower-level components can be hooked into the computation, but without creating dependencies between the lower-level components and the higher-level layers. So, they both have the goal of decoupling, but the Dependency Inversion Principle makes a much stronger and general statement about how to avoid dependencies in design. The Hollywood Principle gives us a technique for creating designs that allow low-level structures to interoperate while preventing other classes from becoming too dependent on them.
ß A "template method" defines the steps of an algorithm, deferring to subclasses for the implementation of those steps.
ß The Template Method Pattern gives us an important technique for code reuse.
ß The template method's abstract class may define concrete methods, abstract methods, and hooks.
ß Abstract methods are implemented by subclasses.
ß Hooks are methods that do nothing or default behavior in the abstract class, but may be overridden in the subclass.
ß To prevent subclasses from changing the algorithm in the template method, declare the template method as final.
ß The Hollywood Principle guides us to put decision-making in high-level modules that can decide how and when to call low-level modules.
ß You'll see lots of uses of the Template Method Pattern in real-world code, but don't expect it all (like any pattern) to be designed "by the book."
ß The Strategy and Template Method Patterns both encapsulate algorithms, one by inheritance and one by composition.
ß The Factory Method is a specialization of Template Method.



Well-managed Collections: the Iterator and Composite Patterns

The Iterator Pattern: provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
The Iterator Pattern allows traversal of the elements of an aggregate without exposing the underlying implementation. It also places the task of traversal on the iterator object, not on the aggregate, which simplifies the aggregate interface and implementation, and places the responsibility where it should be.
The Composite Pattern: allows you to compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
Composite Pattern allows us to build structures of objects in the form of trees that contain both compositions of objects and individual objects as nodes. Using a composite structure, we can apply the same operations over both composites and individual objects. In other words, in most cases we can ignore the differences between compositions of objects and individual objects.
ß An Iterator allows access to an aggregate's elements without exposing its internal structure.
ß An Iterator takes the job of iterating over an aggregate and encapsulates it in another object.
ß When using an Iterator, we relieve the aggregate of the responsibility of supporting operations for traversing its data.
ß An Iterator provides a common interface for traversing the items of an aggregate, allowing you to use polymorphism when writing code that makes use of the items of the aggregate.
ß We should strive to assign only one responsibility to each class.
ß The Composite Pattern provides a structure to hold both individual objects and composites.
ß The Composite Pattern allows clients to treat composites and individual objects uniformly.
ß A Component is any object in a Composite structure. Components may be other composites or leaf nodes.
ß There are many design tradeoffs in implementing Composite. You need to balance transparency and safety with your needs.

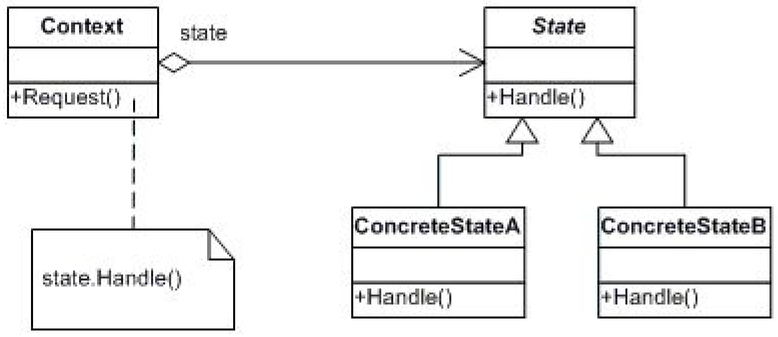




The State of Things: the State Pattern

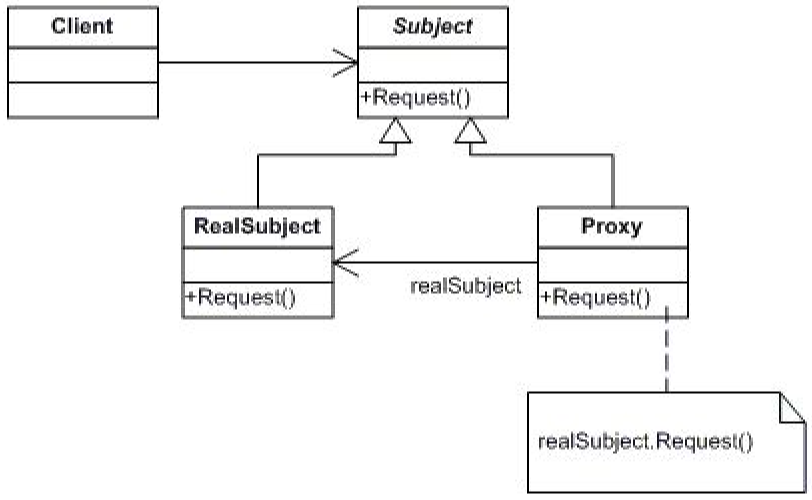
ß The State Pattern allows an object to have many different behaviors that are based on its internal state.
ß Unlike a procedural state machine, the State Pattern represents state as a full-blown class.
ß The Context gets its behavior by delegating to the current state object it is composed with.
ß By encapsulating each state into a class, we localize any changes that will need to be made.
ß The State and Strategy Patterns have the same class diagram, but they differ in intent.
ß Strategy Pattern typically configures Context classes with a behavior or algorithm.

ß State Pattern allows a Context to change its behavior as the state of the Context changes.
ß State transitions can be controlled by the State classes or by the Context classes.
ß Using the State Pattern will typically result in a greater number of classes in your design.
ß State classes may be shared among Context instances.



Controlling Object Access: the Proxy Pattern

ß The Proxy Pattern provides a representative for another object in order to control the client's access to it. There are a number of ways it can manage that access.
ß A Remote Proxy manages interaction between a client and a remote object.
ß A Virtual Proxy controls access to an object that is expensive to instantiate.
ß A Protection Proxy controls access to the methods of an object based on the caller.
ß Many other variants of the Proxy Pattern exist including caching proxies, synchronization proxies, firewall proxies, copy-on-write proxies, and so on.
ß Proxy is structurally similar to Decorator, but the two differ in their purpose.
ß The Decorator Pattern adds behavior to an object, while a Proxy controls access.
ß Java's built-in support for Proxy can build a dynamic proxy class on demand and dispatch all calls on it to a handler of your choosing.
ß Like any wrapper, proxies will increase the number of classes and objects in your designs.



Patterns of Patterns: Compound Patterns

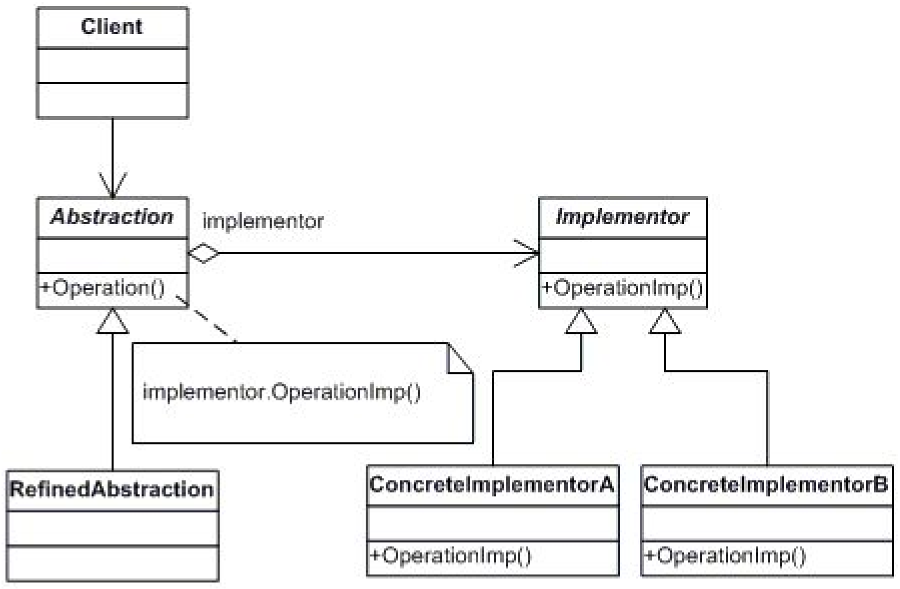
Patterns are often used together and combined within the same design solution.
A compound pattern combines two or more patterns into a solution that solves a recurring or general problem.
ß The Model View Controller Pattern (MVC) is a compound pattern consisting of the Observer, Strategy and Composite patterns.
ß The model makes use of the Observer Pattern so that it can keep observers updated yet stay decoupled from them.
ß The controller is the strategy for the view. The view can use different implementations of the controller to get different behavior.
ß The view uses the Composite Pattern to implement the user interface, which usually consists of nested components like panels, frames and buttons.
ß These patterns work together to decouple the three players in the MVC model, which keeps designs clear and flexible.
ß The Adapter Pattern can be used to adapt a new model to an existing view and controller.
ß Model 2 is an adaptation of MVC for web applications.
ß In Model 2, the controller is implemented as a servlet and JSP & HTML implement the view.

Other Design Patterns definition and class diagram:

Bridge: Use the Bridge Pattern to vary not only your implementations, but also your abstractions.

Definition

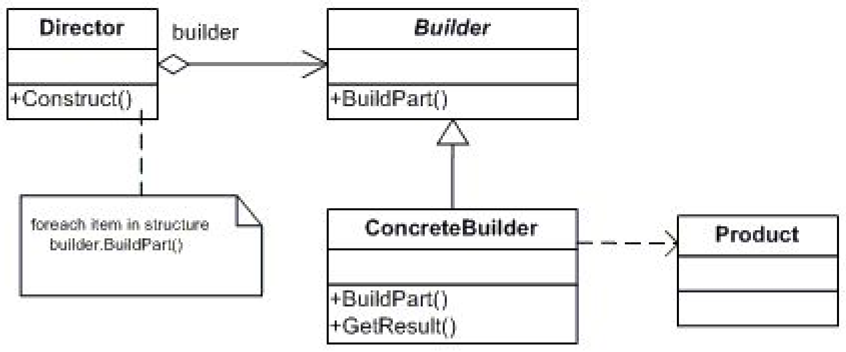
Decouple an abstraction from its implementation so that the two can vary independently.



Builder: Use the Builder Pattern to encapsulate the construction of a product and allow it to be constructed in steps.

Definition

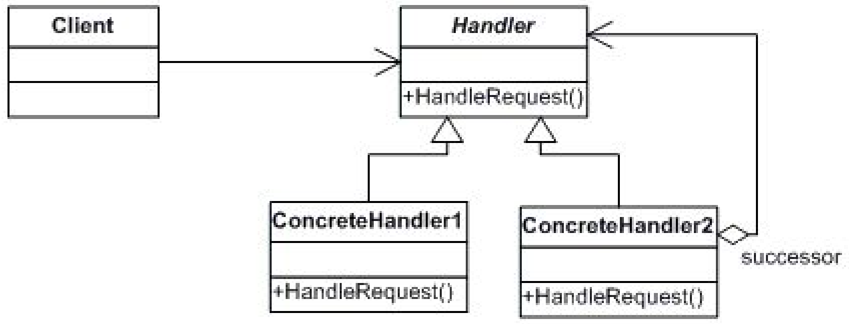
Separate the construction of a complex object from its representation so that the same construction process can create different representations.



Chain of Responsibility: Use the Chain of Responsibility Pattern when you want to give more than one object a chance to handle a request.

Definition

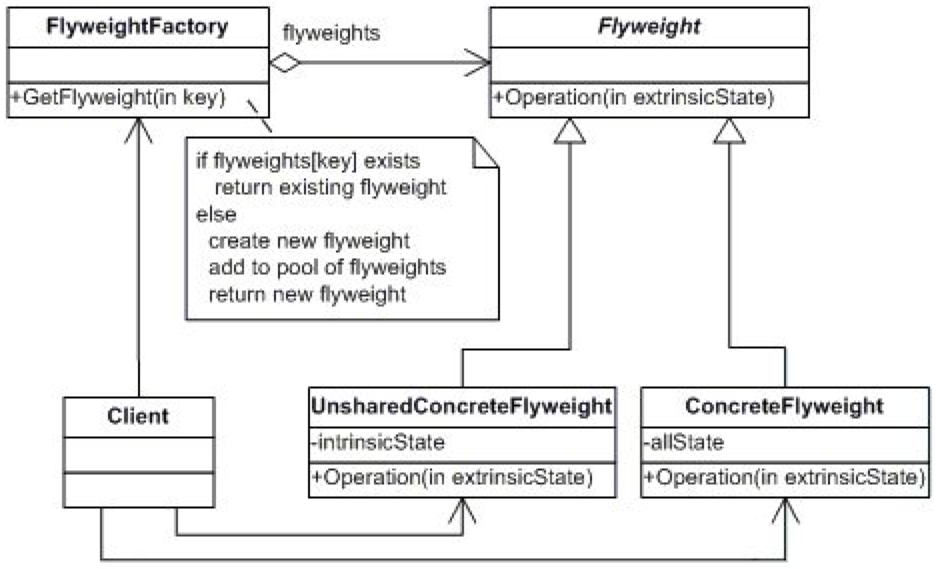
Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.



Flyweight: Use the Flyweight Pattern when one instance of a class can be used to provide many "virtual instances."

Definition

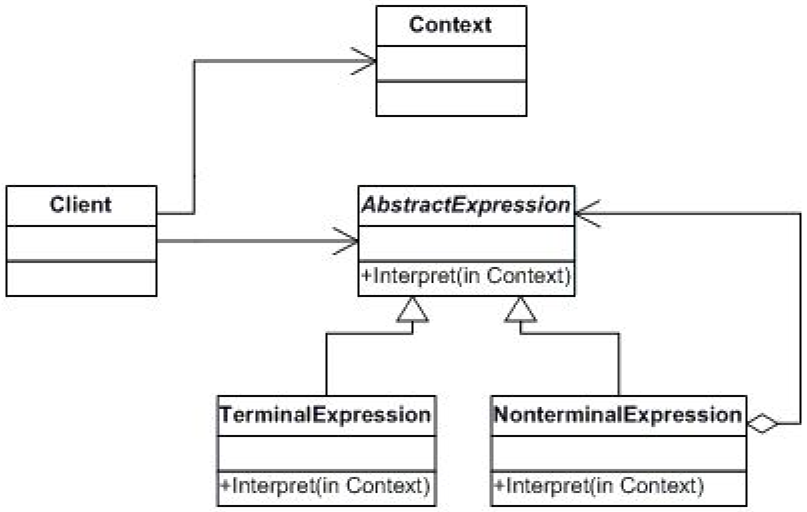
Use sharing to support large numbers of fine-grained objects efficiently.


FlyweightFactory
+GetFlyweight(in key)
flyweights
Flyweight
+Operation(in extrinsicState)
if flyweights[key] exists
return existing flyweight
else
create new flyweight
add to pool of flyweights
return new flyweight
Client
UnsharedConcreteFlyweight
-intrinsicState
+Operation(in extrinsicState)
ConcreteFlyweight
-allState
+Operation(in extrinsicState)

Interpreter: Use the Interpreter Pattern to build an interpreter for a language.

Definition

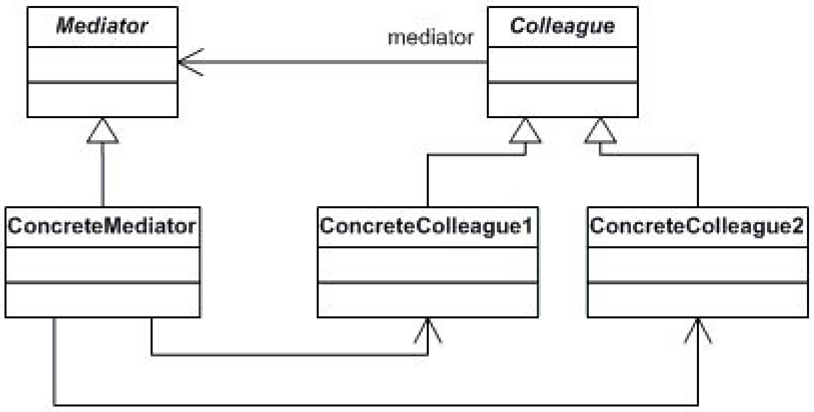
Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.



Mediator: Use the Mediator Pattern to centralize complex communications and control between related objects.

Definition

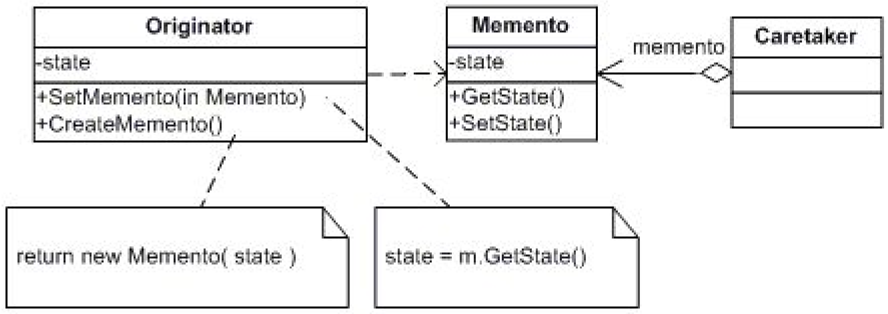
Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.



Memento: Use the Memento Pattern when you need to be able to return an object to one of its previous states; for instance, if your user requests an "undo."

Definition

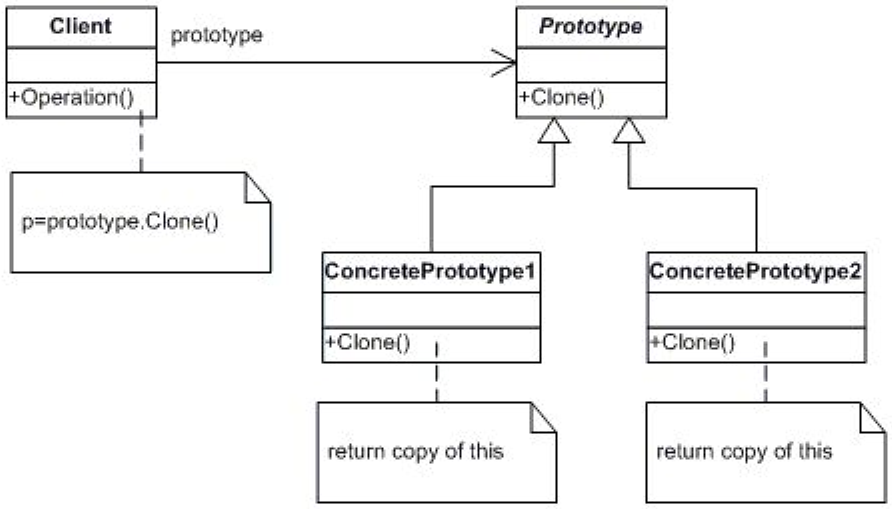
Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.



Prototype: Use the Prototype Pattern when creating an instance of a given class is either expensive or complicated.

Definition

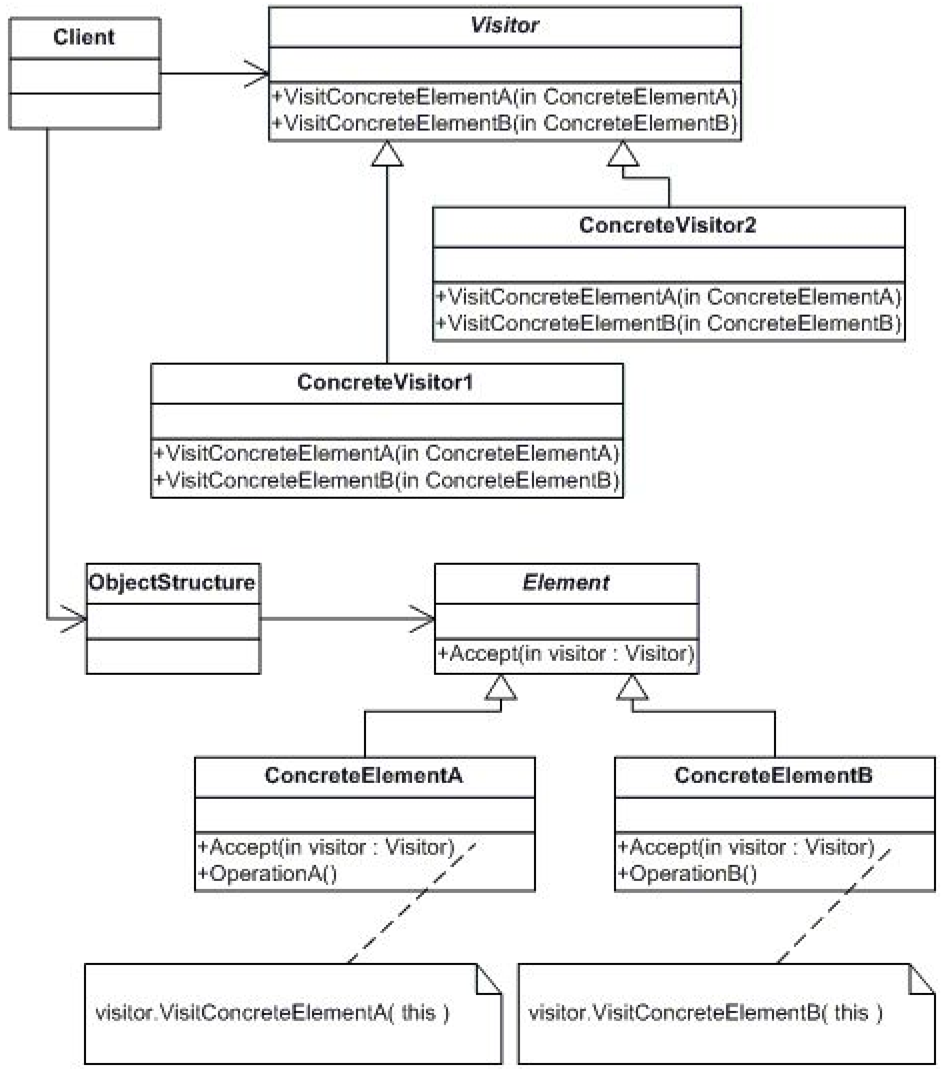
Specify the kind of objects to create using a prototypical instance, and create new objects by copying this prototype.



Visitor: Use the Visitor Pattern when you want to add capabilities to a composite of objects and encapsulation is not important.

Definition

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.


Client
Visitor
+VisitConcreteElementA(in ConcreteElementA)
+VisitConcreteElementB(in ConcreteElementB)
ConcreteVisitor2
+VisitConcreteElementA(in ConcreteElementA)
+VisitConcreteElementB(in ConcreteElementB)
ConcreteVisitor1
+VisitConcreteElementA(in ConcreteElementA)
+VisitConcreteElementB(in ConcreteElementB)
ObjectStructure
Element
+Accept(in visitor : Visitor)
ConcreteElementA
+Accept(in visitor : Visitor)
+OperationA()
ConcreteElementB
+Accept(in visitor : Visitor)
+OperationB()
visitor.VisitConcreteElementA(this)
visitor.VisitConcreteElementB(this)
