

Encapsulation

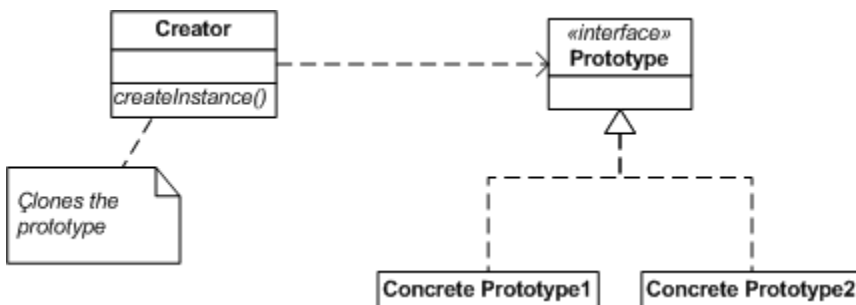
Encapsulation conceals the functional details of a class from objects that send messages to it.

Intro

<p>Encapsulation</p>	<p>Encapsulation conceals the functional details of a class from objects that send messages to it.</p>
<p>Law of Demeter</p>	<p>A method should only use:</p> <ul style="list-style-type: none"> • Instance fields • Method Parameters • Objects constructed with new
<p>5 C's of Class Design</p>	<ul style="list-style-type: none"> • Cohesion Class is a single concept. Operations fit together to support that concept • Completeness Class should support all operations of the concept • Convenience Interface might be complete, but must be convenient • Clarity Interface of a class should not be confusing. • Consistency Operations should be consistent with respect to names, parameters, return values and behavior.
<p>Liskov Principle</p>	<p>You can use a subclass object whenever a superclass object is expected.</p>

Patterns

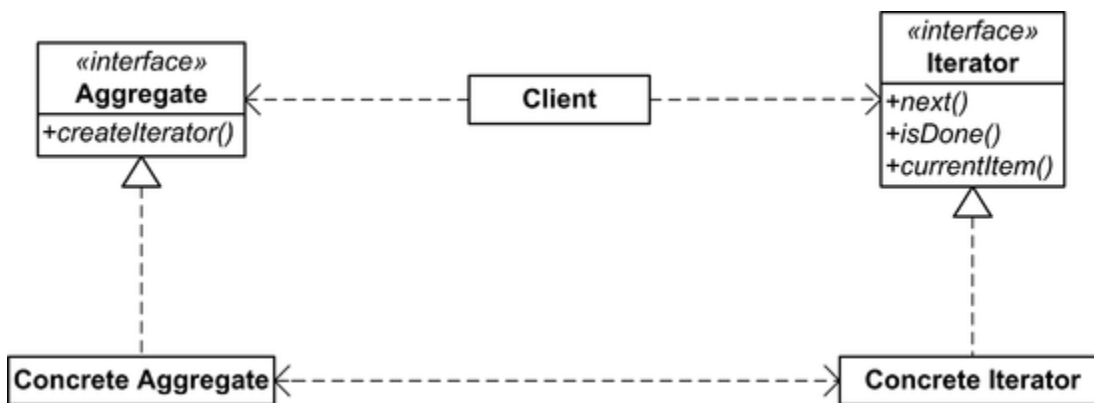
Prototype



The **Prototype** pattern teaches how a system can instantiate classes that are not known when the system is built.

Context	Solution
<ol style="list-style-type: none"> 1. A system needs to create several kinds of objects whose classes are not known when the system is built. 2. You do not want to require a separate class for each kind of object. 3. You want to avoid a separate hierarchy of classes whose responsibility it is to create the objects. 	<ol style="list-style-type: none"> 1. Define a prototype interface that is common to all created objects. 2. Supply a prototype object for each kind of object that the system creates. 3. Clone the prototype object whenever a new object of the given kind is required.

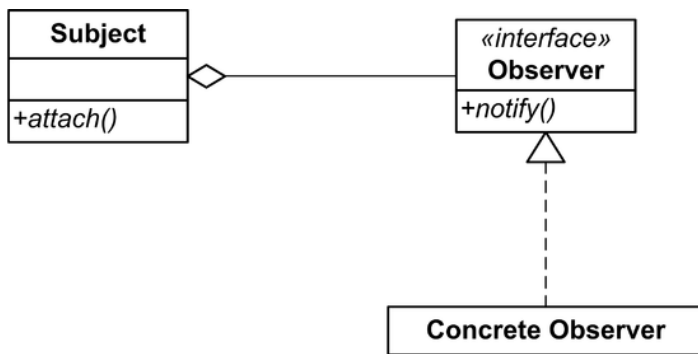
Iterator



The **iterator** pattern teaches how to access elements of an aggregate object.

Context	Solution
<ol style="list-style-type: none"> 1. An object (called aggregate) contains other objects (called elements) 2. Clients (methods using aggregate) need access to elements. 3. The aggregate should not expose the internal structure 4. There may be multiple clients that need simultaneous access. 	<ol style="list-style-type: none"> 1. Define an iterator class that fetches one element at a time. 2. Each iterator object needs to keep track of the position of the next element to fetch. 3. If there are several variations of the aggregate and the iterator classes, it is best if they implement common interface types. Then the client only needs to know the interface types, not the concrete classes.

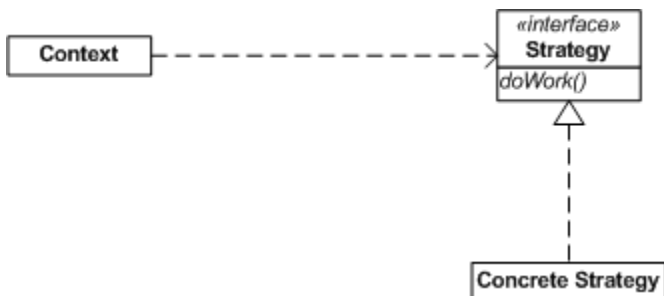
Observer



The **Observer** pattern teaches how an object can tell other objects about events.

Context	Solution
<ol style="list-style-type: none"> 1. An object (called subject) is the source of events. 2. One or more objects (called observers) want to know when an event occurs. 	<ol style="list-style-type: none"> 1. Define an observer interface type. Observer classes must implement this interface type. 2. The subject maintains a collection of observer objects 3. The subject class supplies methods for attaching observers. 4. Whenever an event occurs, the subject notifies all observers.

Strategy

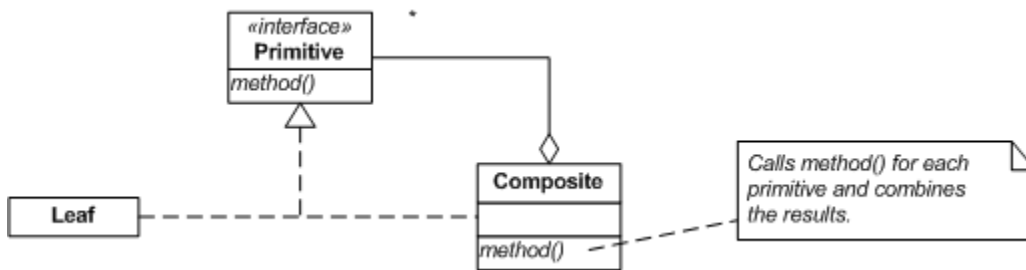


The **Strategy** pattern teaches how to supply variants of an algorithm.

Context	Solution
<ol style="list-style-type: none"> 1. A Class (called context) can benefit from different variations of an algorithm. 2. Clients of the context class sometimes want to supply custom versions of the algorithm 	<ol style="list-style-type: none"> 1. Define an interface type that is an abstraction for the algorithm. We'll call this interface type the strategy. 2. Concrete strategy classes implement the strategy interface type. Each strategy class implements a version of the algorithm. 3. The clients supplies a concrete strategy object to the context class.

- Whenever the algorithm needs to be executed, the context class calls the appropriate methods of the strategy object.

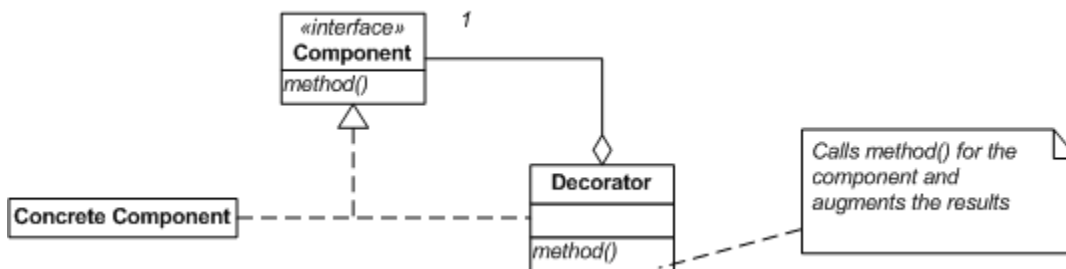
Composite



The **Composite** pattern teaches how to combine several objects into an object that has the same behavior as its parts.

Context	Solution
<ol style="list-style-type: none"> Primitive objects can be combined into composite objects. Clients treat a composite object as a primitive object 	<ol style="list-style-type: none"> Define an interface type that is an abstraction for the primitive objects. A composite object contains a collection of primitive objects. Both primitive classes and composite classes implement that interface type.

Decorator



The **Decorator** pattern teaches how to form a class that adds functionality to another class while keeping its interface.

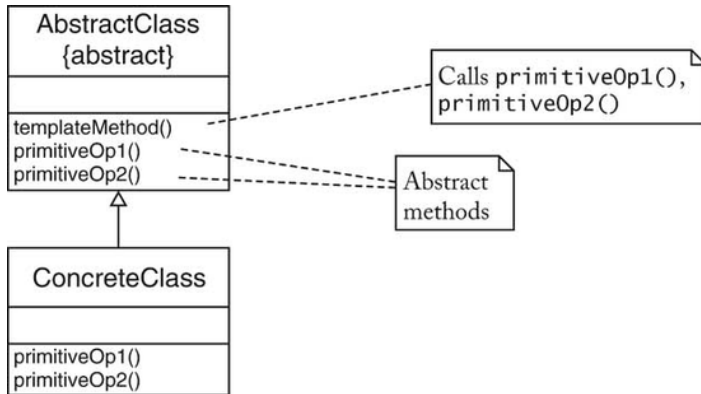
A **decorator** enhances the behavior of a **single** component, whereas a **composite** collects **multiple** components.

Context	Solution
<ol style="list-style-type: none"> You want to enhance the behavior of a class (called component class). A decorated component can be 	<ol style="list-style-type: none"> Define an interface type that is an abstraction for the component. Concrete component classes implement this interface type. Decorator classes also implement this interface type. A decorator object manages the component object that it decorates.

- used in the same way as a plain component.
- The component class does not want to take on the responsibility of the decoration.
 - There may be an open-ended set of possible decorations.

- When implementing a method from a component interface type, the decorator class applies the method to the decorated component and combines the result with the effect of the decoration.

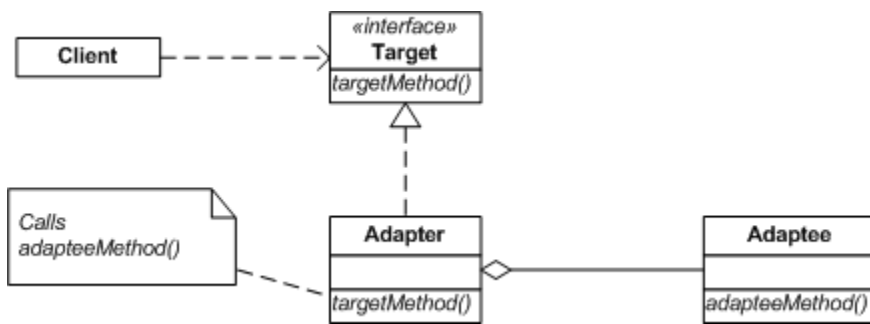
Template Method



The **Template Method** pattern teaches how to supply an algorithm for multiple types, provided that the sequence of steps does not depend on the type.

Context	Solution
<ol style="list-style-type: none"> An algorithm is applicable for multiple types. The algorithm can be broken down into primitive operations. The primitive operations can be different for each type. The order of the primitive operations does not depend on the type. 	<ol style="list-style-type: none"> Define superclass with: <ol style="list-style-type: none"> one method for the algorithm abstract methods for the primitive operations. Implement the algorithm to call the primitive operations in the appropriate order. Do not define the primitive operations in the superclass, or define them to have appropriate default behavior. Each subclass defines the primitive operations (but not the algorithm).

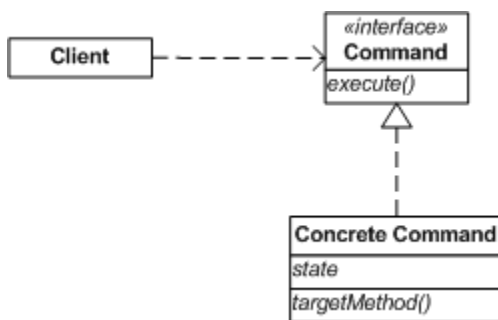
Adapter



The **Adapter** pattern teaches how to use a class in a context that requires a different interface.

Context	Solution
<ol style="list-style-type: none"> 1. You want to use an existing class without modifying it. Class <i>adaptee</i> 2. The context in which you want to use the class requires conformance to a <i>target</i> interface that is different from that of the <i>adaptee</i>. 3. The <i>target</i> interface and the <i>adaptee</i> interface are conceptually related. 	<ol style="list-style-type: none"> 1. Define an adapter class that implements the target interface 2. The adapter class holds a reference to the adaptee. It translates target methods to adaptee methods. 3. The client wraps the adaptee into an adapter class object.

Command

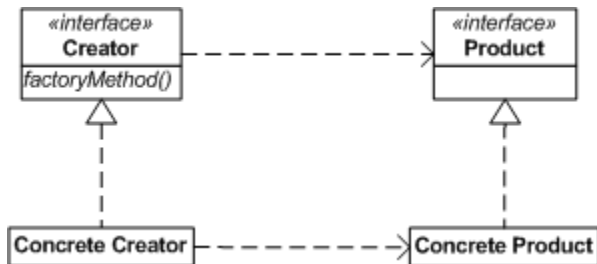


The **Command** pattern teaches how to implement commands as objects whenever a command has both behavior and state.

Context	Solution
<ol style="list-style-type: none"> 1. You want to implement commands that behave like objects, either because you need to store additional information with commands, or because you want to collect commands. 	<ol style="list-style-type: none"> 1. Define a command interface type with a method to execute the command 2. Supply methods in the command interface type to manipulate the state of command objects.

3. Each concrete command class implements the command interface type.
4. To invoke the command, call the **execute** method.

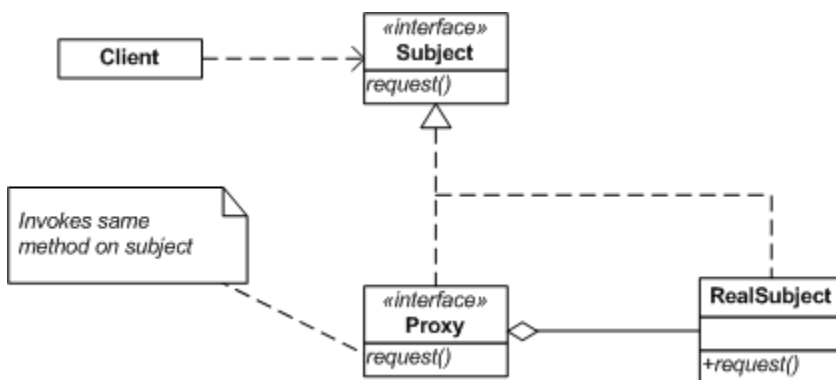
Factory Method



The **Factory Method** pattern teaches how to supply a method that can be overridden to create objects of varying types.

Context	Solution
<ol style="list-style-type: none"> 1. A type (creator) creates objects of another type (product), 2. Subclasses of the creator type need to create different kinds of product objects. 3. Clients do not need to know the exact type of product objects. 	<ol style="list-style-type: none"> 1. Define a creator type that expresses the commonality of all creators. 2. Define a product type that expresses the commonality of all products. 3. Define a method, called the factory method, in the creator type. The <i>factory method</i> yields a product object. 4. Each concrete creator class implements the <i>factory method</i> so that it returns an object of a concrete product class.

Proxy



The **Proxy** pattern teaches how an object can be a placeholder for another object,

Context	Solution
---------	----------

1. A class (**real subject**) provides a service that is specified by an interface type (**subject**) type.
2. There is a need to modify the service in order to make it more versatile
3. Neither the client nor the real subject should be affected by the modification.

1. Define a proxy class that implements the subject interface type. The proxy holds a reference to the real subject, or otherwise knows how to locate it.
2. The client uses a proxy object.
3. Each proxy method invokes the same method on the real subject and provides the necessary modifications.

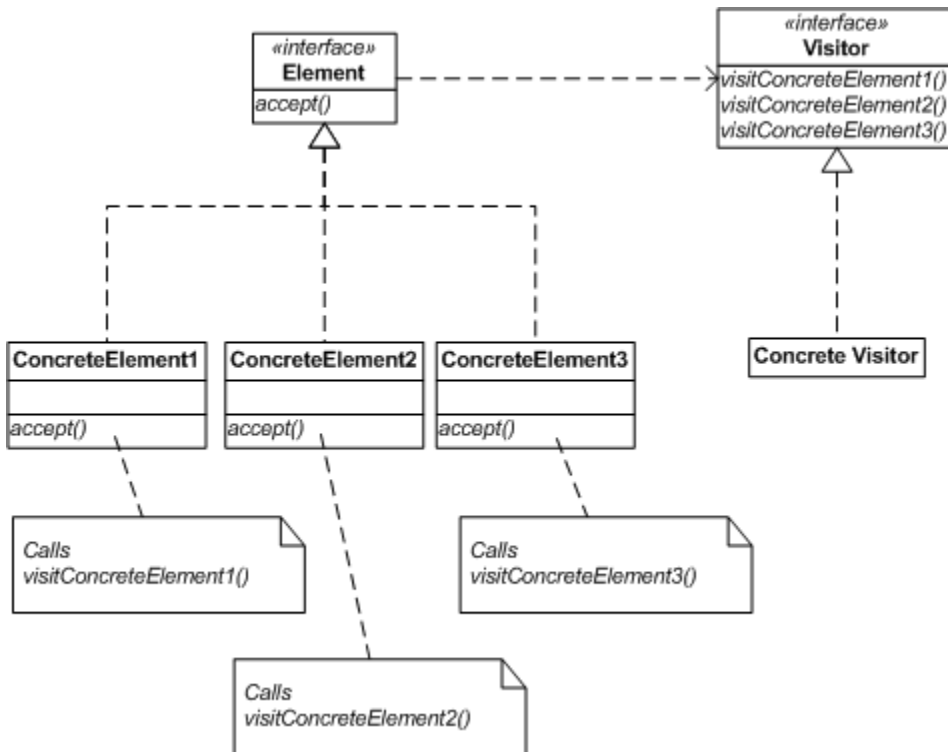
Singleton

A **singleton** class has exactly one instance. The pattern teaches how to implement a class that has exactly one instance.

(Has a **private** constructor, and a static method to get the instance of the class.)

Context	Solution
<ol style="list-style-type: none"> 1. All clients need to access a single shared instance of a class. 2. You want to ensure that no additional instances can be created accidentally. 	<ol style="list-style-type: none"> 1. Define a class with a private constructor. 2. The class constructs a single instance of itself. 3. Supply a static method that returns a reference to the single instance.

Visitor



The **Visitor** pattern teaches how to support an open-ended set of operations on an object structure with a fixed set of element types.

Context	Solution
<ol style="list-style-type: none">1. An object structure contains element classes of multiple types, and you want to carry out operations that depend on the object types.2. The set of operations should be extensible over time.3. The set of element classes is fixed.	<ol style="list-style-type: none">1. Define a visitor interface type that has methods for visiting elements of each of the given types.2. Each element class defines an <code>accept</code> method that invokes the matching element visitation method on the visitor parameter.3. To implement an operation, define a class that implements the visitor interface type and supplies the operation's action for each element type.