Design Patterns

by

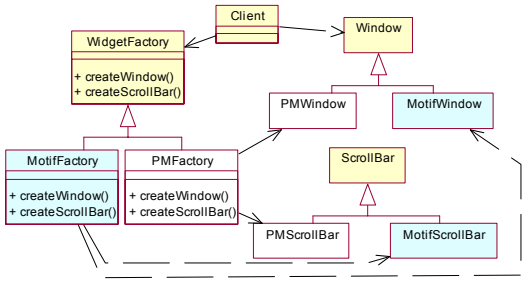
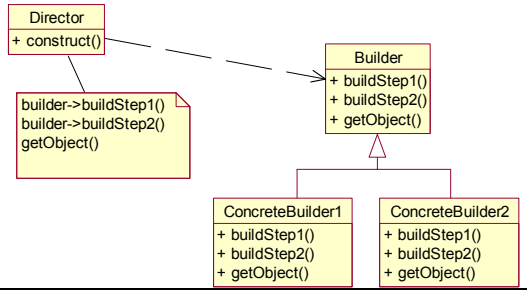
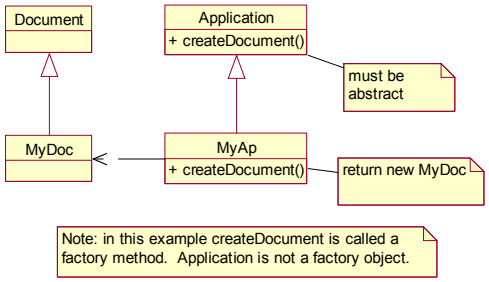
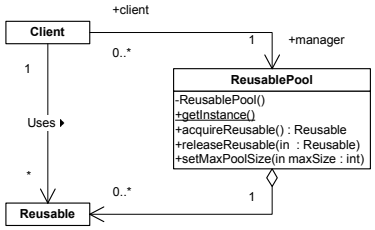
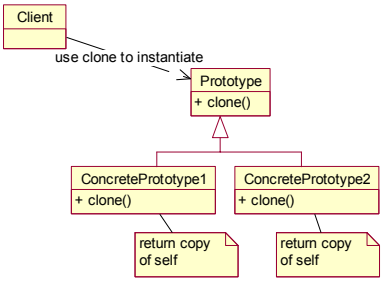
Vijay Nathani

Design Pattern Matrix

CREATIONAL PATTERNS	
	Notes on the patterns
Abstract Factory	**Indicators in analysis:** Different cases exist that require different implementations of sets of rules. **Indicators in design:** Many polymorphic structures exist that are used in pre-defined combinations. These combinations are defined by there being particular cases to implement or different needs of client objects. **Indication pattern is not being used when it should be:** A variable is used in several places to determine which object to instantiate. **Relationships involved:** The Abstract Factory object is responsible for coordinating the family of objects that the client object needs. The client object has the responsibility for using the objects. **Principle manifested:** Encapsulate construction.
Builder	**Indicators in analysis:** Several different kinds of complex objects can be built with the same overall build process, but where there is variation in the individual construction steps. **Indicators in design:** You want to hide the implementation of instantiating complex object, or you want to bring together all of the rules for instantiating complex objects.
Factory Method	**Indicators in analysis:** There are different commonalities whose implementations are coordinated with each other. **Indicators in design:** A class needs to instantiate a derivation of another class, but doesn't know which one. Factory method allows a derived class to make this decision. **Field notes:** The Factory method is often used with frameworks. It is also used when the different implementations of one class hierarchy requires a specific implementation of another class hierarchy. Note that a factory method pattern is not simply a method that serves as a factory. The pattern specifically involves the case where the factory is varied polymorphically. Factory Method is also very useful when unit testing with Mock Objects. **Principle manifested:** Encapsulate the relationship between class hierarchies.
Object Pool	**Indicators in analysis:** There are rules about how to manage a set of objects. These can relate to the number, the lifespan, handling error conditions, load balancing and more. **Indicators in design:** There are rules about creating and managing your objects. **Field notes:** The object pool is a great way to defer the management of the lifespan of your objects. Use some trivial to implement rule, but encapsulate it in the object pool. Once you have the system working and you better understand the rules for object creation and management, implement these rules in the encapsulated methods.
Prototype	**Indicators in analysis:** There are prototypical instances of things. **Indicators in design:** When objects being instantiated need to look like a copy of a particular object. Allows for dynamically specifying what our instantiated objects look like.
Singleton	**Indicators in analysis:** There exists only one entity of something in the problem domain that is used by several different things. **Indicators in design:** Several different client objects need to refer to the same thing and we want to make sure we don't have more than one of them. You only want to have one of an object but there is no higher object controlling the instantiation of the object in questions. **Field notes:** You can get much the same function as Singletons with static methods, however using static methods eliminates the possibility of handing future change through polymorphism, and also prevents the object from being passed by reference, serialized, remoted, etc… in general, statics are to be avoided if possible. Also, keep in mind that a stateful Singleton is essentially a global, and thus can potentially create coupling from any part of your system to any other part, so they shold be used with care. Stateless Singletons do not have this problem.

Design Pattern Matrix

CREATIONAL PATTERNS

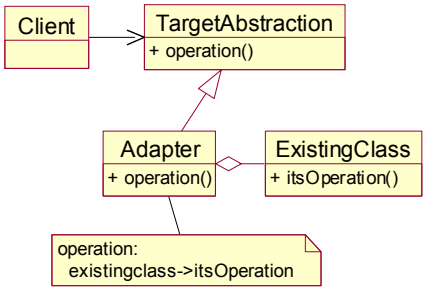
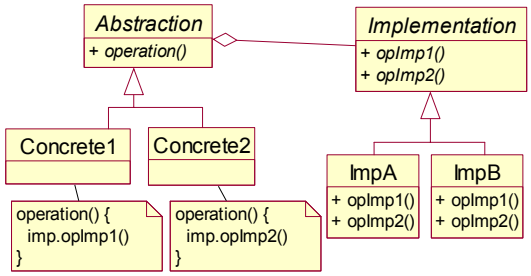
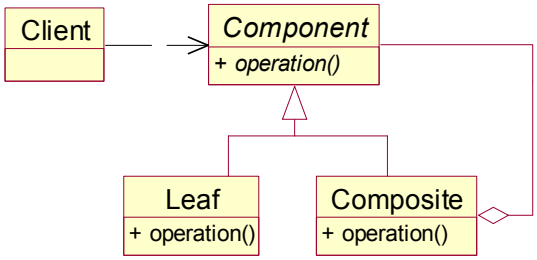
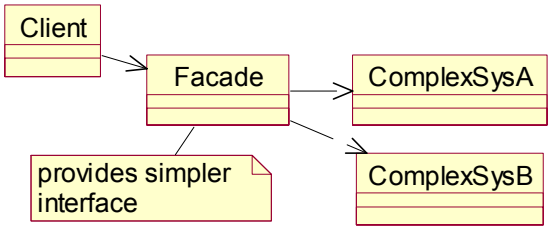
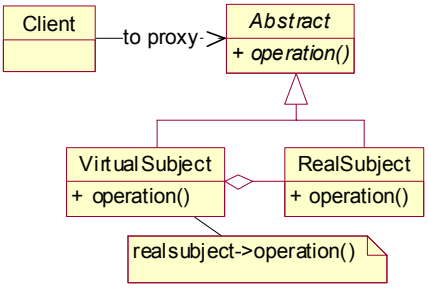
How it is implemented + what it encapsulates	Class Diagram/Implementation	
Define an abstract class that specifies which objects are to be made. Then implement one concrete class for each family. Tables or files can also be used to essentially accomplish the same thing. Names of the desired classes can be kept in a database and then switches or dynamic class loading can be used to instantiate the correct objects. **Encapsulates:** The rules about what our families of objects are – that is, which ones go together.		**Abstract Factory**
Create a factory object that contains several methods. Each method is called separately and performs a necessary step in the building process. When the client object is through, it calls a method to get the constructed object returned to it. Derive classes from the builder object to specialize steps. **Encapsulates:** Rules for constructing complex objects.		**Builder**
Have a method in the abstract class that is abstract (pure virtual). The abstract class's code will refer to this method when it needs to instantiate a contained object. Note, however, that it doesn't know which one it needs. That is why all classes derived from this one must implement this method with the appropriate *new* command to instantiate the proper object. **Encapsulates:** How two inheritance hierarchies are related to each other.		**Factory Method**
Create a class ("ReusablePool") that encapsulates all of the rules for creating and managing the pooled objects. This class has only one instantiation (use Singleton to enforce this). When a new pooled object is needed, the ReusablePool is asked to provide it. It also is told when the object is no longer needed so it can be returned to the shared pool. **Encapsulates:** The rules for creating and managing a set of objects.		**Object Pool**
Set up concrete classes of the class needing to be cloned. Each concrete class will construct itself to the appropriate value (optionally based on input parameters). When a new object is needed, clone an instantiation of this prototypical object. **Encapsulates:** The default settings of an object when instantiated.		**Prototype**

Add a static member to the class that refers to the first instantiation of this object (initially it is null). Then, add a static method that instantiates this class if this member is null (and sets this member's value) and then returns the value of this member. Finally, set the constructor to protected or private so no one can directly instantiate this class and bypass this mechanism.
 	
Encapsulates: That there is only one of these objects allowed.
	PSEUDO CODE (Java)
	
	class Singleton {
	public static Singleton Instance();
	protected Singleton();
	private static _instance= null;
	
	Singleton Instance () {
	if _instance== null)
	_instance= new Singleton;
	return _instance
	} }
		Singleton

Design Pattern Matrix

STRUCTURAL PATTERNS	
	Notes on the patterns
Adapter	**Indicators in analysis:** Normally don't worry about interfaces here, so don't usually think about it. However, if you know some existing code is going to be incorporated into your system, it is likely that an adapter will be needed since it is unlikely this pre-existing code will have the correct interface. **Indicator in design:** Something has the right stuff but the wrong interface. Typically used when you have to make something that's a derivative of an abstract class we are defining or already have. **Field notes:** The adapter pattern allows you to defer concern about what the interfaces of your pre-existing objects look like since you can easily change them. Also, adapter can be implemented through delegation (run-time) or through multiple inheritance (C++). We call these variations Object Adapter, and Class Adapter, respectively. **Principle Illustrated:** Make interfaces work for you, not against you. If you don't have what you want – you can change it easily.
Bridge	**Indicators in analysis:** There are a set of related objects using another set of objects. This second set represents an implementation of the first set. The first set uses the second set in varying ways. **Indicators in design:** There is a set of derivations that use a common set of objects to get implemented. **Indication pattern is not being used when it should be:** There is a class hierarchy that has redundancy in it. The redundancy is in the way these objects use another set of object. Also, if a new case is added to this hierarchy or to the classes being used, that will result in multiple classes being added. **Relationships involved:** The using classes (the GoF's "Abstraction") use the used classes (the GoF's "Implementation") in different ways but don't want to know which implementor is present. The pattern "bridges" the *what* (the abstraction) to the *how* (the implementation). **Field notes:** Although the implementer to use can vary from instance to instance, typically only one implementer is used for the life of the using object. This means we usually select the implementer at construction time, either passing it into the constructor or having the constructor decide which implementer should be used.
Composite	**Indicators in analysis:** There are single things and groups of things that you want to treat the same way. The groups of things are made up of other groups and of single things (i.e., they are hierarchically related). **Indicators in design:** Some objects are comprised of collections of other objects, yet we want to handle all of these objects in the same way. **Indication pattern is not being used when it should be:** The code is distinguishing between whether a single object is present or a collection of objects is present. **Variation encapsulated:** Whether an item is a single entity or whether it is composed of several sub-components. **Field notes:** Whether or not to expose an interface that would allow the client to navigate the composite is a decision that must be considered. The ideal composite would hide its structure, and thus the navigation would not be supported, but specifics in the problem domain often do not allow this. Often, using a Data Object can eliminate the need for traversal by the client
Façade	**Indicators in analysis:** A complex system will be used which will likely not be utilized to its full extent. **Indicators in design:** Reference to an existing system is made in similar ways. That is, you see combinations of calls to a system repeated over and over again. **Indication pattern is not being used when it should be:** Many people on a team have to learn a new system although each person is only using a small aspect of it. **Field notes:** Not usually used for encapsulating variation, but different facades derived from the same abstract class can encapsulate different sub-systems. This is called an *encapsulating façade.* The encapsulating facade can have many positive side-effects, including support for demonstration/limited function versions of an application. **Essence of Pattern:** Don't settle for complex – make it simple.
Proxy Virtual	**Indicators in analysis and design:** Performance issues (speed or memory) can be foreseen because of the cost of having objects around before they are actually used. **Indication pattern is not being used when it should be:** Objects are being instantiated before they are actually used and the extent of this is causing performance problems. **Variation encapsulated:** Although each proxy contains only one new function or way of connecting to the proxy object, this function can be changed (statically) in the future without affecting those objects that use the proxy. **Field notes:** This pattern often comes up to solve scalability issues or performance issues that arise after a system is working.

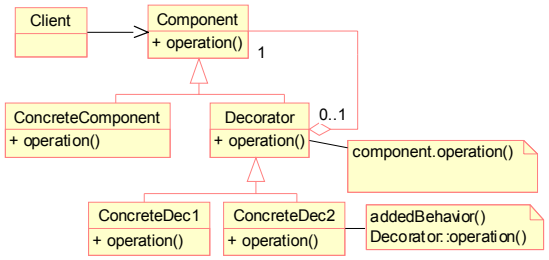
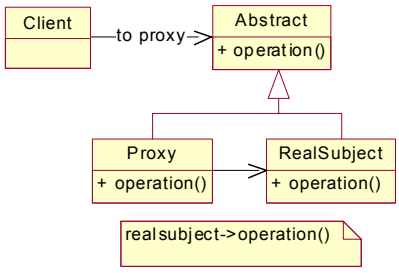
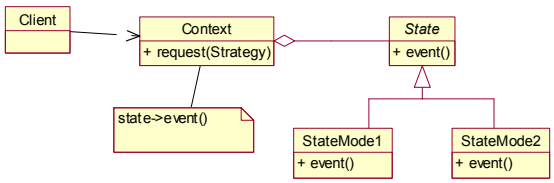
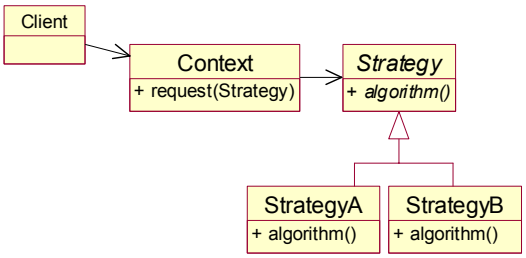
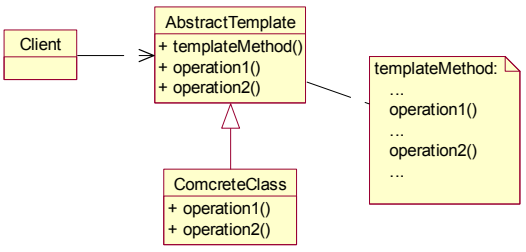
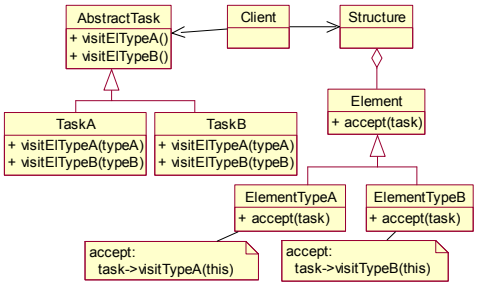
Design Pattern Matrix

STRUCTURAL PATTERNS		
How it is implemented + what it encapsulates	**Class Diagram**	
Contain the existing class in another class. Have the containing class match the required interface and call the contained class's methods **Encapsulates** that another class has the wrong interface.		**Adapter**
Encapsulate the implementations in an abstract class and contain a handle to it in the base class of the abstraction being implemented. In Java can also use interfaces instead of an abstract class for the implementation. **Encapsulates** dependent/nested variations of function.		**Bridge**
Set up an abstract class that represents all elements in the hierarchy. Define at least one derived class that represents the individual components. Also, define at least one other class that represents the composite elements (i.e., those elements that contain multiple components). In the abstract class, define abstract methods that the client objects will use. Finally, implement these for each of the derived classes. **Encapsulates** the structure of a complex collection.		**Composite**
Define a new class (or classes) that has the required interface. Have this new class use the existing system. **Encapsulates** the true nature of a sub-system.		**Façade**
The Client refers to the proxy object instead of an object from the original class. The proxy object remembers the information required to instantiate the original class but defers its instantiation. When the object from the original class is actually needed, the proxy object instantiates it and then makes the necessary request to it. **Encapsulates** the deferring of some action (e.g., instantiation, making a connection) and the rule(s) to manage it.		**Proxy - Virtual**

Design Pattern Matrix

BEHAVIORAL PATTERNS	
	Notes on the patterns
Decorator	**Indicators in analysis:** There is some action that is always done, there are other actions that may need to be done. **Indicators in design:** 1) There is a collection of actions; 2) These actions can be added in any combination to an existing function; 3) You don't want to change the code that is using the decorated function. **Indication pattern is not being used when it should be:** There are switches that determine if some optional function should be called before some existing function. **Field notes:** This pattern is used extensively in the JDK and .NET for I/O. Note the decorators can be one-way (void return) or bucket-brigade (pass down the chain, then "bubble" back up). The Decorator Pattern should be thought of as a collection of optional behavior preceding the always done "ConcreteComponent". The form of this collection does ***not*** have to be a linked-list, but it does have to have the same interface as the "ConcreteComponent" so the Client is unaware that the "ConcreteComponent" is being decorated.
Proxy | New Function	**Indicators in design:** We need some particular action to occur before some object we already have is called. **Indication pattern is not being used when it should be:** We precede a function with the same code every time it is used. Or, we add a switch to an object so it sometimes does some pre-processing and sometimes doesn't. **Variation encapsulated:** Although each proxy contains only one new function or way of connecting to the proxy object, this function can be changed (statically) in the future without affecting those objects that use the proxy. **Field notes:** Proxies are useful to encapsulate a special function that is sometimes used prior to calling an existing object.
State	**Indicators in analysis and design:** We have behaviors that change, depending upon the state we are in. **Indication pattern is not being used when it should be:** The code keeps track of the mode the system is in. Each time an event is handled, a switch determines which code to execute (based on the mode the system is in). The rules for transitioning between the patterns may also be complex. **Field notes:** We define our classes by looking at the following questions: 1. What are our states? 2. What are the events we must handle? 3. How do we handle the transitions between states?
Strategy	**Indicators in analysis:** There are different implementations of a business rule. **Indicators in design:** You have a place where a business rule (or algorithm) changes. **Indication pattern is not being used when it should be:** A switch is present that determines which business-rule to use. A class hierarchy is present where the main difference between the derivations is an overridden method. **Relationships involved:** An object that uses different business rules that do conceptually the same thing (Context-Algorithm relationship). A client object that gives another object the rule to use (Client-Context relationship). **Variation encapsulated:** The different implementations of the business rules. **Field notes:** The essence of this pattern is that the Context does not know which rule it is using. Either the Client object gives the Context the Strategy object to use, the Context asks a factory (or configuration) object for the correct Strategy object to use, or the Context is built by a factory with the right Strategy object to use…or a combination of these approaches. Reduced intimacy between the context and the algorithm can make Strategy challenging at times.
Template Method	**Indicators in analysis:** There are different procedures that are followed that are essentially the same, except that each step does things differently. **Indicators in design:** You have a consistent set of steps to follow but individual steps may have different implementations. **Indication pattern is not being used when it should be:** Different classes implement essentially the same process flow. **Variation encapsulated:** Different steps in a similar procedure. **Field notes:** The template pattern is most useful when it is used to abstract out a common flow between two similar processes.
Visitor	**Indicators in analysis and design:** You have a reasonably stable set of classes for which you need to add new functions. You can add tasks to be performed on this set without having to change it. **Variation encapsulated:** A set of tasks to run against a set of derivations. **Field notes:** This is a useful pattern for writing sets of tests that you can run when needed. The potential for change in the class structure being visited must be considered – Visitor has maintenance issues when the visited classes change. Adding Visitors in the future, on the other hand, tends to be quite easy.

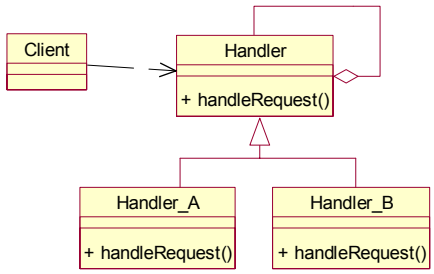
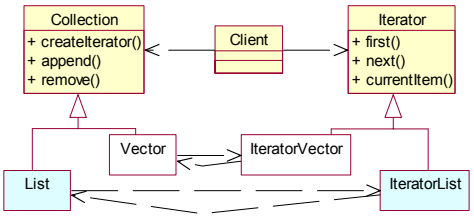
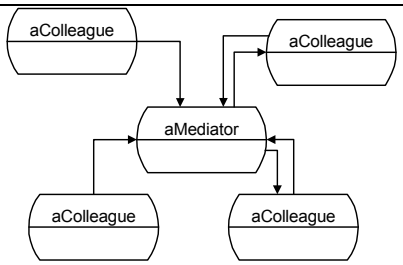
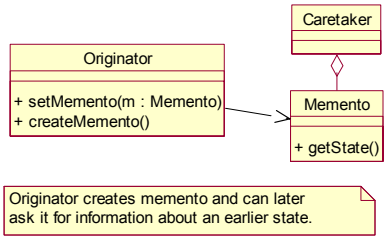
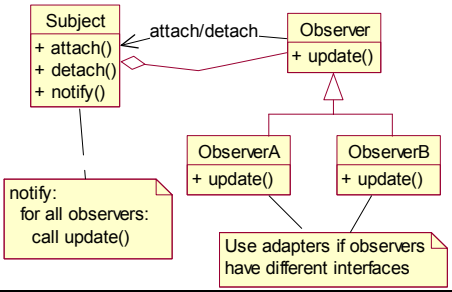
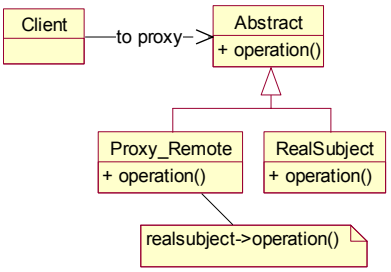
Design Pattern Matrix

BEHAVIORAL PATTERNS		
How it is implemented	**Class Diagram**	
Set up an abstract class that represents both the original class and the new functions to be added. Have each contain a handle to an object of this type (in reality, of a derived type). In our decorators, perform the additional function and then call the contained object's *operation* method. Optionally, call the contained object's *operation* method first, then do your own special function. **Encapsulates** the sequence, cardinality and even the existence of optional behaviors.		**Decorator**
The Client refers to the proxy object instead of an object from the original class. The proxy object creates the RealSubject when it is created. Requests come to the Proxy, which does its initial function (possibly), passes the request (possibly) to the RealSubject and then does (possibly) some post processing. **Encapsulates** the new function required by the client.		**Proxy | New Function**
Define an abstract class that represents the state of an application. Derive a class for each possible state. Each of these classes can now operate independently of each other. State transitions can be handled either in the contextual class or in the states themselves. Information that is persistent across states should be stored in the context. States likely will need to have access to this (through *get* routines, of course). **Encapsulates** the states (or modalities) of a system and how it transitions from one to another.		**State**
Have the class that uses the algorithm contain an abstract class that has an abstract method specifying how to call the algorithm. Each derived class implements the algorithm as needed. **Encapsulates** that there are multiple algorithms.		**Strategy**
Create an abstract class that implements a procedure using abstract methods. These abstract methods must be implemented in derived classes to actually perform each step of the procedure. If the steps vary independently, each step may be implemented with a strategy pattern. **Encapsulates** the different implementations of the method.		**Template Method**
Make an abstract class that represents the tasks to be performed. Add a method to this class for each concrete class you started with (your original entities). Add a method to the classes that you are working on to call the appropriate method in this task class, giving a reference to itself to this method. **Encapsulates** additional functions added to a class hierarchy.		**Visitor**

Design Pattern Matrix

DECOUPLING PATTERNS	
	Notes on the patterns
Chain of Responsibility	**Indicators in analysis:** We have the several actions that may be done by different things. **Indicators in design:** We have several potential candidates to do a function. However, we don't want the client object to know which of these objects will actually do it. **Field notes:** This pattern can be used to chain potential candidates to perform an action together. A variation of Chain of Responsibility is to not stop when one object performs its function but to allow each object to do its action. Factories are useful when chains have dependencies and business rules that specific a particular order for the objects in the chain.
Iterator	**Indicators in analysis and design:** We have a collection of things but aren't clear what the right type of collection to use is. You want to hide the structure of a collection. Alternatively, you need to have variations in the way a collection is traversed. **Indication pattern is not being used when it should be:** Changing the underlying structure of a collection (say from a vector to a composite) will affect the way the collection is iterated over. **Variations encapsulated:** Type of collection used. **Field notes:** The Iterator pattern enables us to defer a decision on which type of collection structure to use.
Mediator	**Indicators in analysis and design:** Many objects need to communicate with many other objects yet this communication cannot be handled with the observer pattern. **Indication pattern is not being used when it should be:** The system is tightly coupled due to inter-object communication requirements. **Field notes:** When several objects are highly coupled in the way they interact, yet this set of rules can be encapsulated in one place. Mediator can be useful in decoupling tool-generated code (GUI's and database connections, for example) from hand-crafted code, and can also help in making dependent objects more testable.
Memento	**Indicators in analysis and design:** The state of an object needs to be remembered so we can go back to it (e.g., undo an action). **Indication pattern is not being used when it should be:** The internal state of an object is exposed to another object. Or, copies of an object are being made to remember the object's state, yet this object contains much information that is not state dependent. This means the object is larger than it needs to be or contains an open connection that doesn't need to be remembered. **Field notes:** This pattern is useful when making copies of the object whose state is being remembered would be inefficient. It's also commonly used to create "multiple undo levels". The use of inner classes can help to make Mementos robust, since it prevents a memento generated by one instance being applied accidentally to another.
Observer	**Indicators in analysis and design:** Different things (objects) that need to know when an event has occurred. This list of objects may vary from time to time or from case to case. **Indication pattern is not being used when it should be:** When a new object needs to be notified of an event occurring the programmer has to change the object that detects the event. **Variation encapsulated:** The list of objects that need to know about an event occurring. **Field notes:** This pattern is used extensively in the JFC for event handling and is supported with the *Observable* class and *Observer* interface. Also note that C# multicast delegates are essentially implementations of the Observer pattern. **Essence of pattern:** 1) there is a changing list of observers, 2) all observers have the same interface, 3) it is the observers responsibility to register it the event they are 'observing'
Proxy | Access-ibility	**Indicators in analysis and design:** Are any of the things we work with remote (i.e., on other machines)? An existing object needs to use an object on another machine and doesn't want to have to worry about making the connection (or even know about the remote connection). **Indication pattern is not being used when it should be:** The use of an object and the set-up of the connection to the object are found together in more than one place. **Variation encapsulated:** Although each proxy contains only one new function or way of connecting to the proxy object, this function can be changed (statically) in the future without affecting those objects that use the proxy. **Field notes:** The Proxy is a useful pattern to use when it is possible a remote connection will be needed in the future. In this case, only the Proxy object need be changed - not the object actually being used.

Design Pattern Matrix

DECOUPLING PATTERNS		
How it is implemented	**Class Diagram**	
Define an abstract class that represents possible handlers of a function. This class contains a reference to at most one other object derived from this type. Define an abstract method that the client will call. Each derived class must implement this method by either performing the requested operation (in its own particular way) or by handing it off to the Handler it refers to. Note: it may be that the job is never handled. You can implement a default method in the abstract class that is called when you reach the end of the chain. **Encapsulates** both which services are available and the preference of which service to do the task.		**Chain of Responsibility**
Define abstract classes for both collections and iterators. Have each derived collection include a method which instantiates the appropriate iterator. The iterator must be able to request the required information from the collection in order to traverse it appropriately. **Encapsulates** the nature of a collection.		**Iterator**
Define a central class that acts as a message routing service to all other classes. **Encapsulates** the dependencies between objects.		**Mediator**
Define a new class that can remember the internal state of another object. The Caretaker controls when to create these, but the Originator will actually use them when it restores its state. **Encapsulates** the internal state of an object.		**Memento**
Have objects (Observers) that want to know when an event happens, attach themselves to another object (Subject) that is actually looking for it to occur. When the event occurs, the subject tells the observers that it occurred. The Adapter pattern is sometimes needed to be able to implement the Observer interface for all the Observer type objects. **Encapsulates** who is dependent upon an event that an object looks for.		**Observer**
The Proxy pattern has a new object (the Proxy) stand in place of another, already existing object (the Real Subject). The proxy encapsulates any rules required for access to the real subject. The proxy object and the real subject object must have the same interface so that the Client does not need to know a proxy is being used. Requests made by the Client to the proxy are passed through to the Real Subject with the proxy doing any necessary processing to make the remote connection. **Encapsulates** how to access an object.		**Proxy | Access**

Design Pattern Matrix

THINGS TO LOOK FOR

Guide to finding patterns in the problem domain

Is there variation of a business rule or an implementation?
Do we need to add some function?

Strategy – do we have a varying rule?
Bridge – do we have one variation using another variation in a varying way?
Proxy – do we need to optionally add some new functionality to something that already exists?
Decorator – do we have multiple additional functions we may need to apply, but which and how many we add varies? Do we need to capture an order dependency for multiple additional functions?
Visitor – do we have new tasks that we will need to apply to our existing classes?

Are you concerned with interfaces, changing, simplifying or handling disparate type objects in the same way?

Adapter – do we have the right stuff but the wrong interface? (Used to fit classes into patterns as well)
Composite – do we have units and groups and want to treat them the same way?
Façade – do we want to simplify, beautify, or OO-ify an existing class or syb-system?

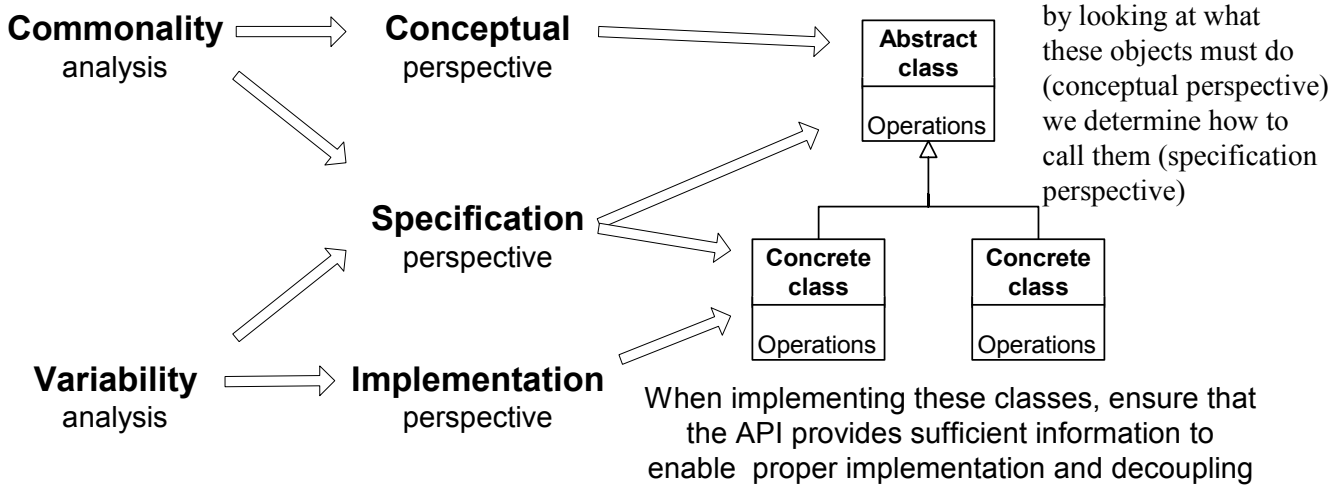
Are we trying to decouple things?

Observer – do various entities need to know about events that have occurred?
Chain of Responsibility – do we have different objects that can do the job but we don't want the client object know which is actually going to do it?
Iterator – do we want to separate the collection from the client that is using it so we don't have to worry about having the right collection implementation?
Mediator – do we have a lot of coupling in who must talk to who?
State – do we have a system with lots of states where keeping track of code for the different states is difficult?

Are we trying to make things?

Abstract Factory – do we need to create families (or sets) of objects?
Builder - do we need to create our objects with several steps?
Factory Method – do we need to have derived classes figure out what to instantiate?

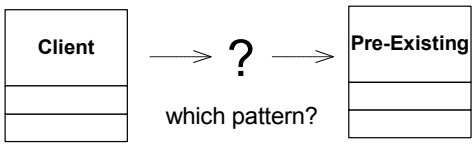
Remember the relationship between commonality/variability analysis, the conceptual, specification, implementation perspectives and how these are implemented in object-oriented languages.



Design Pattern Matrix

Comparing patterns

Comparing Adapter, Façade, Proxy and Decorator



	Adapter	Façade	Proxy	Decorator
Have pre-existing stuff?	**Yes**	**Yes**	**Yes**	**Yes**
Is the client object expecting a particular interface?	**Yes**	**No**	**Yes**	**Yes**
Want to make a new interface to simplify things?	**No**	**Yes**	**No**	**No**
Modifying interface to what the client objects expect?	**Yes**	**No**	**No**	**No**
Want some new function before or after the normal action?	**Maybe, but not the intent**	**Maybe, but not the intent**	**Yes (statically)**	**Yes (dynamically)**

Comparing Strategy and Bridge

	Strategy	Bridge	State
Contains base class with derivations?	**Yes**	**Yes**	**Yes**
Number of public methods in used classes?	**1**	**1-n**	**1-n**
Number of classes using the used classes.	**1**	**1-n**	**1**
Does implementation used change from call to call?	**Usually**	**No**	**Yes**
Who decides which implementation to use?	**Client or config**	**Client, config or constuctor**	**Context, States, or Transition Method**
Does current implementation have impact on next?	**No**	**No**	**Yes**

Comparing Bridge and Visitor

The ***Bridge Pattern*** is about how to have different implementations of the same functionality for a group of related objects (the derivations of the "Abstraction"). The ***Visitor Pattern*** is about how to add new functionality to a group of related objects (the derivations of the "Structure").

CONTENTS INCLUDE:

- Chain of Responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Observer
- Template Method and more...

Design Patterns

ABOUT DESIGN PATTERNS

This Design Patterns refcard provides a quick reference to the original 23 Gang of Four design patterns, as listed in the book *Design Patterns: Elements of Reusable Object-Oriented Software*. Each pattern includes class diagrams, explanation, usage information, and a real world example.

Creational Patterns: Used to construct objects such that they can be decoupled from their implementing system.

Structural Patterns: Used to form large object structures between many disparate objects.

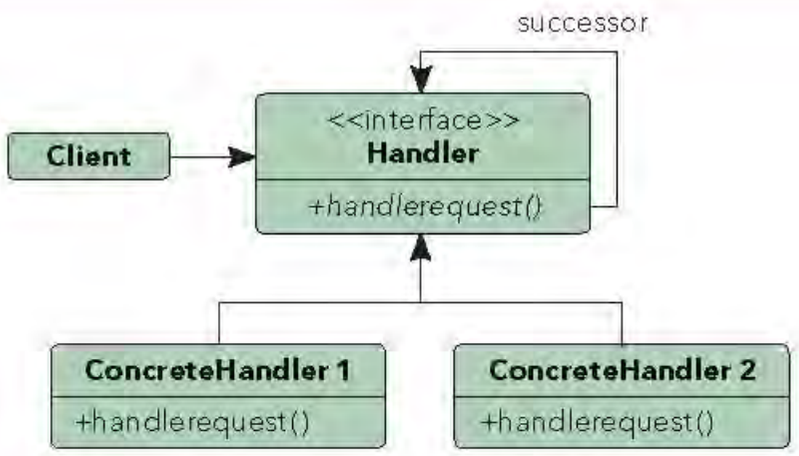
Behavioral Patterns: Used to manage algorithms, relationships, and responsibilities between objects.

Object Scope: Deals with object relationships that can be changed at runtime.

Class Scope: Deals with class relationships that can be changed at compile time.

C	Abstract Factory	S	Decorator	C	Prototype
S	Adapter	S	Facade	S	Proxy
S	Bridge	C	Factory Method	B	Observer
C	Builder	S	Flyweight	C	Singleton
B	Chain of Responsibility	B	Interpreter	B	State
B	Command	B	Iterator	B	Strategy
S	Composite	B	Mediator	B	Template Method
		B	Memento	B	Visitor

CHAIN OF RESPONSIBILITY — Object Behavioral



Purpose

Gives more than one object an opportunity to handle a request by linking receiving objects together.

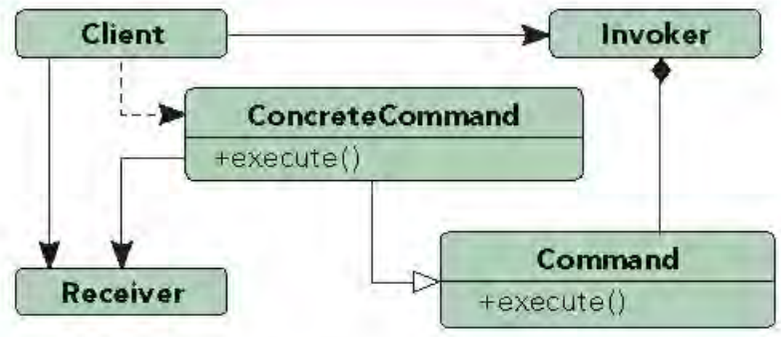
Use When

- Multiple objects may handle a request and the handler doesn't have to be a specific object.
- A set of objects should be able to handle a request with the handler determined at runtime.
- A request not being handled is an acceptable potential outcome.

Example

Exception handling in some languages implements this pattern. When an exception is thrown in a method the runtime checks to see if the method has a mechanism to handle the exception or if it should be passed up the call stack. When passed up the call stack the process repeats until code to handle the exception is encountered or until there are no more parent objects to hand the request to.

COMMAND — Object Behavioral



Purpose

Encapsulates a request allowing it to be treated as an object. This allows the request to be handled in traditionally object based relationships such as queuing and callbacks.

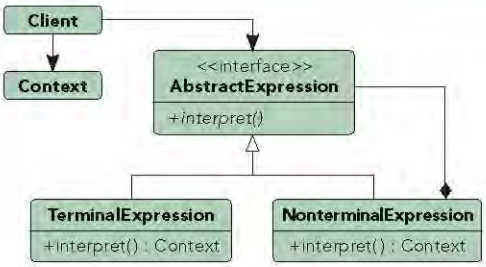
Use When

- You need callback functionality.
- Requests need to be handled at variant times or in variant orders.
- A history of requests is needed.
- The invoker should be decoupled from the object handling the invocation.

Example

Job queues are widely used to facilitate the asynchronous processing of algorithms. By utilizing the command pattern the functionality to be executed can be given to a job queue for processing without any need for the queue to have knowledge of the actual implementation it is invoking. The command object that is enqueued implements its particular algorithm within the confines of the interface the queue is expecting.

INTERPRETER — Class Behavioral



Purpose

Defines a representation for a grammar as well as a mechanism to understand and act upon the grammar.

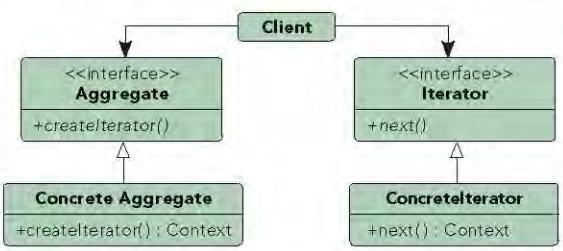
Use When

- There is grammar to interpret that can be represented as large syntax trees.
- The grammar is simple.
- Efficiency is not important.
- Decoupling grammar from underlying expressions is desired.

Example

Text based adventures, wildly popular in the 1980's, provide a good example of this. Many had simple commands, such as "step down" that allowed traversal of the game. These commands could be nested such that it altered their meaning. For example, "go in" would result in a different outcome than "go up". By creating a hierarchy of commands based upon the command and the qualifier (non-terminal and terminal expressions) the application could easily map many command variations to a relating tree of actions.

ITERATOR — Object Behavioral



Purpose

Allows for access to the elements of an aggregate object without allowing access to its underlying representation.

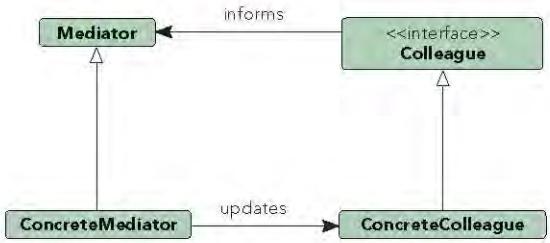
Use When

- Access to elements is needed without access to the entire representation.
- Multiple or concurrent traversals of the elements are needed.
- A uniform interface for traversal is needed.
- Subtle differences exist between the implementation details of various iterators.

Example

The Java implementation of the iterator pattern allows users to traverse various types of data sets without worrying about the underlying implementation of the collection. Since clients simply interact with the iterator interface, collections are left to define the appropriate iterator for themselves. Some will allow full access to the underlying data set while others may restrict certain functionalities, such as removing items.

MEDIATOR — Object Behavioral



Purpose

Allows loose coupling by encapsulating the way disparate sets of objects interact and communicate with each other. Allows for the actions of each object set to vary independently of one another.

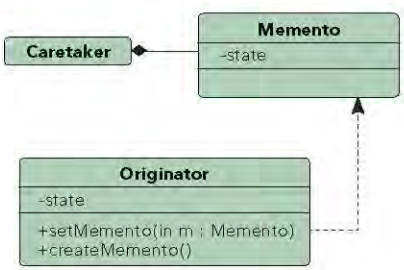
Use When

- Communication between sets of objects is well defined and complex.
- Too many relationships exist and common point of control or communication is needed.

Example

Mailing list software keeps track of who is signed up to the mailing list and provides a single point of access through which any one person can communicate with the entire list. Without a mediator implementation a person wanting to send a message to the group would have to constantly keep track of who was signed up and who was not. By implementing the mediator pattern the system is able to receive messages from any point then determine which recipients to forward the message on to, without the sender of the message having to be concerned with the actual recipient list.

MEMENTO — Object Behavioral



Purpose

Allows for capturing and externalizing an object's internal state so that it can be restored later, all without violating encapsulation.

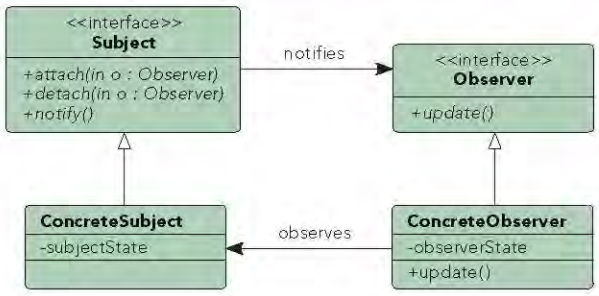
Use When

- The internal state of an object must be saved and restored at a later time.
- Internal state cannot be exposed by interfaces without exposing implementation.
- Encapsulation boundaries must be preserved.

Example

Undo functionality can nicely be implemented using the memento pattern. By serializing and deserializing the state of an object before the change occurs we can preserve a snapshot of it that can later be restored should the user choose to undo the operation.

OBSERVER — Object Behavioral



Purpose

Lets one or more objects be notified of state changes in other objects within the system.

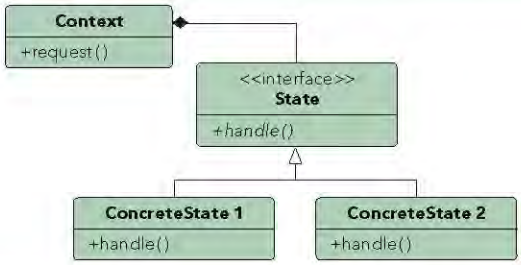
Use When

- State changes in one or more objects should trigger behavior in other objects
- Broadcasting capabilities are required.
- An understanding exists that objects will be blind to the expense of notification.

Example

This pattern can be found in almost every GUI environment. When buttons, text, and other fields are placed in applications the application typically registers as a listener for those controls. When a user triggers an event, such as clicking a button, the control iterates through its registered observers and sends a notification to each.

STATE — Object Behavioral



Purpose

Ties object circumstances to its behavior, allowing the object to behave in different ways based upon its internal state.

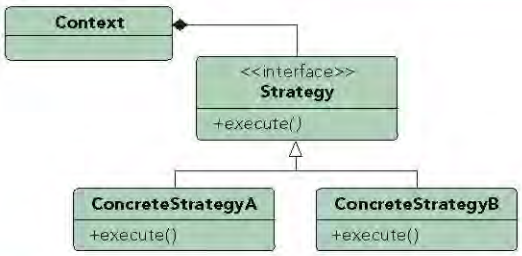
Use When

- The behavior of an object should be influenced by its state.
- Complex conditions tie object behavior to its state.
- Transitions between states need to be explicit.

Example

An email object can have various states, all of which will change how the object handles different functions. If the state is "not sent" then the call to send() is going to send the message while a call to recallMessage() will either throw an error or do nothing. However, if the state is "sent" then the call to send() would either throw an error or do nothing while the call to recallMessage() would attempt to send a recall notification to recipients. To avoid conditional statements in most or all methods there would be multiple state objects that handle the implementation with respect to their particular state. The calls within the Email object would then be delegated down to the appropriate state object for handling.

STRATEGY — Object Behavioral



Purpose

Defines a set of encapsulated algorithms that can be swapped to carry out a specific behavior.

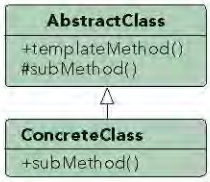
Use When

- The only difference between many related classes is their behavior.
- Multiple versions or variations of an algorithm are required.
- Algorithms access or utilize data that calling code shouldn't be exposed to.
- The behavior of a class should be defined at runtime.
- Conditional statements are complex and hard to maintain.

Example

When importing data into a new system different validation algorithms may be run based on the data set. By configuring the import to utilize strategies the conditional logic to determine what validation set to run can be removed and the import can be decoupled from the actual validation code. This will allow us to dynamically call one or more strategies during the import.

TEMPLATE METHOD — Class Behavioral



Purpose

Identifies the framework of an algorithm, allowing implementing classes to define the actual behavior.

Use When

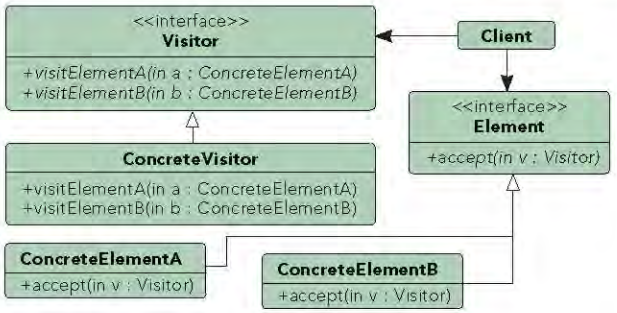
- A single abstract implementation of an algorithm is needed.
- Common behavior among subclasses should be localized to a common class.
- Parent classes should be able to uniformly invoke behavior in their subclasses.
- Most or all subclasses need to implement the behavior.

Example

A parent class, InstantMessage, will likely have all the methods required to handle sending a message. However, the actual serialization of the data to send may vary depending on the implementation. A video message and a plain text message will require different algorithms in order to serialize the data correctly. Subclasses of InstantMessage can provide their own implementation of the serialization method, allowing the parent class to work with them without understanding their implementation details.

VISITOR

Object Behavioral



Purpose

Allows for one or more operations to be applied to a set of objects at runtime, decoupling the operations from the object structure.

Use When

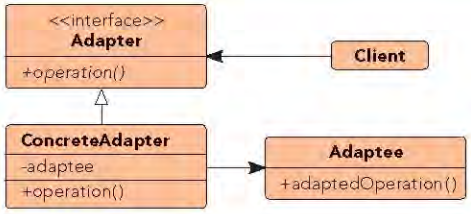
- An object structure must have many unrelated operations performed upon it.
- The object structure can't change but operations performed on it can.
- Operations must be performed on the concrete classes of an object structure.
- Exposing internal state or operations of the object structure is acceptable.
- Operations should be able to operate on multiple object structures that implement the same interface sets.

Example

Calculating taxes in different regions on sets of invoices would require many different variations of calculation logic. Implementing a visitor allows the logic to be decoupled from the invoices and line items. This allows the hierarchy of items to be visited by calculation code that can then apply the proper rates for the region. Changing regions is as simple as substituting a different visitor.

ADAPTER

Class and Object Structural



Purpose

Permits classes with disparate interfaces to work together by creating a common object by which they may communicate and interact.

Use When

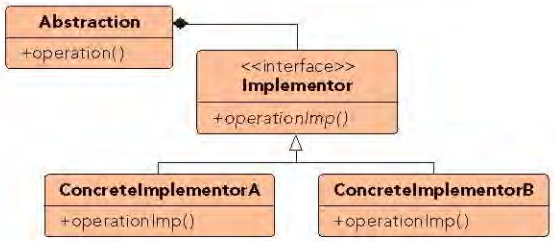
- A class to be used doesn't meet interface requirements.
- Complex conditions tie object behavior to its state.
- Transitions between states need to be explicit.

Example

A billing application needs to interface with an HR application in order to exchange employee data, however each has its own interface and implementation for the Employee object. In addition, the SSN is stored in different formats by each system. By creating an adapter we can create a common interface between the two applications that allows them to communicate using their native objects and is able to transform the SSN format in the process.

BRIDGE

Object Structural



Purpose

Defines an abstract object structure independently of the implementation object structure in order to limit coupling.

Use When

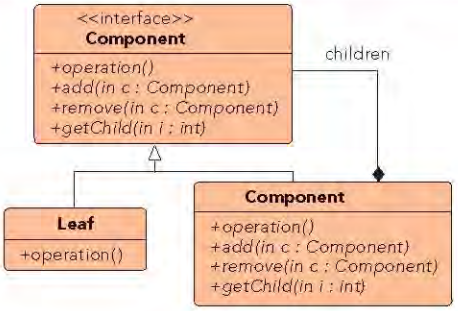
- Abstractions and implementations should not be bound at compile time.
- Abstractions and implementations should be independently extensible.
- Changes in the implementation of an abstraction should have no impact on clients.
- Implementation details should be hidden from the client.

Example

The Java Virtual Machine (JVM) has its own native set of functions that abstract the use of windowing, system logging, and byte code execution but the actual implementation of these functions is delegated to the operating system the JVM is running on. When an application instructs the JVM to render a window it delegates the rendering call to the concrete implementation of the JVM that knows how to communicate with the operating system in order to render the window.

COMPOSITE

Object Structural



Purpose

Facilitates the creation of object hierarchies where each object can be treated independently or as a set of nested objects through the same interface.

Use When

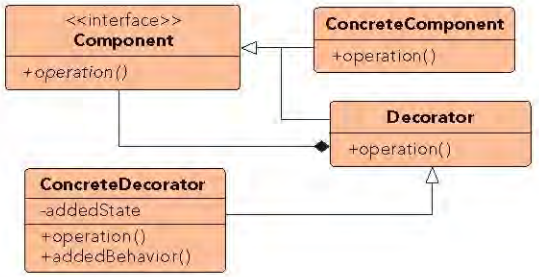
- Hierarchical representations of objects are needed..
- Objects and compositions of objects should be treated uniformly.

Example

Sometimes the information displayed in a shopping cart is the product of a single item while other times it is an aggregation of multiple items. By implementing items as composites we can treat the aggregates and the items in the same way, allowing us to simply iterate over the tree and invoke functionality on each item. By calling the getCost() method on any given node we would get the cost of that item plus the cost of all child items, allowing items to be uniformly treated whether they were single items or groups of items.

DECORATOR

Object Structural



Purpose

Allows for the dynamic wrapping of objects in order to modify their existing responsibilities and behaviors.

Use When

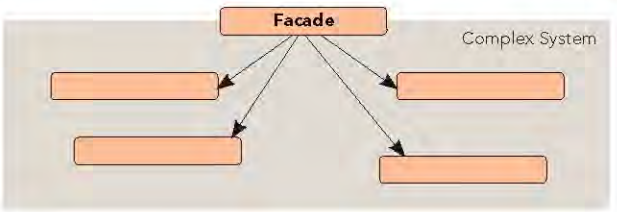
- Object responsibilities and behaviors should be dynamically modifiable.
- Concrete implementations should be decoupled from responsibilities and behaviors.
- Subclassing to achieve modification is impractical or impossible.
- Specific functionality should not reside high in the object hierarchy.
- A lot of little objects surrounding a concrete implementation is acceptable.

Example

Many businesses set up their mail systems to take advantage of decorators. When messages are sent from someone in the company to an external address the mail server decorates the original message with copyright and confidentiality information. As long as the message remains internal the information is not attached. This decoration allows the message itself to remain unchanged until a runtime decision is made to wrap the message with additional information.

FACADE

Object Structural



Purpose

Supplies a single interface to a set of interfaces within a system.

Use When

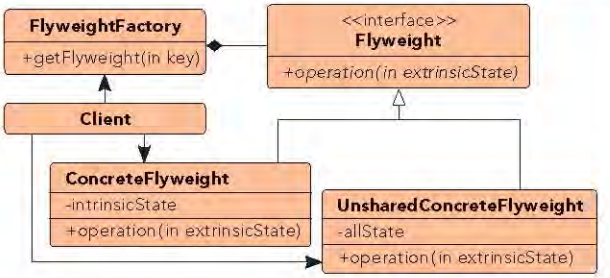
- A simple interface is needed to provide access to a complex system.
- There are many dependencies between system implementations and clients.
- Systems and subsystems should be layered.

Example

By exposing a set of functionalities through a web service the client code needs to only worry about the simple interface being exposed to them and not the complex relationships that may or may not exist behind the web service layer. A single web service call to update a system with new data may actually involve communication with a number of databases and systems, however this detail is hidden due to the implementation of the façade pattern.

FLYWEIGHT

Object Structural



Purpose

Facilitates the reuse of many fine grained objects, making the utilization of large numbers of objects more efficient.

Use When

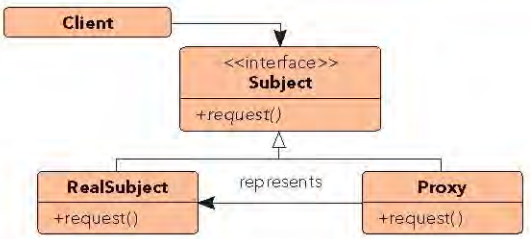
- Many like objects are used and storage cost is high.
- The majority of each object's state can be made extrinsic.
- A few shared objects can replace many unshared ones.
- The identity of each object does not matter.

Example

Systems that allow users to define their own application flows and layouts often have a need to keep track of large numbers of fields, pages, and other items that are almost identical to each other. By making these items into flyweights all instances of each object can share the intrinsic state while keeping the extrinsic state separate. The intrinsic state would store the shared properties, such as how a textbox looks, how much data it can hold, and what events it exposes. The extrinsic state would store the unshared properties, such as where the item belongs, how to react to a user click, and how to handle events.

PROXY

Object Structural



Purpose

Allows for object level access control by acting as a pass through entity or a placeholder object.

Use When

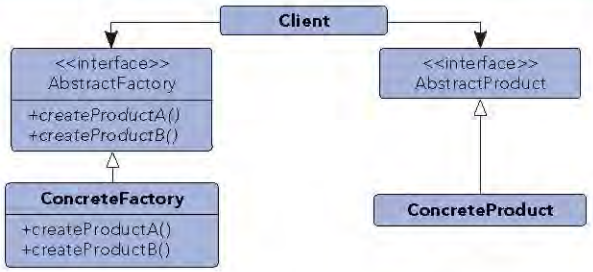
- The object being represented is external to the system.
- Objects need to be created on demand.
- Access control for the original object is required.
- Added functionality is required when an object is accessed.

Example

Ledger applications often provide a way for users to reconcile their bank statements with their ledger data on demand, automating much of the process. The actual operation of communicating with a third party is a relatively expensive operation that should be limited. By using a proxy to represent the communications object we can limit the number of times or the intervals the communication is invoked. In addition, we can wrap the complex instantiation of the communication object inside the proxy class, decoupling calling code from the implementation details.

ABSTRACT FACTORY

Object Creational



Purpose

Provide an interface that delegates creation calls to one or more concrete classes in order to deliver specific objects.

Use When

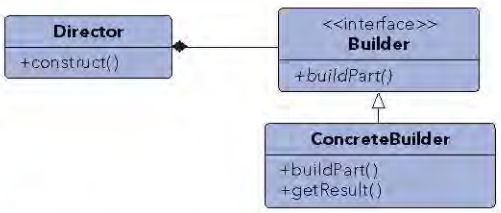
- The creation of objects should be independent of the system utilizing them.
- Systems should be capable of using multiple families of objects.
- Families of objects must be used together.
- Libraries must be published without exposing implementation details.
- Concrete classes should be decoupled from clients.

Example

Email editors will allow for editing in multiple formats including plain text, rich text, and HTML. Depending on the format being used, different objects will need to be created. If the message is plain text then there could be a body object that represented just plain text and an attachment object that simply encrypted the attachment into Base64. If the message is HTML then the body object would represent HTML encoded text and the attachment object would allow for inline representation and a standard attachment. By utilizing an abstract factory for creation we can then ensure that the appropriate object sets are created based upon the style of email that is being sent.

BUILDER

Object Creational



Purpose

Allows for the dynamic creation of objects based upon easily interchangeable algorithms.

Use When

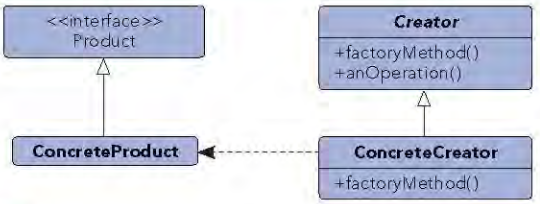
- Object creation algorithms should be decoupled from the system.
- Multiple representations of creation algorithms are required.
- The addition of new creation functionality without changing the core code is necessary.
- Runtime control over the creation process is required.

Example

A file transfer application could possibly use many different protocols to send files and the actual transfer object that will be created will be directly dependent on the chosen protocol. Using a builder we can determine the right builder to use to instantiate the right object. If the setting is FTP then the FTP builder would be used when creating the object.

FACTORY METHOD

Object Creational



Purpose

Exposes a method for creating objects, allowing subclasses to control the actual creation process.

Use When

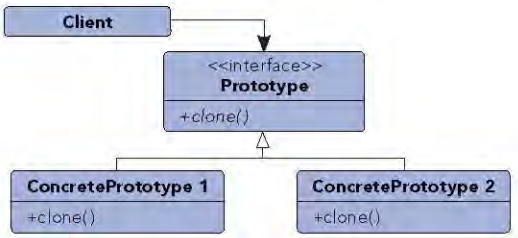
- A class will not know what classes it will be required to create.
- Subclasses may specify what objects should be created.
- Parent classes wish to defer creation to their subclasses.

Example

Many applications have some form of user and group structure for security. When the application needs to create a user it will typically delegate the creation of the user to multiple user implementations. The parent user object will handle most operations for each user but the subclasses will define the factory method that handles the distinctions in the creation of each type of user. A system may have AdminUser and StandardUser objects each of which extend the User object. The AdminUser object may perform some extra tasks to ensure access while the StandardUser may do the same to limit access.

PROTOTYPE

Object Creational



Purpose

Create objects based upon a template of an existing objects through cloning.

Use When

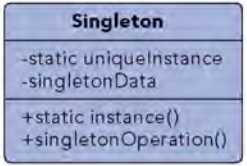
- Composition, creation, and representation of objects should be decoupled from a system.
- Classes to be created are specified at runtime.
- A limited number of state combinations exist in an object.
- Objects or object structures are required that are identical or closely resemble other existing objects or object structures.
- The initial creation of each object is an expensive operation.

Example

Rates processing engines often require the lookup of many different configuration values, making the initialization of the engine a relatively expensive process. When multiple instances of the engine is needed, say for importing data in a multi-threaded manner, the expense of initializing many engines is high. By utilizing the prototype pattern we can ensure that only a single copy of the engine has to be initialized then simply clone the engine to create a duplicate of the already initialized object. The added benefit of this is that the clones can be streamlined to only include relevant data for their situation.

SINGLETON

Object Creational



Purpose

Ensures that only one instance of a class is allowed within a system.

Use When

- Exactly one instance of a class is required.
- Controlled access to a single object is necessary.

Example

Most languages provide some sort of system or environment object that allows the language to interact with the native operating system. Since the application is physically running on only one operating system there is only ever a need for a single instance of this system object. The singleton pattern would be implemented by the language runtime to ensure that only a single copy of the system object is created and to ensure only appropriate processes are allowed access to it.

Patterns Description

There are many books on patterns. In fact, as our profession matures, we will probably identify many more patterns than those we have identified to this point in time. Perhaps you will discover a few. Because of this, I think that it is a forlorn hope to try to cover "the right set" of patterns in software development at this point. However, we can derive a lot of value if we discuss the forces behind the patterns. In doing so, we can use a number of very common patterns as examples for our discussion.

This, I hope, is one of the things I have achieved in the book overall.

The purpose of this appendix is to allow you to look up the patterns mentioned in the book when you do not know them or if you feel that your understanding of one pattern or another is lacking detail.

I present the patterns here in the way that I think of them and in the way that they have been the most useful to me: as collections of forces.

By *forces*, I mean the knowns, the driving issues, the critical decisions, and the understood effects of what we do. I divide these forces for each pattern to be discussed into three categories:

- ***Contextual forces*** (how we know to consider a given pattern)
  - *Motivation*. Why this pattern is used, what it is for, and the domain problem it solves.
  - *Encapsulation*. What the use of this pattern hides from other entities in the system.
  - *Procedural analog*. How we would have solved this same problem procedurally.

    - *Non-software analog*. A way of seeing the pattern conceptually, from life experience.
- ***Implementation forces*** (how to proceed once we decide to use a pattern)
    - *Example (UML and pseudo-code)*. A concrete instance of the pattern (which is not the pattern, but only an example).
    - *Questions, concerns, credibility checks*. Those issues that should come into the discussion once the pattern is suggested as a possible solution.
    - *Options in implementation*. Various options that are well-known in implementing this pattern. Often, the various implementations arise out of the questions and concerns that accompany the pattern.
- ***Consequent forces*** (what happens if we use a pattern)
    - *Testing issues*. Testing wisdom that accompanies the pattern. These are hints, tricks, and tips that can make patterns, which usually depend heavily on delegation, easier to test. They depend on a reasonable knowledge of mock and fake objects, which are covered in the sections on testing and test-driven development.
    - *Cost-benefit (gain-loss)*. What we get and what we pay when we use this pattern. Understanding these issues is a key aspect of making the decision to use or not to use a pattern. Patterns are as just valuable to us when they lead us *away* from themselves.

Code examples will be presented in a Java-like, C#-ish pseudo-code style that hopefully will be readable by all. The purpose of both the code and UML examples is to *illustrate*, not to *define*. Patterns are not code, nor are they diagrams; they are collections of forces that define the best-practices for solving a recurring problem in a given context. We provide code and UML examples because many people understand concepts better when they are accompanied by concrete examples.

Similarly, the procedural analogs are not meant to be literally applied: you would *not* use a Strategy pattern every time you see a branching conditional in procedural code. The purpose of the analog is to allow us to use the part of our brains that understands procedural/algorithmic programming to suggest a possible pattern alternative, but only to ensure that we consider all of our options before making an implementation decision. The decision is, and in my opinion always will be, a human one.

The patterns that help to define our profession are not static, but are rather a living, changing thing that we must always seek to improve and expand upon. This must be a community effort.

The Abstract Factory Pattern

Contextual Forces

Motivation

Create an interface for creating sets of dependant or related instances that implement a set of abstract types. The Abstract Factory coordinates the instantiation of sets of objects that have varying implementations in such a way that only legitimate combinations of instances are possible, and hides these concrete instances behind a set of abstractions.

Encapsulation

Issues hidden from the consuming (client) objects include

- The number of sets of instances supported by the system
- Which set is currently in use
- The concrete types that are instantiated at any point
- The issue upon which the sets vary

Procedural Analog

A fairly common need for something like the Abstract Factory arises when supporting multiple operating systems in a single application. To accomplish this, you would need to select the right set of behaviors: a disk driver, a mouse driver, a graphics driver, and so forth, for the operating system that the application is currently being installed for.

In a non-object-oriented system, you could accomplish this via conditional compilation, switching in the proper library for the operating system in question.

```
#IFDEF Linux
     include Linux_Drv.lib
#ENDIF

#IFDEF Windows
     include Windows_Drv.lib
#ENDIF
```

If the libraries in question contained routines that were named the same, and worked the same as other libraries for other operating systems, such that the consuming application code could reference them without regard to the particular operating system in use in a given case, it would simplify the code overall, and allow for a smoother transition between operating systems and in supporting new operating systems in the future. The Abstract Factory performs a similar role, but using objects, abstractions, and instantiation.

Another procedural form would be a set of switch/case statements, all based on the same variable, each of which varied a different service. The services in question would likely be implemented with patterns that hid their variations, such as the Strategy pattern, and would transition together when the value of the variable changed.

Non-Software Analog

I have two toolkits in my main toolbox.

One contains a set of wrenches (box-end wrenches, socket wrenches, closed-end wrenches) that are designed to work on a car that comes from overseas, and thus has metric measurements (centimeters, millimeters, and so forth).

The other contains an identical set (box-end, socket, closed-end) that is designed instead for an engine with English measurements (inches, quarter-inches, and so forth).

I do not have engines that have both metric and English bolts, and therefore I only use one set of wrenches or the other. The two toolkits encapsulate the difference: I choose the toolkit that applies to a given engine, and I know that all the wrenches I pull from it will be appropriate,

making my maintenance process much simpler and allowing me to concentrate on the tune-up or other process I am trying to accomplish.

Implementation Forces

Example

Let's assume we are doing business in two countries, the United States and Canada. Because of this, our application must be able to calculate taxes for both countries, as we as calculate freight charges based on the services we use in each case. Also, we must determine that an address is properly formatted given the country we're dealing with at any point in time. To support this, we have the abstractions in place CalcTax, CalcFreight, and AddressVer, which have implementing classes for each country in each case.

The client would be designed to take an implementation of Abstract Factory in its constructor, and then use it to make all the helper objects that it needs. Note that since the client is designed to work only with a single factory, and since there is no factory version that produces, say, a USTax object and a CanadaFreight object, it is impossible for the client to obtain this illegitimate combination of helper objects (see Figure B.1).

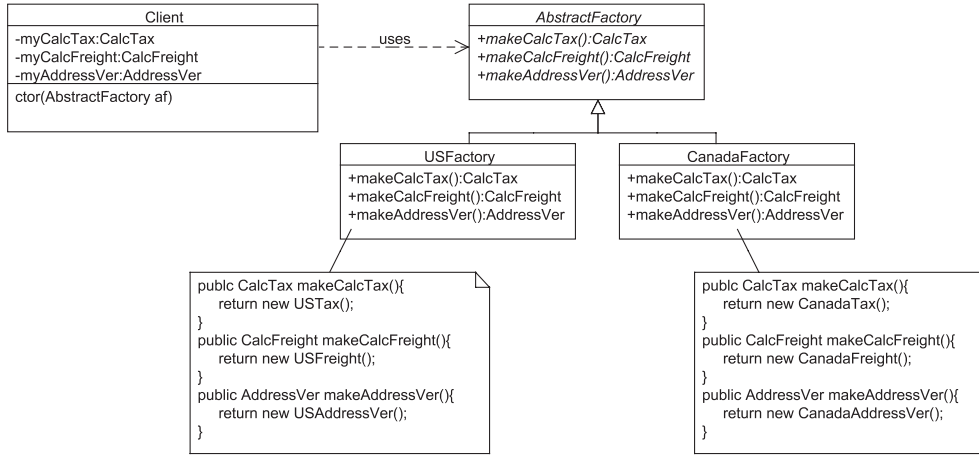


Figure B.1 A typical Abstract Factory, in this case, varying services that discreetly bind to a given nation

```
public class Client{
     private CalcTax myCalcTax;
     private CalcFreight myCalcFreight;
     private AddressVer myAddressVer;

     public Client(AbstractFactory af){
          myCalcTax = af.makeCalcTax();
          myCalcFreight = af.makeCalcFreight();
          myAddressVer = af.makeAddressVer();
     }

     // The rest of the code uses the helper objects generically
}

public abstract class AbstractFactory{
     abstract CalcTax makeCalcTax();
     abstract CalcFreight makeCalcFreight();
     abstract AddressVer makeAddressVer();
}

public class USFactory : AbstractFactory{
     public CalcTax makeCalcTax(){
          return new USCalcTax();
     }
     public CalcFreight makeCalcFreight(){
          return new USCalcFreight();
     }
     public AddressVer makeAddressVer(){
          return new USAddressVer();
}

public class CanadaFactory : AbstractFactory{
     public CalcTax makeCalcTax(){
          return new CanadaCalcTax();
     }
     public CalcFreight makeCalcFreight(){
          return new CanadaCalcFreight();
     }
     public AddressVer makeAddressVer(){
          return new CanadaAddressVer();
}
```

Questions, Concerns, Credibility Checks

For the Abstract Factory to be effective, there must be a set of abstractions with multiple implementations; these implementations must transition together under some circumstance (usually a large variation in the system); and these must all be resolvable to a consistent set of interfaces. In the preceding example, all tax systems in the supported countries must be supportable by a common interface, and the same must be true for the other services.

An obvious question arises: What entity determines the right concrete factory to instantiate, and deliver to the client entity? As this is an instantiation issue, the preferred approach is to encapsulate this behavior in an entity that is separate from any consuming object (encapsulation of construction). This can be done in an additional factory (arguably a Factory factory), or as is often the case, in a static method in the base (Abstract) class of the factory.

```
public abstract class AbstractFactory{
     abstract CalcTax makeCalcTax();
     abstract CalcFreight makeCalcFreight();
     abstract AddressVer makeAddressVer();
     public static AbstractFactory getAFtoUse(String customerCode){
          if(customerCode.startsWith("U")){
               return new USFactory();
          } else {
               return new CanadaFactory();
          }
     }
}
```

Note that I am not overly fond of this approach (it creates a mixture of perspectives in this abstract type; it is both a conceptual entity and an implementation of a factory), but it is nonetheless quite common.

Also, you must ask how often and under what circumstances does the proper set of objects change? The preceding example sets the factory implementation in the client's constructor, which implies little change during its lifetime. If more flexibility is required, this can be accomplished another way.

In determining many how object types are in each set, how many sets there are, and the resulting number of combinations, you can get an early view of the complexity of the implementation. Also, there are sometimes objects that can be used in more than one family, such as a Euro object

that might be used as a Currency implementation for many different countries in the European Union, though their Tax object might all be distinct. The degree to which implementations can be shared, and intermixed, tends to reduce the burden on the implementing team, and is an early question to be asked.

Options in Implementation

That fact that the Gang of Four example, and the example shown earlier, uses an abstract class with derivations should not mislead you to assume that this is required for an Abstract Factory. The fact that abstract class may be used is not the reason the pattern is named *Abstract* Factory; in fact, it is simply one way of implementing the pattern.

For example, you could create a single concrete factory implementation, and for each method (`makeCalcTaxI()`, for instance) simply use a procedural switch or if/then logic to generate the correct instance. Although this would not be particularly object-oriented (and probably not ideal), it would still be rightly termed an Abstract Factory.

The term Abstract Factory indicates that all the entities being created are themselves abstractions. In the preceding example, `CalcTax`, `CalcFreight`, and `AddressVer` are all abstractions.

Another very popular way of implementing an Abstract Factory is to bind the factory to a database table, where fields contain the names of the proper classes to instantiate for a given client, and dynamic class loading is then used to instantiate the proper class. The advantage here is that changes to the rules of instantiation can be made simply by changing the values in the table.

```
//Example is in Java
//Assumes myDataSource and ID are set elsewhere. ID reflects
//the current customer.

CalcTax makeCalcTax () {
    String db = "jdbc:odbc:"+ myDataSource;
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
    String query = "SELECT CALC_TAX FROM mytable WHERE ID = "
                   + ID;

    Connection myConn = DriverManager.getConnection(db, "", "");
    Statement myStatement = myConn.createStatement();
    ResultSet myResults = myStatement.executeQuery(query);
```

```
    String classToInstantiate;
    classToInstantiate= myResults.getString("CALC_TAX");
    return Class.forName(classToInstantiate);
}
```

Consequent Forces

Testing Issues

As with factories in general, the Abstract Factory's responsibility is limited to the creation of instances, and thus the testable issue is whether the right set of instances is created under a given circumstance. Often, this is covered by the test of the entities that use the factory, but if it is not, the test can use type-checking to determine that the proper concrete types are created under the right set of circumstances.

Cost-Benefit (Gain-Loss)

When we use the Abstract Factory, we gain protection from illegitimate combinations of service objects. This means we can design the rest of the system for maximum flexibility, since we know that the Abstract Factory eliminates any concerns of the flexibility yielding bugs. Also, the consuming entity (client) or entities are incrementally simpler, since they can deal with the components at the abstract level. In our preceding e-commerce example, all notion of nationality is eliminated from the client, meaning that this same client will be usable in other nations in the future with little or no maintenance.

The Abstract Factory holds up well if the maintenance aspects are limited to new sets (new countries), or a given set changing an implementation (Canada changes its tax system). On the other hand, if an entirely new abstract concept enters the domain (trade restrictions to a new country), the maintenance issues are more profound as the Abstract Factory interface, all the existing factory implementations, and the client entities all have to be changed.

This is not a fault of the pattern, but rather points out the degree to which object-oriented systems are vulnerable to missing/new abstractions, and therefore reinforces the value of analysis practices like commonality-variability analysis.

The Adapter Pattern

Contextual Forces

Motivation

There are two primary reasons to use an adapter:

- To use the behavior of an existing object, using a different interface than it was designed with.
- To make an existing object exchangeable with a polymorphic set of objects.

The Adapter pattern is generally used when one or both of these motivations exists but the existing object cannot be changed.

Encapsulation

The Adapter encapsulates the *differentness* of the object being adapted. It hides the fact that the adaptee has a different interface, and also that it is of a different abstract type (or is purely concrete and has no abstract type).

Also, since the client would deal with a nonadapted (local) object and an adapted (foreign) object in the same way, the fact that the client is using one object in the first case and two in the second case is encapsulated. This is shown in Figure B.2. This is a limited form of the encapsulation of cardinality (see the Decorator pattern).

Procedural Analog

A method, function, or other code segment that redirects behavior to another method, function, or code segment hiding this redirection from the consuming code.

```
rval m(para p) {
      c = (cast)p;
      x = n(c);
      return (cast)x;
}
```

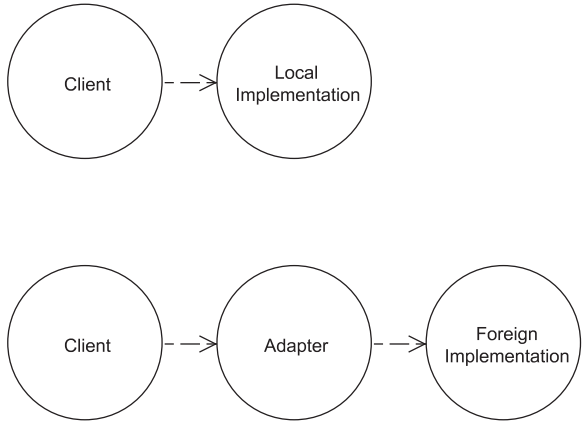


Figure B.2 Case 1: Local implementation, one object used by the client. Case 2: Foreign implementation with Adapter, two objects used by the client.

Non-Software Analog

Since I travel a lot, my primary computing device is a laptop. I notice that even though laptop batteries have improved dramatically (lithium-ion or nickel-mental-hydride as opposed to nickel-cadmium), they still seem to quickly lose their capability to hold a charge for very long.

As a result, when I am at a coffee shop, airport terminal gate, or hotel lobby, I am always "on the hunt" for a wall outlet to plug my power brick into. Unfortunately, I am rarely the only person on this hunt, and typically someone else will have found the one outlet, and is "hogging" it.

My solution is to look for a lamp. A lamp has what I need: 110 volts at 60 cycles, paid for by somebody else! It does not present it in the form that I need it, however (two little slots that my power brick plug will fit into), but in a form that is appropriate for a light bulb (a threaded socket and a copper tab in the bottom).

So, I carry one of these (see Figure B.3) in my laptop bag.

I (quietly) unscrew the light bulb from the lamp, screw in my adapter, and plug my laptop into it.

Figure B.3 Adapter for socket

Figure B.4 Wall socket

My laptop was designed to the specific interface shown in Figure B.4.

All wall sockets like the one in Figure B.4 (across the United States) are exchangeable for one another, from my laptop's point of view. In a sense, they are a polymorphic set. The lamp socket is not exchangeable because its interface is not acceptable, even though the "stuff it has to offer" is just what I need.

The adapter allows my laptop to consume the electricity the way it was designed to, and makes the lamp socket interchangeable with the wall sockets of the world.

Implementation Forces

Example

See Figure B.5.

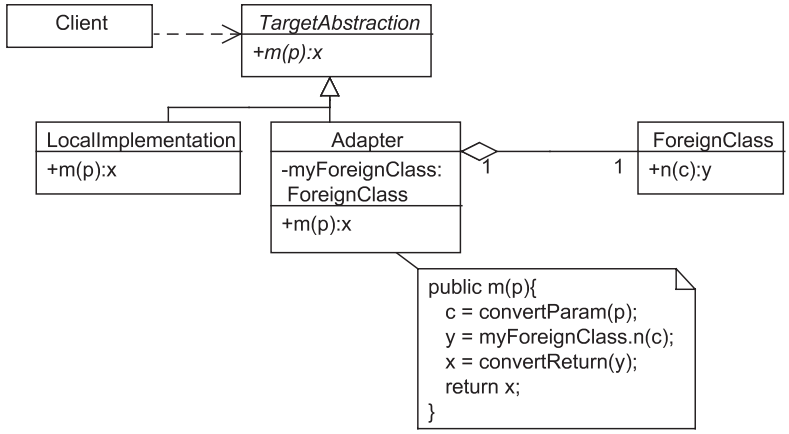


Figure B.5 Implementation forces

Code

```
public class TargetAbstraction {
      public abstract ret m(par p);
}

public class Adapter extends TargetAbstraction {
      private ForeignClass myForeignClass();
      public Adapter() {
            myForeignClass = new ForeignClass();
      }
      public ret m(par p) {
            var y = myForeignClass.n((cast)p);
            return (cast)y;
      }
}
```

Questions, Concerns, Credibility Checks

- How large is the delta (difference) between the interface that the foreign class offers and the interface of the target abstraction? If it is very large, this may be difficult to accomplish with a simple adapter. The Façade pattern should be considered in this case.
- If the foreign class throws exceptions, should the adapter rethrow them directly, rethrow them as a different exception, or deal with them by itself?
- How will the adapter obtain the instance of the foreign class? Direct instantiation (shown in the code) is actually not preferable, as it violates the encapsulation of construction principle; but the foreign class may not have been designed with this principle in mind. If this is the case, we may need to create an object factory to encapsulate the instantiation.
- Will the adapter need to add functionality to the foreign class? In other words, is the foreign class feature-poor to some degree?
- Will the adapter need to be stateful?

Options in Implementation

The preceding form uses delegation from the adapter to the foreign class to reuse the preexisting behavior. As shown in Figure B.6, another form of the adapter (called the *class adapter*) uses inheritance instead.

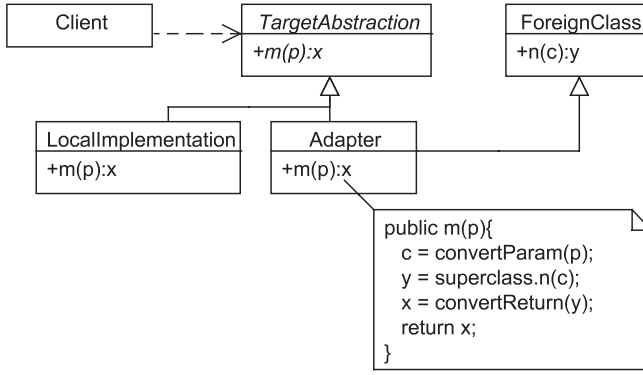


Figure B.6 Class adapter with inheritance

This implementation is more common with languages that support multiple inheritances, but it is also possible in single-inheritance languages if the `TargetAbstraction` is an Interface type.

Consequent Forces

Testing Issues

To test the adapter, you can use a mock or fake object in place of the foreign object (which would normally be adapted). The mock or fake can return predictable behavior for the adapter to convert, and also can record the action the adapter takes with the adaptee if this is deemed an appropriate issue to test.

Cost-Benefit (Gain-Loss)

The adapter is a low-cost solution and is therefore quite commonplace. The cost is the creation of an additional class, but the benefits are

- Encapsulated reuse of existing behavior
- Polymorphism (through an upcast) with a foreign class
- Promotes the open-closed principle
- If the construction of the foreign class was not encapsulated (which is common), the adapter can encapsulate it in its constructor. However, an object factory is preferred.

The Bridge Pattern

Contextual Forces

Motivation

Separate a varying entity from a varying behavior[1] so that these issues can vary independently. Sometimes, we say it this way: separate what something *is* from what it *does*, where both of these things vary for different reasons.

Encapsulation

The entity variation, the behavior variation, and the relationship between these variations are encapsulated. Also, we may wish to encapsulate that the Bridge pattern is being used at all.

Procedural Analog

Similar to a Strategy, a Bridge is analogous to the use of branching, conditional logic. However, in this case the logic is nested.

```
if(entityConditionA) {
     if(behaviorCondition1){
          // Algorithm A1
     } else {
          // Algorithm A2
     }
} else {
     if(behaviorCondition1){
          // Algorithm B1
     } else {
          // Algorithm B2
     }
}
```

1. The Gang of Four (Gamma, Helms, Johnson, Vlissides) said it this way: "Separate an abstraction from its implementation so the two can vary independently." Because most people use these words to mean "abstract superclass" and "implementing derived class," this can be a confusing motivational statement and does not reflect the original intent of the GoF. However, this wording is quite accurate given that these terms are being used in a different way. Abstraction = Entity, Implementation = Behavior or Action.

Non-Software Analog

My friends and I like to go to dinner together a lot. From the waitress' point of view, we are all the same—she considers us to be *patrons*, and she wants the same question answered no matter which of us she is speaking with: "whatlyahave?"

We are actually all different, of course, but that is not her concern. One difference that influences the outcome of this interaction is our ordering preference.

I usually get what is on special for the day. I like adventure. My younger brother, Christopher, says that I am crazy, because he says everyone knows "the special" is the thing that is just about to go bad. He says, "Get the specialty of the house; they take pride in doing that right." "Sure," I say, "but you always get the same thing if you do that." It is an old argument.

Andrea, my wife, likes salads and usually orders one. However, some places do not have them, so then she needs to get something meatless, whatever that might be.

Brenner always gets the most expensive thing on the menu. He has noticed that we always divide the check equally . . . and, well, that is Brenner.

That is one variation, or really, two: the variation of who we are and the variation of how we order food. These do not vary independently, however. The variation in our ordering preferences is what makes us different, in the context of being restaurant patrons.

The independent variation has to do with the restaurant we are currently visiting on a given evening. For example

> **Ocean Breeze:** Seafood, primarily, and very experimental cuisine
>
> **RedRocks:** Barbecue, mostly meat, very large portions: not Andrea's favorite place

Who we are varies and the food available varies, but each of us can use the menu in our own way to get what we want. From the waitress' point of view, she just wants an answer.

Implementation Forces

Example

See Figure B.7.

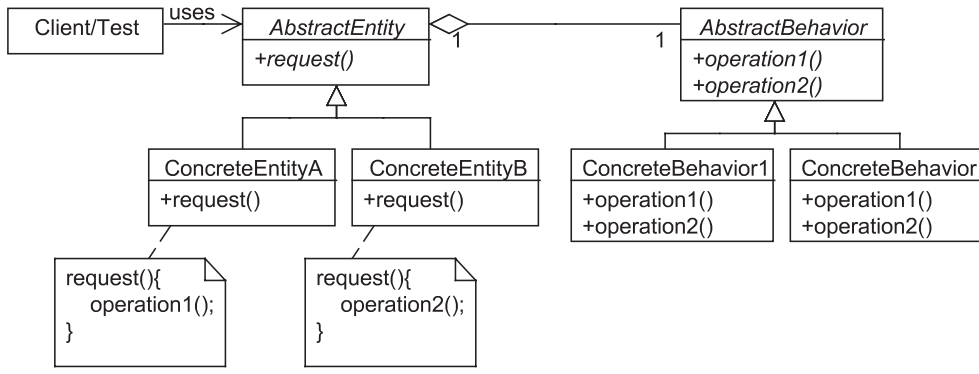


Figure B.7 Generic example of a Bridge implementation

The Bridge can be tricky, so Figure B.8 shows the same example using my non-software analogy.

Note that Andrea has to take some extra actions because `getSalad()` is not reliable across all restaurants. Because she is isolated from everyone else, only she has to be so high-maintenance. ☺

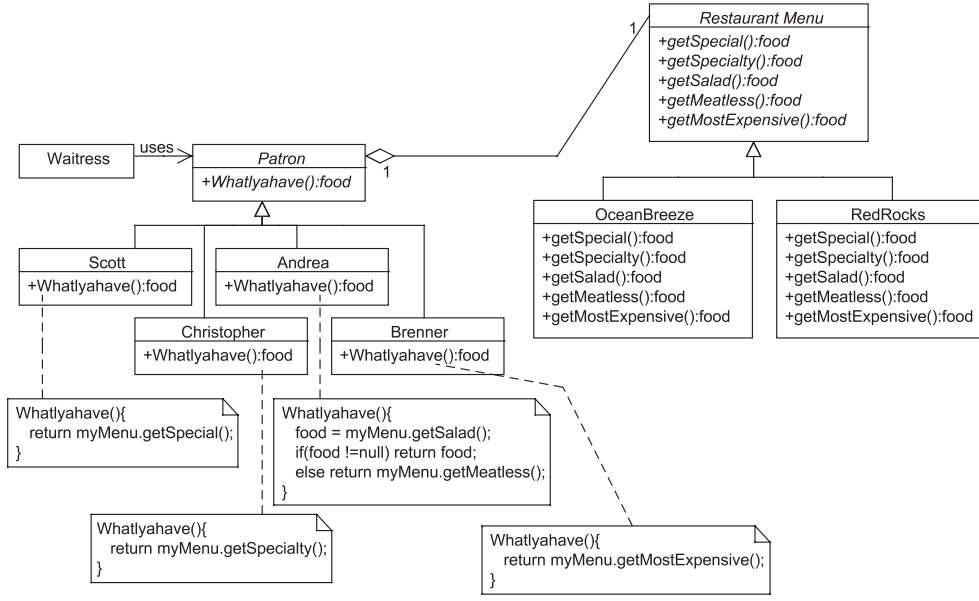


Figure B.8 Restaurant example with non-software analogy

Code

```
public class AbstractEntity {
      protected AbstractBehavior myBehavior;
      public AbstractEntity(AbstractBehavior aBehavior) {
            myBehavior = aBehavior;
      }
      public abstract void request();
}

public class ConcreteEntityA extends AbstractEntity {
      public ConcreteEntityA(AbstractBehavior aBehavior) {
            superclassConstructor(aBehavior);
      }
      public void request() {
            myBehavior.operation1();
      }
}

public class ConcreteEntityB extends AbstractEntity {
      public ConcreteEntityB(AbstractBehavior aBehavior) {
            superclassConstructor(aBehavior);
      }
      public void request() {
            myBehavior.operation2();
      }
}

public abstract class AbstractBehavior {
      public abstract void operation1();
      public abstract void operation2();
}

public class ConcreteBehaviorA extends AbstractBehavior {
      public void operation1() {
            // Behavior 1A
      }
      public void operation2() {
            // Behavior 2A
      }
}

public class ConcreteBehaviorB extends AbstractBehavior {
      public void operation1() {
            // Behavior 1B
      }
      public void operation2() {
            // Behavior 2B
      }
}
```

Questions, Concerns, Credibility Checks

- How likely is it that all of the entities can share the same interface without giving up any key capabilities?
- How likely is it that behaviors can share the same interface without giving up any key capabilities?
- How consistently do the entities use the behaviors? If they are very orthogonal, the interface of the `AbstractBehavior` will tend to be broad.
- How likely is it that new entities that may be added later will be able to use the existing `AbstractBehavior` interface? If this is unlikely, the interface will tend to grow, and perhaps bloat over time.
- Are the behaviors likely to be stateful or stateless? If they are stateless, they can be shared to increase efficiency and performance.

Options in Implementation

Because the Bridge requires that all behaviors must share the same interface and because it is quite common for these behaviors to come from foreign classes (device drivers, APIs from other systems, and so on), Bridges often use Adapters and Façades to bring the behaviors to a common interface. See Figure B.9 for an example.

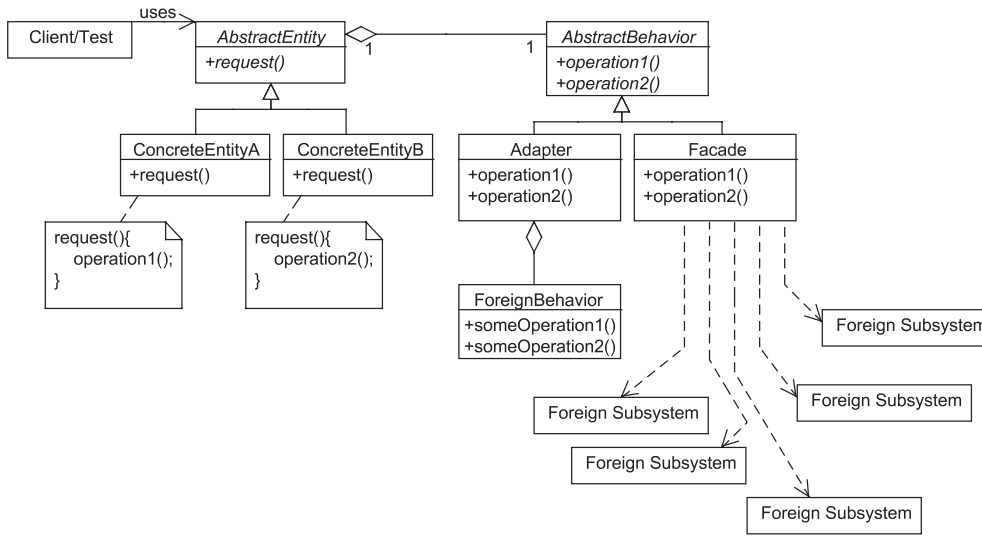


Figure B.9 Bridges using Adapters and Façades

Consequent Forces

Testing Issues

As shown in Figure B.10, the behavior classes are probably testable on their own (unless they are Adapters and/or Façades, in which case, see the testing forces accompanying those patterns). However, the entity classes are dependant upon behaviors and so a mock or fake object can be used to control the returns from these dependencies. It can also check on the action taken upon the behavior by the entity, if this is deemed an appropriate thing to test.

Cost-Benefit (Gain-Loss)

- The Bridge creates flexibility because the entities and behaviors can each vary without affecting the other.
- Both the entities and behaviors are open-closed, if we build the bridge in an object factory, which is recommended.

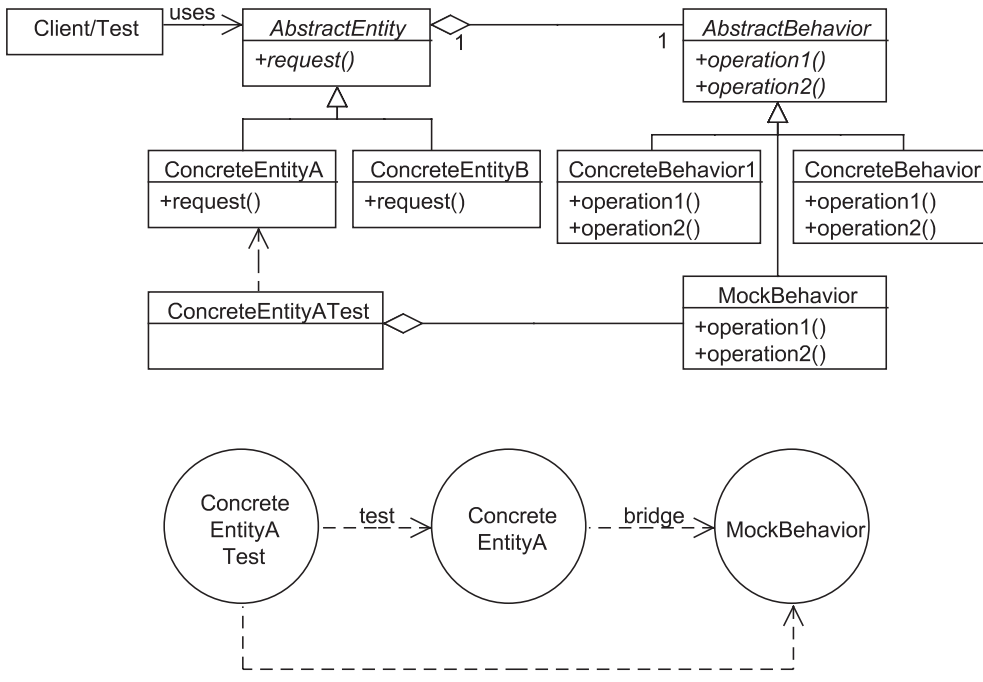


Figure B.10 Behavior classes testable on their own

- The interface of the behavior can require changes over time, which can cause maintenance problems.
- The delegation from the entities to the behaviors can degrade performance.

The Chain of Responsibility Pattern

Contextual Forces

Motivation

Where there are a number of different entities that handle the same request, and where only one of them will be the correct entity in a given circumstance, we can decouple the client (requester) from the entity that will handle the request in a given case.

Encapsulation

The client cannot see

- How many different entities there are that may handle the request (cardinality)
- Which entity actually handles any given request (variation)
- The order that the entities are given their chance (sequence)
- How the selection is achieved (selection)

These issues are all encapsulated by the pattern, especially if the chain itself is created in an object factory.

Procedural Analog

Each entity will be given *a chance* to handle the request until one of them elects to handle it. Once the request is handled, no further entities are given a chance to handle it. Any entity will self-select (or not), and therefore the selection is done from the point of view of the entity.

A clear procedural analog to this is the switch/case statement.

```
switch(condition) {

      case A:
      // Behavior A

      case B
      // Behavior B

      case C:
      // Behavior C

      default:
      // Default Behavior
}
```

The condition is evaluated from the point of view of each behavior, not from the point of view of the client or system (see the Strategy pattern).

Non-Software Analog

Imagine you have a large number of coins that you need to sort. A simple (old-fashioned) way of doing this is to create a stack of trays, each one with holes in it that match the size of a given coin (one for half-dollars, one for quarters, one for nickels, and so on).

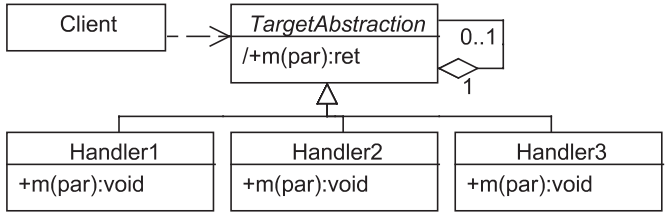
In this stack, as displayed in Figure B.11, the tray with the largest holes will be on top, the next largest will be underneath that, and so on until you have a tray with no holes at the bottom. Only the dimes (which have the smallest diameter) fall all the way to the bottom tray. The pennies (next smallest) are caught one tray up, and so forth.

Implementation Forces

Example

See Figure B.12.

Figure B.11 Coin-sorting tray design



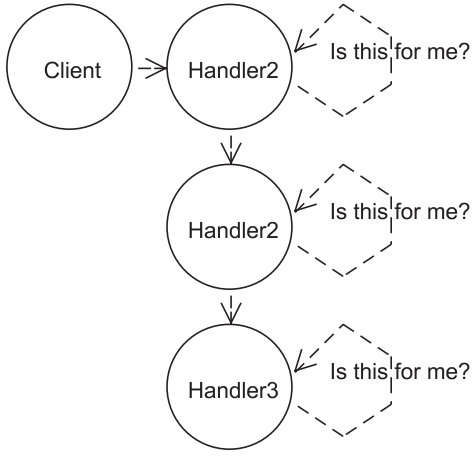


Figure B.12 Implementation forces

Code

```
public abstract class TargetAbstraction {
      private TargetAbstraction myTrailer;
      public TargetAbstraction(TargetAbstraction aTrailer) {
            myTrailer = aTrailer;
      }
      public ret m(par p) {
            if (isMyAction(p)) {
                  return myAction(p);
            }
            if(myTrailer != null {
                  return myTrailer.m(p);
            }
            throw new EndOfChainReachedException();
      }
      protected abstract bool isMyAction(par p);
      protected abstract ret myAction(par p);
}

public class Handler1 extends TargetAbstraction {
      public Handler1(TargetAbstraction aTrailer) {
            superclassConstructor(aTrailer);
      }
      protected bool isMyAction(par p) {
            // Decision logic to select Handler 1
      }
      protected ret myAction(par p) {
            // Handler 1 behavior
            return ret;
      }
}

public class Handler2 extends TargetAbstraction {
      public Handler1(TargetAbstraction aTrailer) {
            superclassConstructor(aTrailer);
      }
      protected bool isMyAction(par p) {
            // Decision logic to select Handler 2
      }
      protected ret myAction(par p) {
            // Handler 2 behavior
            return ret;
      }
}
```

```
public class Handler3 extends TargetAbstraction {
      public Handler1(TargetAbstraction aTrailer) {
            superclassConstructor(aTrailer);
      }
      protected bool isMyAction(par p) {
            // Decision logic to select Handler 3
      }
      protected ret myAction(par p) {
            // Handler 3 behavior
            return ret;
      }
}
```

Questions, Concerns, Credibility Checks

- All handlers must have a common interface. How difficult will this be to achieve without sacrificing any key distinction in one or more of the handlers?
- Is there significance in the order of the chain?
- Will different chain sequences affect performance in beneficial or harmful ways?
- How many handlers are there, as a performance issue?
- What should happen if none of the handlers elects? Is there a default case? Should an exception be thrown (as shown in the code example)?

Options in Implementation

As with the Decorator pattern, the Chain of Responsibility can be implemented with any desired collection. The advantage of the linked list is that it entirely encapsulates the handlers, and the selection of the correct handler under a given circumstance. However, a manager can be added to any collection in order to hide these issues.

Thus, a Chain of Responsibility can be implemented in an array, array list, vector, or any desired collection.

The Chain of Responsibility is also a good example of the capability of object-oriented designs to replace procedural logic with structures of objects. See the upcoming section, "Chain of Responsibility: The Poker Example," for more details.

Consequent Forces

Testing Issues

Testing each chain implementation is simply a matter of giving it the right and wrong state, to see if it responds by returning a correct value, or delegating to the next implementation. To test this second issue shown in Figure B.13, a mock or fake object should be used.

Cost-Benefit (Gain-Loss)

- The client is freed from containing the selection logic, strengthening its cohesion.
- The selection logic is spread out over many objects, and is therefore hard for the developer to see.
- Business logic regarding ordering dependencies is captured in the ordering of the chain rather than expressed in code (see the following poker example).
- If the chain is added to, it may get lengthy and may introduce performance problems.

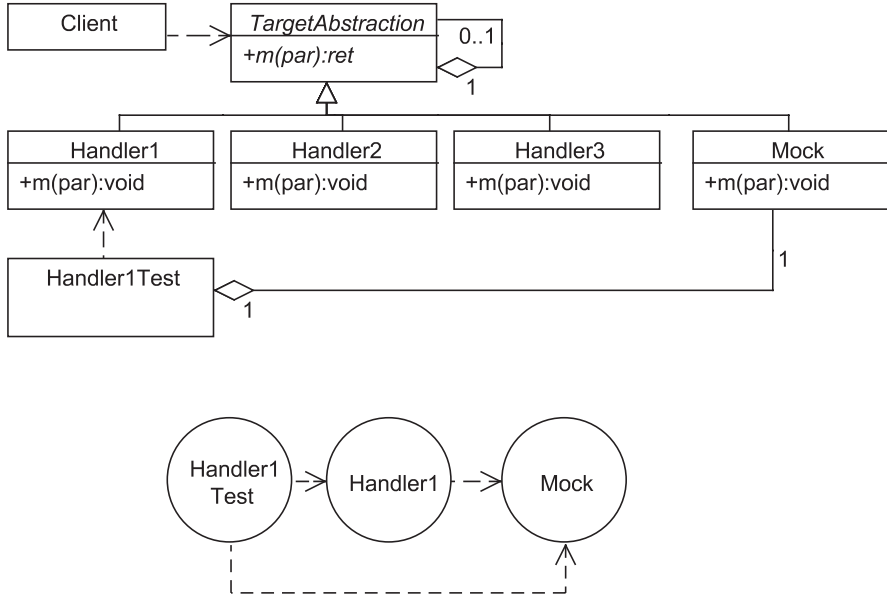


Figure B.13 Testing a chain implementation with a mock or fake object

Chain of Responsibility: The Poker Example

The Chain of Responsibility gives us an opportunity to demonstrate how various, sometimes surprising, issues can be composed into objects, rather than rendered into procedural code. This is not to say that *all* issues that were once dealt with procedurally should now be dealt with in an object-oriented way, but rather that it is helpful to know that such options exist, to allow the development team to make the most advantageous decision.

Our example here is the game of poker. If we were creating a software version of this game, one issue we would have to deal with is the rules that specify what hand beats what.

In poker, the hand with the highest card wins, unless one hand has a pair. The highest pair wins unless one hand has two pair. The hand with the highest "higher pair" wins unless one hand has three of a kind. Three of a kind is beaten by a straight (five cards in sequence, like 5, 6, 7, 8, 9). A straight is beaten by a flush (five cards in the same suit). A flush is beaten by a full house (three of one kind, a pair of another). A full house is beaten by four of a kind. Four of a kind is beaten by a straight-flush (five cards in sequence, all of the same suit), and the highest hand is a royal flush (the ace-high version of the straight flush).

These are business rules, essentially. Imagine that we decide to assign each of these hand types a numeric value so that once we determine the value of two or more hands, we could easily determine the winner, second place, and so forth. The Chain of Responsibility, as shown in Figure B.14, could be used to capture the rules that bound each hand type to a specific numeric value.

The `Hand` sends an array of its five cards to the first *evaluator* in the chain. Of course, we would likely build this chain in an object factory, and so the hand would not "see" anything more than a single service object.

The `Royal Flush Eval` object looks at these five cards and then determines yes or no, if they qualify as a royal flush. If the cards do qualify, the `Royal Flush Eval` object returns the numeric value we have assigned to `Royal Flushes`. If the cards do not qualify, the `Royal Flush Eval` delegates to the next object in the chain and waits for a result. It then returns that result to the `Hand`.

This delegation continues until one of the `Eval` objects self-elects, and then the value returns up the chain and back to the `Hand`. Note that every possible poker hand has some card that is highest, and so we have a default condition we can use to end the chain, the `High Card Eval`.

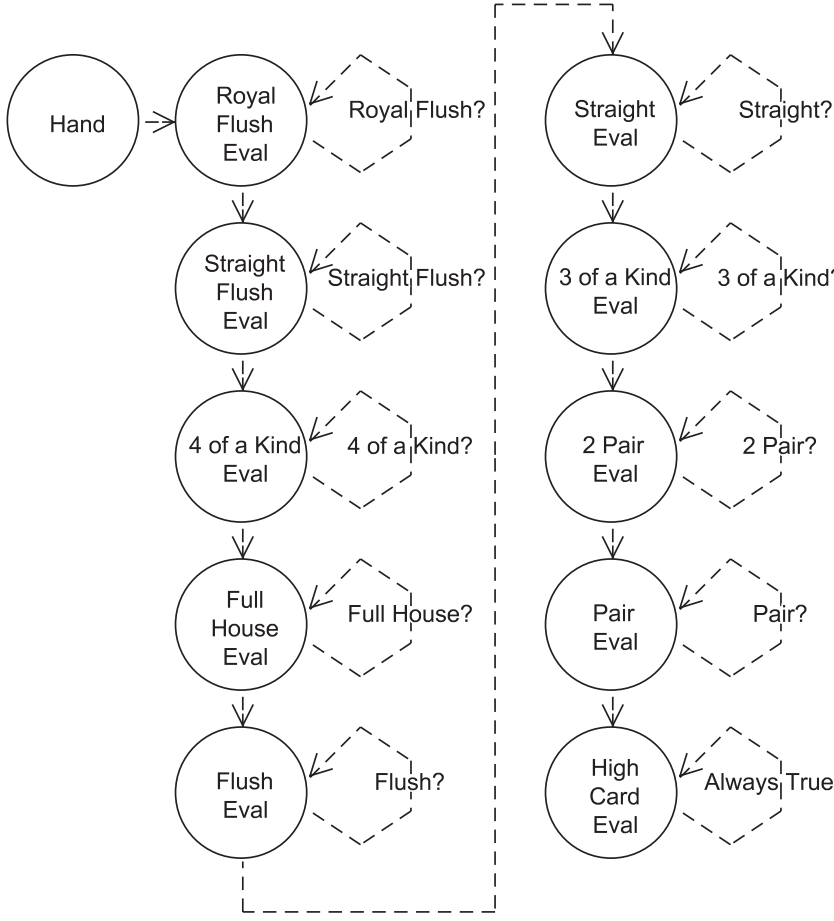


Figure B.14 The Chain of Responsibility

The key point here is this: A Royal Flush is also a Flush. It is also a Straight. If we hand five cards, which were in fact a Royal Flush, to the `Straight Eval` class, it responds in the positive: "Yes, that is a Straight." So, 3 of a Kind is also a pair, a Full House is also 3 of a Kind, and so forth.

There is a business rule about this: If you have a Royal Flush, no one can call your hand a Straight even though this is technically true. There is an ordering dependency about these rules: After you determine that a hand is a Full House, you do not ask if it is a Pair.

This design captures this set of dependencies, but not in conditional code. It is captured simply in the order of the evaluators in the chain, and can therefore be maintained simply by changing that order (there are many variations on Poker, as you may know).

The point is not to say that all procedural logic should be converted to object structures. This would create overly complex designs, would tend to degrade performance, and would challenge many developers who tried to understand the design. However, it is important to know that object structures are a possible solution to issues that we would normally think of procedurally, to enable us to make informed decisions.

The advantages to using an object structure include testability, encapsulation, and open-closed-ness.

The Composite Pattern

Contextual Forces

Motivation

Model simple and complex components in such a way as to allow client entities to consume their behavior in the same way. The Composite pattern captures hierarchical relationships of varying complexity and structure.

Terms:

- *Simple component*. A single class, also called a *leaf*.
- *Complex component.* A class that contains pointers to subordinate or "child" instances, and may delegate some or all of its responsibilities to them. These child instances may be simple or complex themselves. Also called a *node*.

Encapsulation

The difference between a simple (Leaf) and a complex (Node) component.

The structure of the composite relationship: a tree, ring, web, and so on (see the upcoming section, "Options in Implementation," for a discussion).

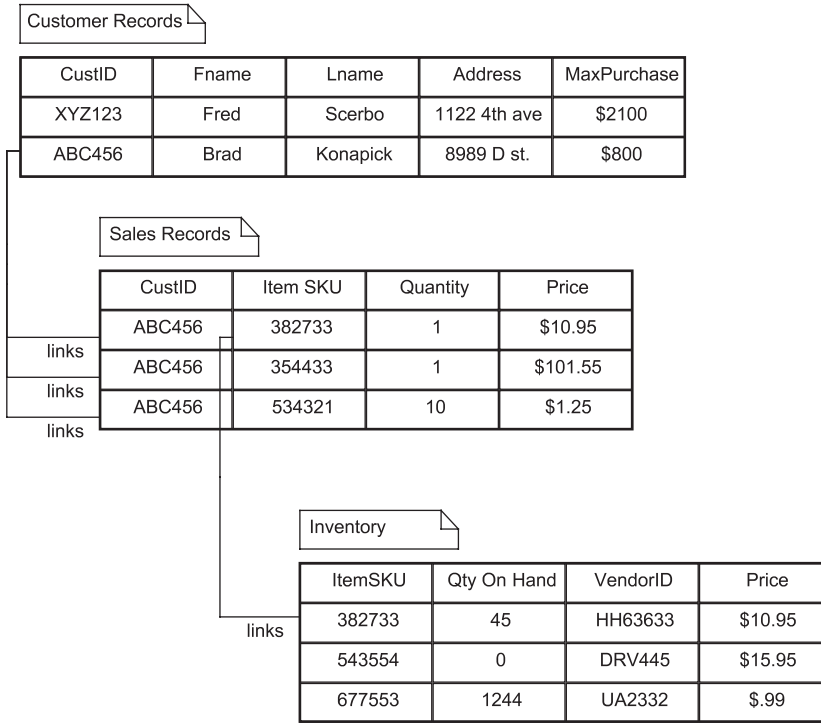


Figure B.15 Procedural analog

The behavior of the complex components: aggregating, best/worse case, investigatory, and so on (see the upcoming section, "Questions, Concerns, Credibility Checks," for a discussion).

Procedural Analog

In procedural systems, process and data are usually represented separately. In data structures, there is a clear analog to a Composite pattern in the way tables can contain foreign keys to other tables, allowing for variation in the depth of the data being represented. As with most procedural analogs, encapsulation is missing in Figure B.15.

Non-Software Analog

In the military, responsibilities flow from the top of a hierarchy down to the bottom in various ways. Let us assume that a Captain is assigning responsibilities for a number of tasks.

- She orders a Lieutenant to write up a set of reports for the Pentagon. This Lieutenant writes the reports himself with no further delegation.
- She orders another Lieutenant to create a training program for the upcoming maneuvers. This Lieutenant actually delegates the specifics to a set of Drill Sergeants and then assembles their work into a plan. He then reports back to the Captain.
- She orders a third Lieutenant to take hill 403. This Lieutenant delegates this responsibility to a set of Sergeants who delegate further to Corporals, Privates, and other resources (medics, an armor squad), each of whom may delegate further, in a very complex arrangement.

In each case, the Captain "orders" a single Lieutenant. Figure B.16 illustrates the variations in how these orders are accomplished are encapsulated.

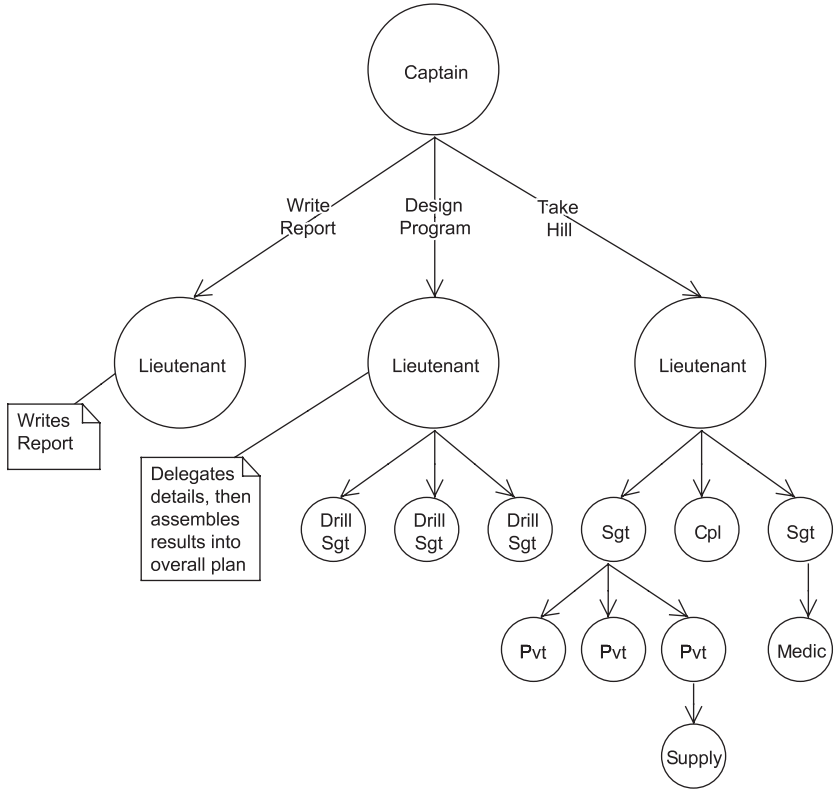


Figure B.16 Non-software analog

Implementation Forces

Example

See Figure B.17.

Code

```
public abstract class Component{
      public abstract void operation();
}

public class Leaf extends Component{
      public void operation() {
            // leaf operation goes here
      }
}

public class Node extends Component{
      private Component[] myChildren;
      public void operation() {
            foreach(Component Child in myChildren) {
                  child.operation();
            }
      }
}
```

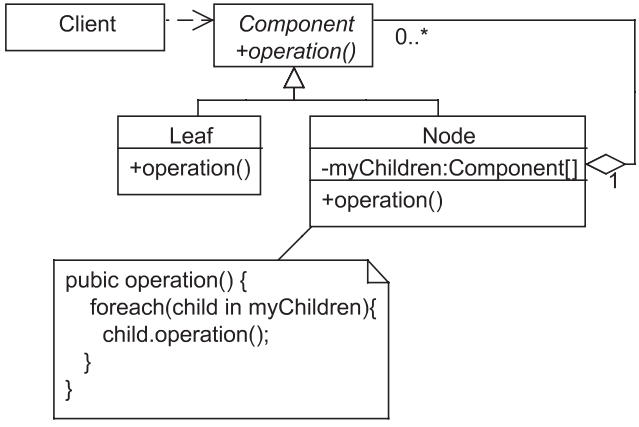


Figure B.17 Implementation forces

Questions, Concerns, Credibility Checks

There are many different forms of the Composite pattern, but at a high level they can generally be grouped into one of two categories: Bill of Material and Taxonomy.

A Bill of Material Composite captures the relationship that exists when a complex object is composed of many smaller objects, each of which may be further composed of other, yet smaller objects. For example, your car could be said to be composed of a body and a chassis. The body is further made up of fenders, doors, panels, a trunk, a hood, and so on. The chassis is further composed of a drive train and suspension. The drive train is still further composed of an engine, transmission, differential, wheels, and so on.

A Taxonomical Composite captures the logical relationships that exist between more and less general types. The biological phylum works this way. Living things are divided into Mammals, Amphibians, Fish, and Birds. Mammals are further subdivided into oviparous (egg-laying) and viviparous (live birth), and so forth.

This distinction is important because it gives us clues into the decisions we have to make when implementing the pattern (see "Options in Implementation").

If we are implementing a Taxonomical Composite, our motivation is almost certainly to allow a client entity to traverse the structure, to find information or classify an entity within the context of the existing structure.

If we are implementing a Bill of Material Composite, however, we have many options when implementing its behavior, each of which has different implications for encapsulation.

- *Aggregating behavior*. If we need to determine the weight of the car, for example, we need to aggregate the weights of all the subparts (which in turn need to aggregate their subparts and such). Often, this behavior can be hidden from consuming client entities in the way the Node component is implemented.
- *Best/worst case behavior*. If we need to determine the overall "health" of the car, the rule might be that the car is healthy if and only if all its parts are healthy. If any one part is in error, perhaps, we might say the entire car is in error. Thus, we would say the car is the worst case of all its parts. Again, this behavior can often be hidden from the consuming class in the way the Node component is implemented.
- *Investigatory.* If we need to determine something specific about a part or about a set of parts that meet a criteria (all the parts of the car that

come in contact with oil, for example), we might need to allow the client entity to "walk the tree of parts," looking for those that qualify. In this case, we may not be able to hide the specifics of the composite, much as we generally cannot in a Taxonomical Composite.

In any case, there is a key decision to make when creating a Composite: whether to put methods into the target abstraction that allow for the client entity to traverse the Composite. These *traversal methods* create coupling between the client and the components and should not be included unless they are needed.

In Taxonomy, traversal methods are almost always needed, since the purpose of the Composite is for the client to traverse it.

In a Bill of Material, traversal methods may or may not be needed, depending on the behavior desired.

Options in Implementation

The nature of traversal methods, if they are needed, reflects the nature of the Composite. The standard Composite, shown in Figure B.18, is the tree hierarchy.

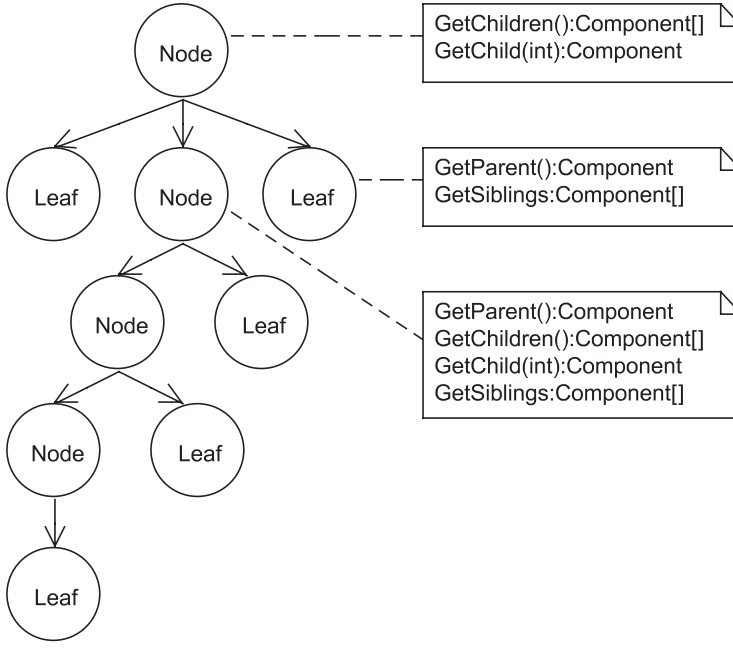


Figure B.18 The tree Composite

However, there are many other Composite structures possible. For example, Figure B.19 is also a Composite.

This is a ring and, as you can see, the traversal methods are different. Similar to this form of Composite is also the bi-directional ring, which allows for "next" and "previous" traversal (not shown).

Yet another Composite is a web, as shown in Figure B.20, which has yet again different traversal methods implied by its structure.

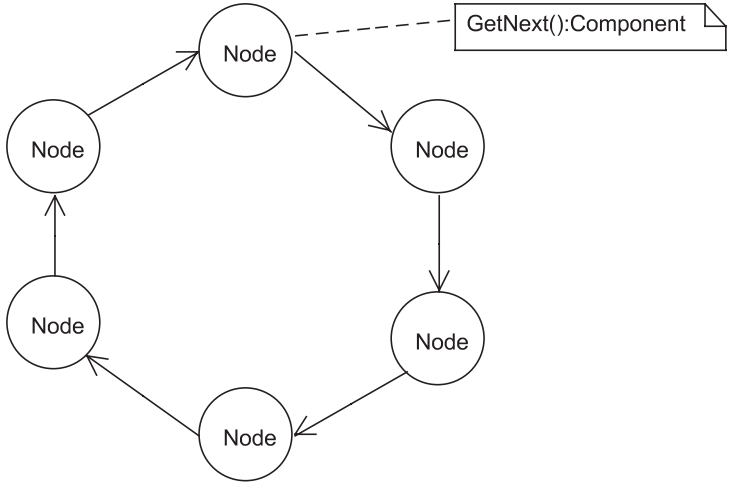


Figure B.19 A ring Composite

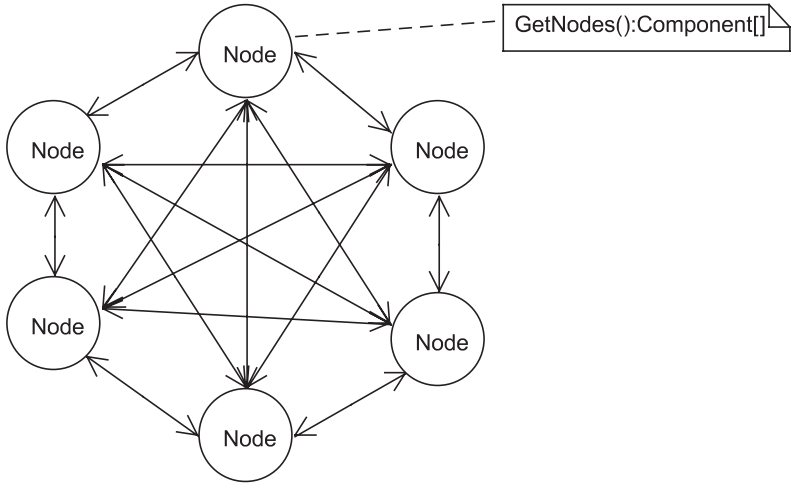


Figure B.20 A web Composite

The point is that the nature of the Composite structure is reflected in the traversal methods provided. Therefore, if traversal methods are provided, the fact that a Composite is a tree, ring, web, or any other structure is not encapsulated from entities that consume it. If a Composite were to begin as a tree, then later be changed to a ring, the client entities that interacted with it via these traversal methods would all have to be maintained.

Consequent Forces

Testing Issues

Testing Leaf objects is straightforward. However, testing Node objects should include verification that they interact as suspected with their subordinate instances at runtime. As displayed in Figure B.21, a mock or fake object can be substituted for these subordinate instances for testing.

Cost Benefit (Gain-Loss)

The client entity is freed from responsibilities that can be delegated to the Composite and therefore, the client entity is made simpler and more cohesive.

However, we tend to create very generic components, which can create difficulties when attempting to model very complex behavior.

The Composite is always a trade-off between these forces.

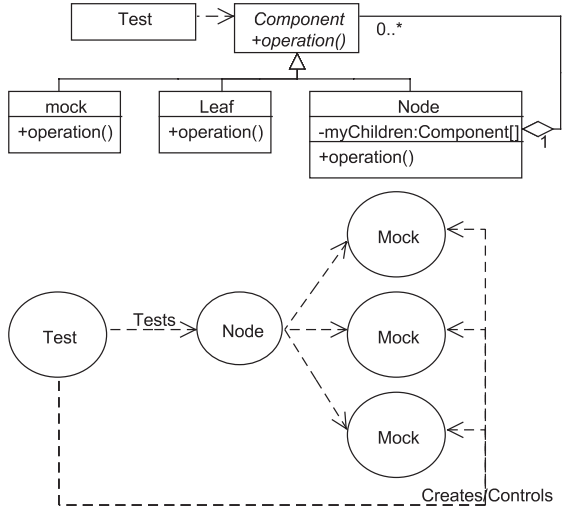


Figure B.21 Testing Node objects

The Decorator Pattern

Contextual Forces

Motivation

Add optional, additional behaviors to an entity dynamically. Ideally, the design should allow for a wide variety of combinations of these additional behaviors without altering the clients that consume the results.

Encapsulation

The number (cardinality) and order (sequence) of the optional behaviors. Ideally, the fact that a Decorator is being used can also be hidden from the client. In all the runtime variations shown in Figure B.22, the client takes the same action.

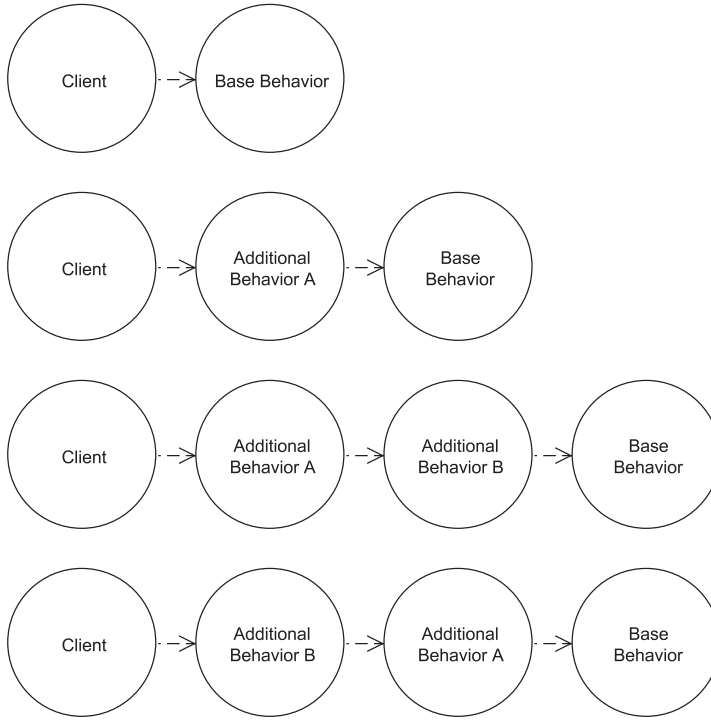


Figure B.22 Case 1: Base behavior only. Case 2: Base behavior and a single additional behavior. Case 3: Base behavior and multiple additional behaviors. Case 4: Base behavior and the same additional behaviors in a different sequence.

Procedural Analog

A "stack" of simple conditionals is analogous to one main aspect of the Decorator—that is, the capability to add one, many, all, or none of a set of optional behaviors.

```
//Non-optional behavior here

if(conditionA) {
      // Optional Behavior A
}

if(conditionB) {
      // Optional Behavior B
}

//Non-optional behavior here
```

This procedural approach does not allow for variations in the order of the behaviors, however. It is, therefore, only a partial analog.

Non-Software Analog

Those who have used a single-lens reflex camera (see Figure B.23) have experienced a design that is very much like the Decorator pattern.

Figure B.23 Single lens reflex camera

Such a camera has a basic behavior set that is required for it to fulfill its function in photography.

- It allows light to enter the camera through a lens.
- It focuses the light at a fixed point called the backplane where the film is held rigid.
- It opens the shutter for a controllable and reliable period of time.
- It advances the film.

The first step of this process can be altered without changing anything about the camera itself by using a filter, or any number of filters, that change the light before it enters the camera. The resulting photograph will be different, but the camera (and the outside world) remains unchanged.

The filters are designed to attach to the front of the camera lens. They have male threads on their trailing edges that mate to the female threads on the leading edge of the lens housing. However, they also have these female threads on their leading edges as well, so a filter can be attached to a lens or to another filter, allowing them to be stacked up. The overall effect on the resulting photograph is the accumulated effect of all the filters currently included in the stack.

A camera can take a photograph without any filters at all; it is sufficient by itself.

A filter cannot accomplish any kind of photograph alone. It must always be attached to something—either another filter or the camera lens. This is enforced in its design.

Implementation Forces

Example

See Figure B.24.

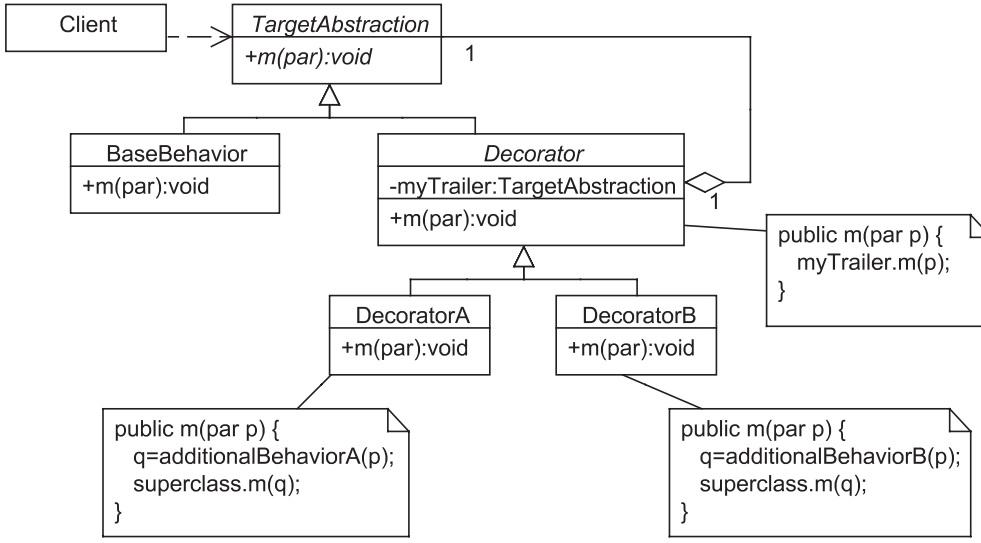


Figure B.24 Implementation forces

Code

```
public class TargetAbstraction {
      public abstract void m(par p);
}

public class BaseBehavior extends TargetAbstraction {
      public void m(par p) {
            // Base behavior
      }
}

public abstract Decorator extends TargetAbstraction {
      private TargetAbstration myTrailer;
      public Decorator(TargetAbstraction aTrailer) {
            myTrailer = aTrailer;
      }
      public void m(par p) {
            myTrailer.m(p);
      }
}
```

```
public class DecoratorA extends Decorator {
      public DecoratorA(TargetAbstraction aTrailer) {
            superclassConstructor(aTrailer);
      }
      public void m(par p) {
      // Decorator A behavior here
            superclass.m(p);
      }
}

public class DecoratorB extends Decorator {
      public DecoratorB TargetAbstraction aTrailer) {
            superclassConstructor(aTrailer);
      }
      public void m(par p) {
            // Decorator B behavior here
            superclass.m(p);
      }
}
```

Questions, Concerns, Credibility Checks

- Can the various Decorators and the base behavior share a common interface? If there are differences between them, how can this be resolved without sacrificing any key capabilities?
- What is the structure of the collection for the Decorators and the base behavior? The classic implementation uses a linked list (see preceding), but a Decorator can also be implemented using an array, a vector, or any suitable collection.
- Do the methods being decorated have a void return (aggregating behaviors in a pass-down the chain), or is there a return from them? If there is a return, each Decorator has two opportunities to add behavior, as the calls propagate down the chain, and as the returns propagate back up. (See the next section, "Options in Implementation," for more on this issue.)
- Can the Decorator be totally encapsulated? That is, can the client be designed to use any combination of decorators, and/or the base behavior, in precisely the same way? (See the next section, "Options in Implementation," for more on this issue.)

- Are there particular combinations of Decorators and/or particular sequences that should be prevented? If so, an object factory to build only the legitimate decorator chains is advised. In this case, if the restriction is highly critical, all concrete classes (everything apart from the `Client` and the `TargetAbstraction` in the preceding code) can be private inner classes of the factory.

Options in Implementation

An alternate form of the Decorator, shown in Figure B.25, uses a method or methods that have a return. This creates three different types of Decorators: those that act "on the way down," those that act "one the way back up," and those that do both.

These three types can then be implemented for different behaviors by subclassing. Also, a void return Decorator can be implemented with similar behavior, by coding some Decorators to simply wait until the next call returns before they take action. This behavior is simply easier to see and understand when we discuss it in terms of a nonvoid return.

Sometimes, one of the Decorators is used to add methods to the interface of the target abstraction. When this is done, the client is exposed to the presence of the Decorator, the specific decorating type, and the fact that this particular Decorator must be first in the chain.

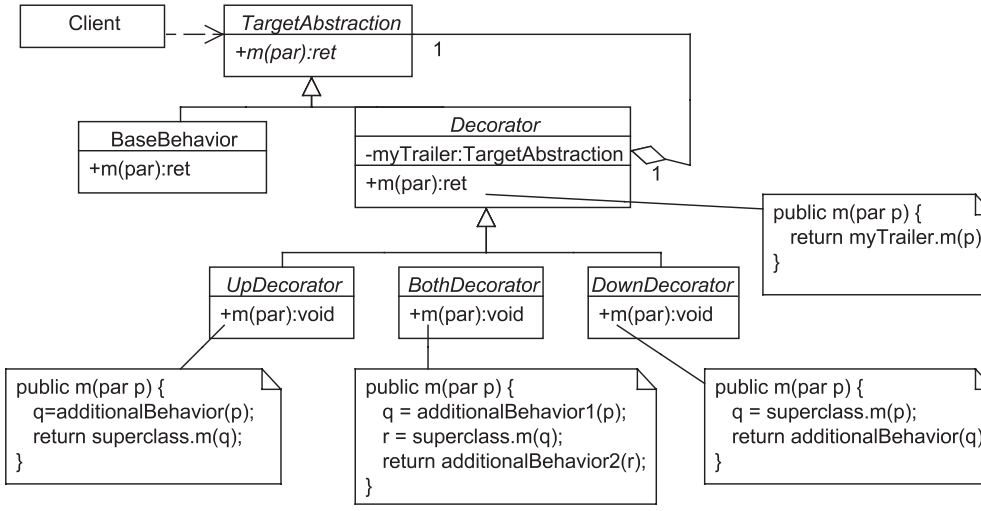


Figure B.25 The Bucket Brigade form of the Decorator pattern

An example of this is the way streaming I/O is done in Java. Figure B.26 is not literally accurate, but shows the general design of the I/O API in Java and similar languages.

A similar approach is taken to the `OutputStream` type. This creates an important bit of functionality in that it allows developers to use this API without requiring them to compose and decompose their data into bytes before using the IO capabilities of the framework. The Data version of the Decorator does this for them.

The downside is that the client must hold this Decorator in a downcast to its specific type, and therefore it cannot be encapsulated. Furthermore, the client must hold this reference directly (see Figure B.27).

The first case works, but the second does not, because the client has no reference to the `DataInputStream` instance, and therefore cannot use the expanded interface (it consequently provides no value).

In other words, the `DataInputStream` must be the first object in the chain. The design of the specific Decorator being used and the presence of the pattern are all visible to the client, and are therefore not encapsulated. This was considered a worthwhile trade-off given the importance of the Data interface.

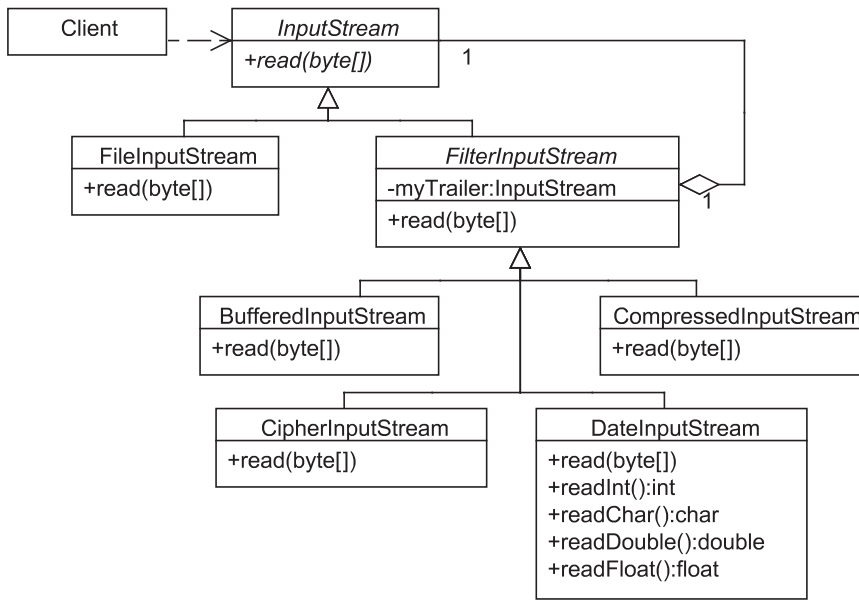


Figure B.26 Streaming I/O with Java

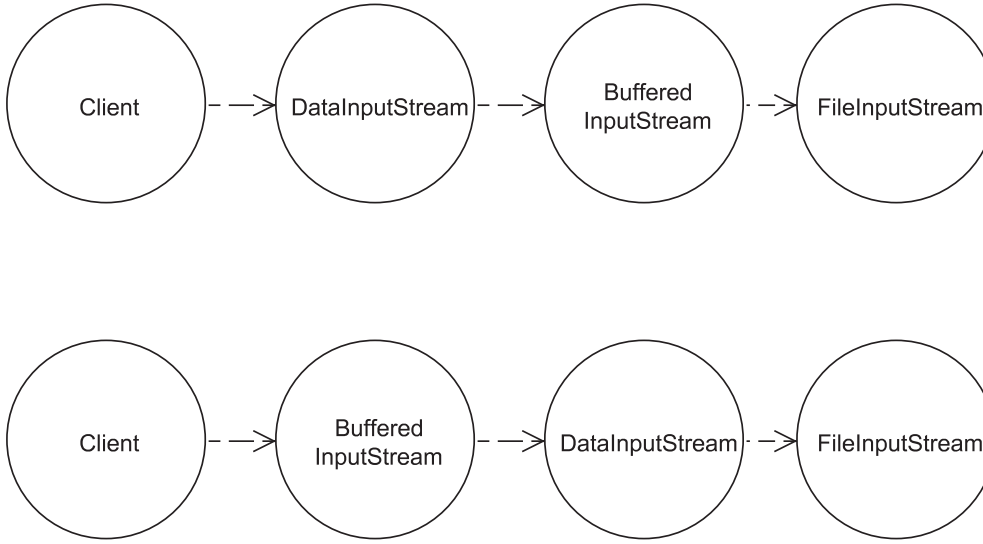


Figure B.27 Case 1 and Case 2 (with error)

However, if, we consider the patterns as *collections of forces* as we are suggesting here, there is another alternative. Reusing behavior with a changed interface is a contextual force of the Adapter pattern. If we think of our problem in this way, the following design, as shown in Figure B.28, comes to mind.

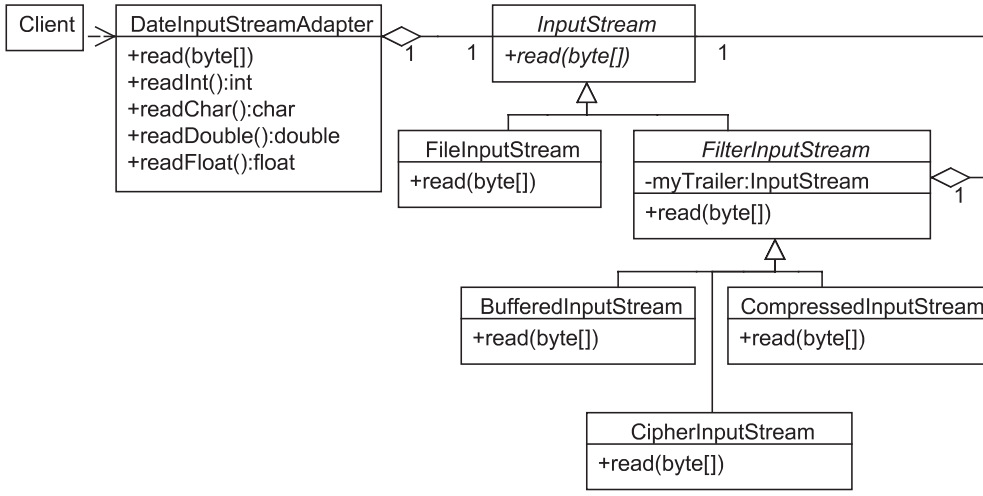


Figure B.28 Decorator with Adapter

This keeps the Decorator design encapsulated, and only exposes the Adapter, which of course the client would have to have visibility to (this is a force in the Adapter, after all).

Consequent Forces

Testing Issues

The base behavior can be tested in a normal way, but the Decorators impose a bit of a challenge in that they are designed to *decorate something*. If we want to test them in isolation from one another, we can use a mock or fake object to stand in place of the next object in the chain. Figure B.29 is an example of this. This allows us to inspect the effect of the decorator being tested by examining the state of the mock or fake object after the decorator behaves.

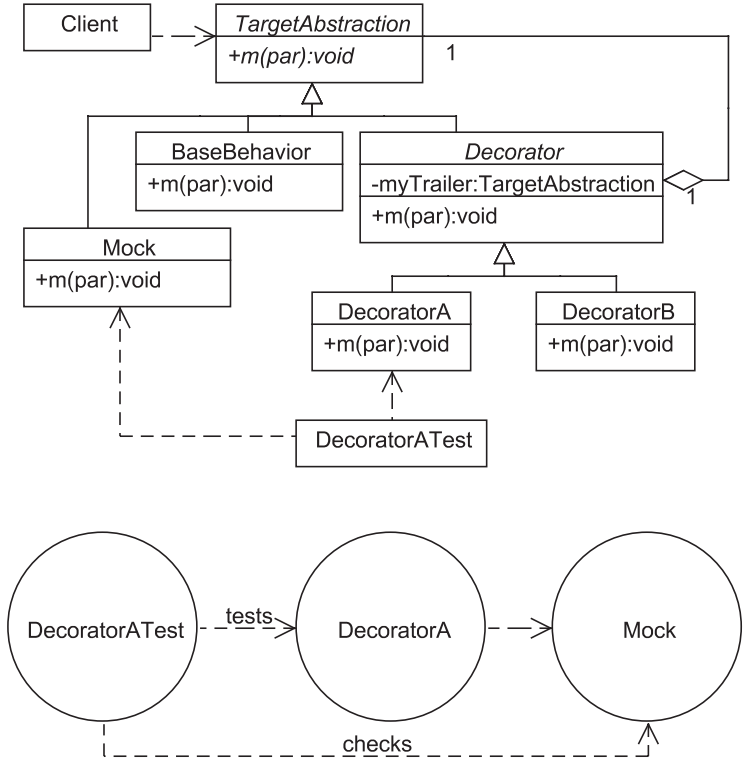


Figure B.29 Testing base behavior with mock objects

Cost-Benefit (Gain-Loss)

- The design is extremely flexible, allowing for different combinations of additional behavior, different numbers of behaviors, and different sequences to put them in.
- Avoids creating a feature-laden class. Features can be easily added with decorators.
- Promotes strong cohesion.
- New Decorators can be derived later, leaving the design very open-closed.
- The design can confuse developers who are not familiar with it.
- Some combinations of Decorators may be unwise, buggy, or violate business rules.

The Façade Pattern

Contextual Forces

Motivation

Provide a simplified, improved, or more object-oriented interface to a subsystem that is overly complex (or may simply be more complex than is needed for the current use), poorly designed, a decayed legacy system, or otherwise inappropriate for the system consuming it. Façade allows for the reuse of a valuable subsystem without coupling to the specifics of its nature.

Encapsulation

The subsystem's nature (complexity, design, decay, object-oriented, procedural nature, and so forth).

Optionally, the presence of the subsystem itself. See the upcoming section, "Questions, Concerns, Credibility," for a discussion.

Procedural Analog

Often, when creating large procedural systems, we create a *gateway routine* as a sort of API that allows the aggregation of many different parts of the system into a single called routine. This gateway routine makes the system easier to use and is essentially the precursor to a Façade in an object-oriented system. It is not strictly an analog; it is in fact an early version of the pattern.

Non-Software Analog

A hotel concierge (at an upscale hotel) stands between the hotel guest and all the various services that are needed to make for a successful, satisfying stay at the hotel.

The guest has goals, which are expressed from the guest's point of view. The concierge translates these goals into needed interactions with the concrete services needed to achieve them.

For example, the guest may say, "We would like to go out to an evening dinner, and a show, and then return to the hotel for an intimate dessert in our room."

The concierge, as shown in Figure B.30, handles all the nitty-gritty details. She arranges for a taxi, restaurant reservations, theatre tickets, housekeeping prep of the room, the kitchen preparing the dessert, and room-service delivery, among other responsibilities.

The concierge here in Figure B.30 is the Façade analog.

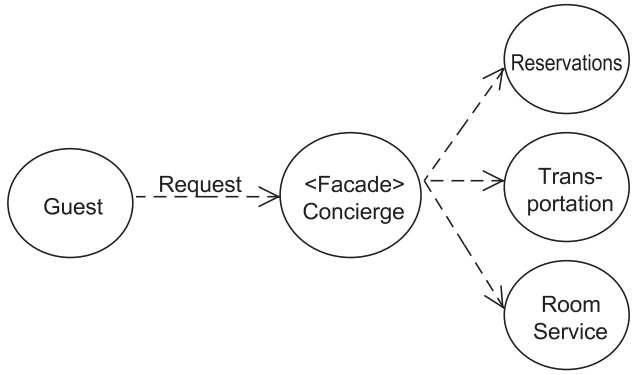


Figure B.30 The concierge as a Façade analog

Implementation Forces

Example

See Figure B.31.

Code

```
public class Facade {
      public void m1() {
            // make all calls into the existing system,
            // hiding the complexity
      }
      public string m2() {
            // make all calls into the existing system,
            // converting the return
            return rval;
      }
}
```

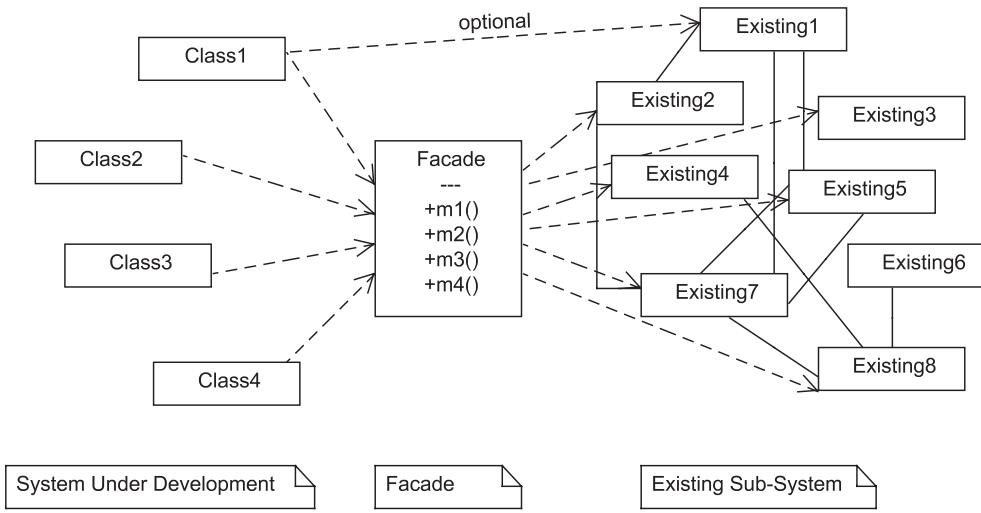


Figure B.31 Implementation forces

Questions, Concerns, Credibility Checks

Can we totally encapsulate the subsystem? The Façade can be a convenience service or a constraining service.

- *Convenience.* The system under development can "go around" and use the existing subsystem directly for infrequent, unusual, or orthogonal needs. This keeps the interface of the Façade relatively small and cohesive but couples the system under development to the subsystem. The call marked "optional" in the diagram shown in Figure B.31 shows this behavior.
- *Constraining.* The system under development must use the Façade for all accesses to the existing subsystem. This may broaden the interface of the Façade and weaken its cohesion, but it makes the subsystem itself completely swappable. Because the subsystem is hereby encapsulated, we often refer to such a Façade as an *Encapsulating Façade*.

Can we make the Façade stateless? Façades are often fairly heavyweight and instantiating them can impact performance. Therefore, if they can be made stateless, they can be implemented as Singletons, which alleviates the performance and resource impact of the Façade.

Options in Implementation

Façade is often implemented using other patterns

In fact, the Façade is not really a specific design but rather a role that is implemented in whatever way is considered most appropriate given the forces in the problem.

For example, Figure B.32 shows a Façade implemented using a Chain of Responsibility.

To keep a Façade stateless, often it is advisable to externalize the state into a lightweight object (see Figure B.33).

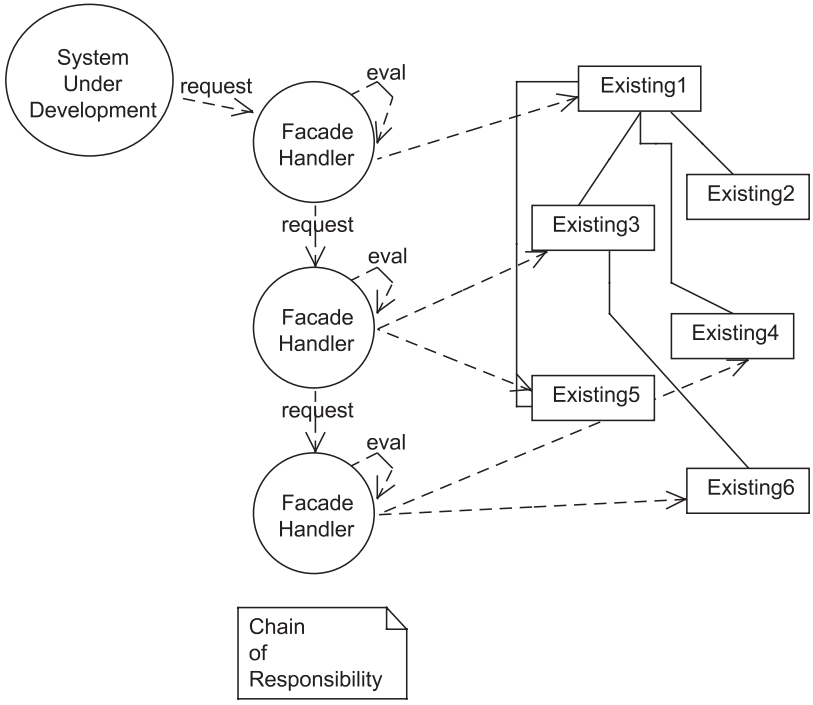


Figure B.32 A Façade implemented using a Chain of Responsibility

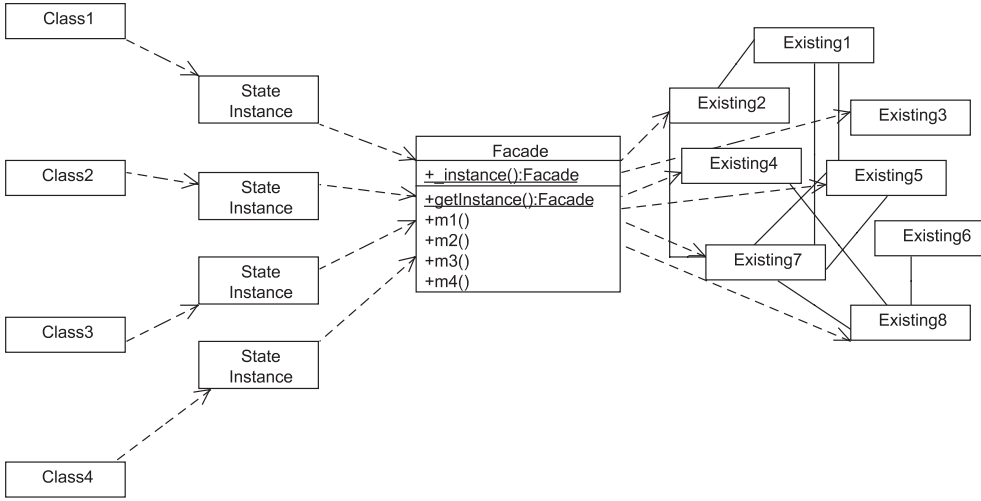


Figure B.33 Externalizing the state into a lightweight object

Legacy conversion

Façades can also have a strong role in converting legacy systems incrementally to more modern software.[2]

"Strangling" a Legacy System

Imagine an old, decayed, and poorly implemented legacy system. You have probably had to work with one or more of these in your practice. We often think of them as shown in Figure B.34, a "big ball of mud," because they are incomprehensible, brittle, and dense.

Job one is to stop using the system in the old way when writing new code that has to interact with it. If we could, we would stop here and refactor or rewrite the system entirely, but often that is simply not an option: We have to live with it, at least for now.

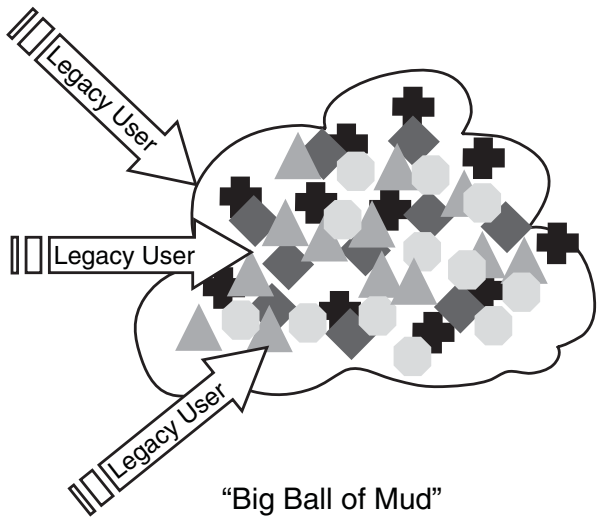


Figure B.34 Poor legacy system

2. This example is drawn from the work of Martin Fowler. I came across it while reading Brian Marick's work on testing, reflected in his blog, "Exploration Through Example" (http://www.testing.com/cgi-bin/blog/2005/05/11).

However, using the old system directly (as the legacy users do) means writing code that is similar in nature to the code in the system. At the very least, we want our new code to be clean, tested, and object-oriented. So, we create a Façade with the interface that we wish we had (see Figure B.35) and then delegate from there into the various routines in the legacy system.

Now that we have this in place, the new users can be developed more cleanly and with higher quality. Figures B.36, B.37, B.38, and B.39 all show that now, over time, and without significant time pressure on the project, we can incrementally pull out behaviors from the legacy system into more clean, tested, object-oriented code, and delegate from the old routines into the new ones.

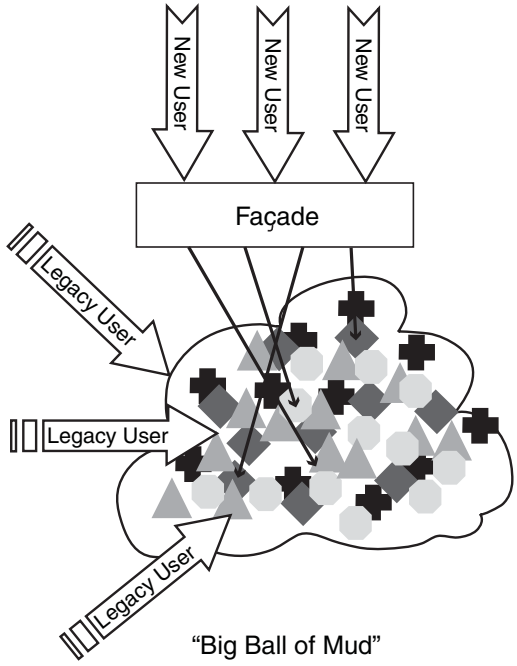


Figure B.35 Interface we wish we had

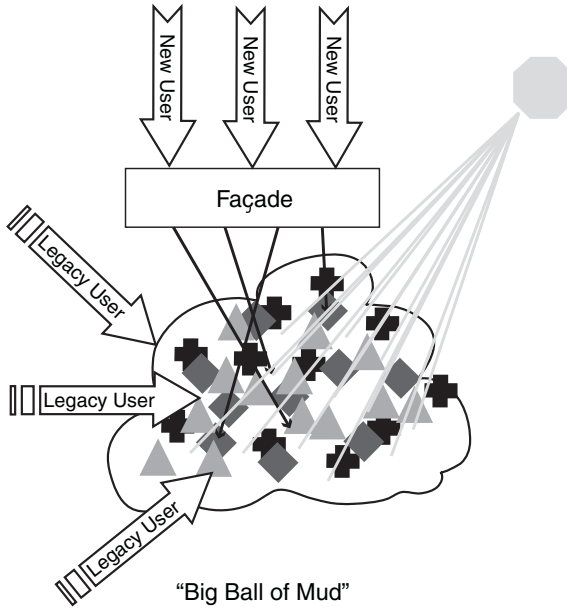


Figure B.36 New users developed cleanly with high quality

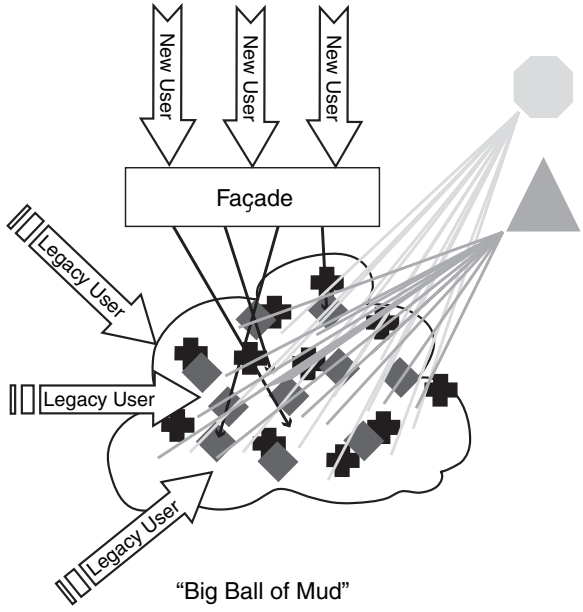


Figure B.37 Another new user developed cleanly with high quality

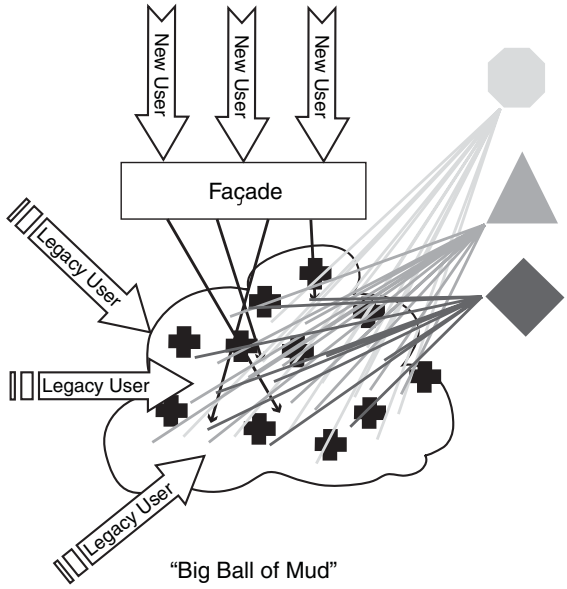


Figure B.38 A further increment in new development

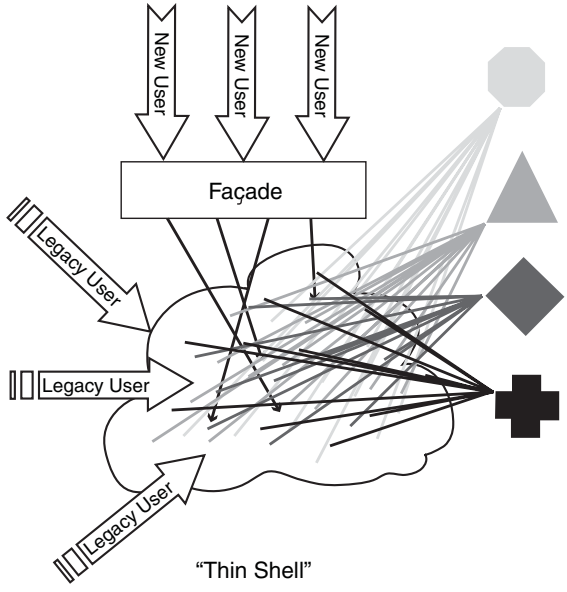


Figure B.39 A thin shell

After we reach the point that the legacy system remains only as a series of delegation points and nothing but a thin shell, as shown in Figure B.39, around the new behaviors, we can begin to refactor the legacy users to use the Façade instead of the delegating routines in the older system. As Figure B.40 shows, the Façade itself can also be refactored to use the new code directly. Again, this can be done incrementally, as time permits, without a lot of pressure. Naturally, after all of the legacy users and the Façade no longer use the thin-shell legacy system, as shown in Figure B.41, it can be retired.

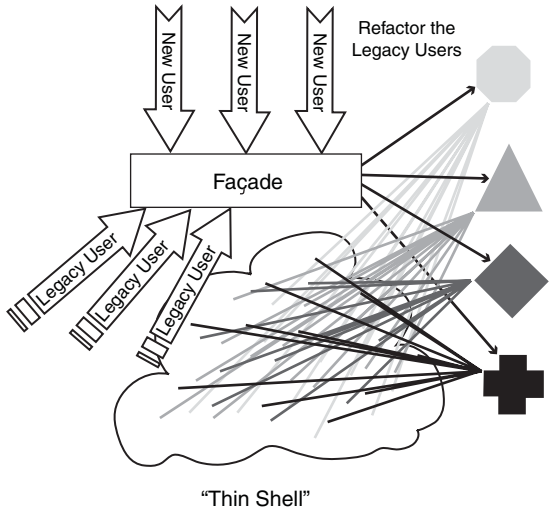


Figure B.40 Refactoring the legacy users and a thin shell

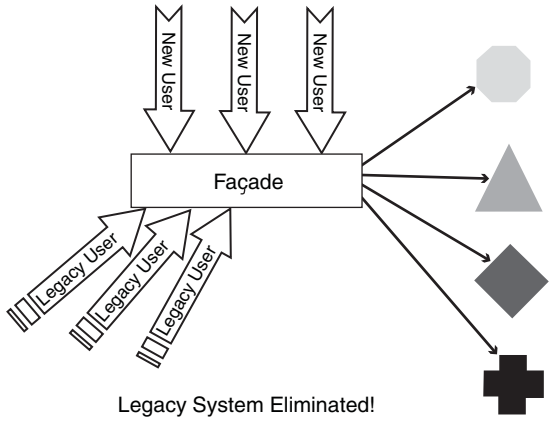


Figure B.41 Legacy system eliminated

Consequent Forces

Testing Issues

Façades are inherently difficult to test if (as is common) the subsystems that they cover are hard to test. However, the presence of the Façade makes the system under development fundamentally easier to test, as shown in Figure B.42, because the Façade, once in place, can be mocked or faked to eliminate unpredictable dependencies.

Cost-Benefit (Gain-Loss)

A Façade keeps the client system clean, reduces complexity when accessing existing subsystems, and can allow for the reuse of valuable legacy systems in a new context without reproducing their problems.

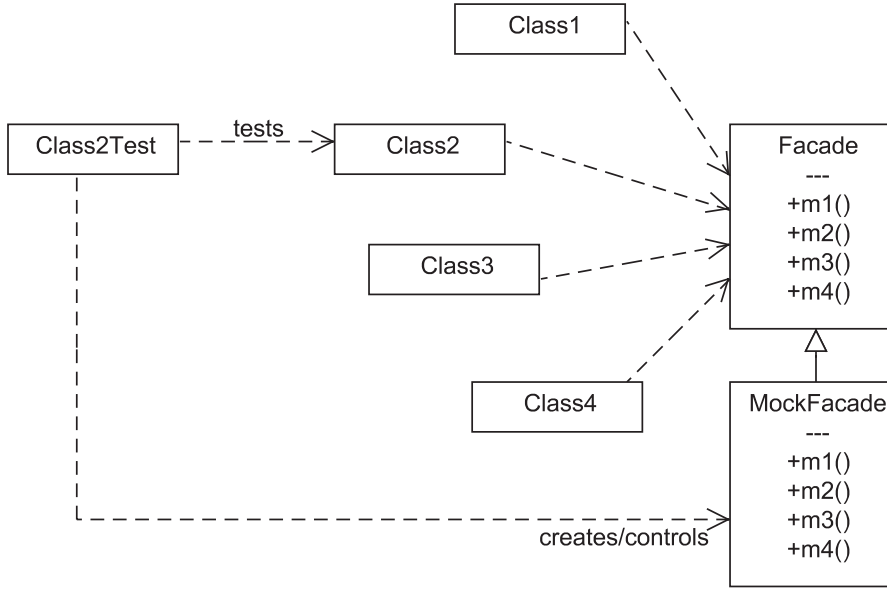


Figure B.42 System under development easier to test

An Encapsulating Façade decouples the client system entirely from the subsystem. This means that the subsystems can be swapped, eliminated, crippled, and so on. This allows for easy creation of

- *Demonstration versions of software*. A demo version of the Façade can create stubbed-out behavior for the subsystem, and stand in its place, allowing the frontend system to be demonstrated without the backend software/hardware present.
- *Crippled versions of software*. A crippling Façade can create a limited-functionality version of a system, which can easily be upgraded to the full version by replacing the crippling Façade with the full Façade.
- *Testing*. The Façade can be mocked or faked (see the preceding section, "Testing Issues") making the client system testable, which would otherwise be much more difficult.
- *Training versions of software*. Similar to Demos, a training Façade can be used to create predictable scenarios for training while allowing the students to interact with the "real" frontend of the overall system.

Façade and N-Tier Architecture

Imagine that we are working on an application with a three-tiered architecture (UI, Middle, Data), and we want to work in a test-driven way. Unfortunately, our teams are all blocked.

- *UI Tier*. I cannot start until the middle tier is done; they provide the interface for the services I will consume.
- *Middle Tier*. I cannot start until the data layer is done, because I need to access it in order to provide services to the UI.
- *Data Layer*. I cannot start until I know what the data needs are. Also, I am involved in day-to-day business, which must always take priority over new development. I will get to it when I get to it.

We can use Façades to separate these layers. The Façades present the interface of the consuming layer, and later can be coded to the implementation of the service layer in each case, when this becomes available (it can be stubbed out, initially). Also, each Façade is a mockable entity, and so, as in Figure B.43, the layers can be test-driven.

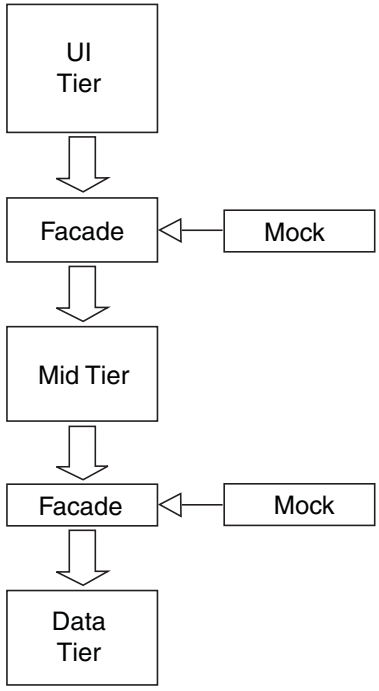


Figure B.43 Each Façade as a mockable entity

The Proxy Pattern

Contextual Forces

Motivation

We want to add an optional behavior to an existing class. This new behavior may protect the class, log access to it, or delay its behavior or instantiation, or any other single additional behavior. We also need to be able to use the existing class at times without this additional behavior.

Encapsulation

It is hidden from the client whether the additional behavior is in place at any given point at runtime. We may also want to hide the fact that an optional behavior exists at all.

Procedural Analog

A simple condition to include or exclude an optional behavior in the midst of non-optional behavior:

```
//Non-optional behavior here

if(condition) {
      // Optional Behavior here
}

//Non-optional behavior here
```

Non-Software Analog

I use a handheld hose to water my garden. A few times a year, I add a container between the hose fitting and the watering head that puts plant food into the water as it passes through (see Figure B.44).

I take no different action when watering in this way, and yet there is additional behavior, like feeding the plants. Also, various different plant foods could be introduced in this way, and again I would take no different action at watering time.

Figure B.44 Handheld hose with container for plant food

Implementation Forces

Example

See Figure B.45.

Code

```
public abstract class TargetAbstraction {
      public abstract ret m(par p);
{

public class BaseBehavior extends TargetAbstraction {
      public ret m(par p) {
            // unproxied, or base behavior
      }
}

public class Proxy extends TargetAbstraction {
      private BaseBehavior myBaseBehavior;
      public Proxy(BaseBehavior aBaseBehavior) {
            myBaseBehavior = aBaseBehavior;
      }
      public ret m(par p) {
            // Can add behavior before the base behavior
            myBaseBehavior.m(p);
            // Can add behavior after the base behavior
            return ret;
      }
}
```

Questions, Concerns, Credibility Checks

- The proxy and the `BaseBehavior` must have the same interface, so the client does not have to take any additional or different action when the proxy is in place.
- If the `BaseBehavior` preexists, how do we get existing clients to use the proxy, when appropriate, instead of the `BaseBehavior`?
- How does the proxy get the reference to the `BaseBehavior`? In the code example, the reference is passed in, but an object factory or other encapsulating entity can be used as well.

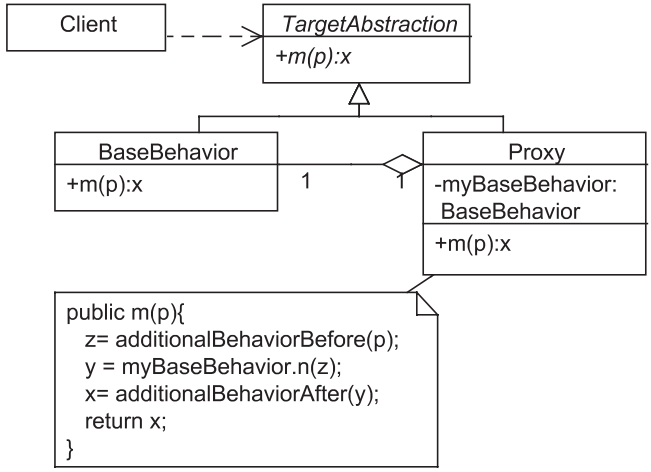


Figure B.45 Implementation forces

Options in Implementation

The preceding form uses delegation from the proxy to the `BaseBehavior` class to reuse the preexisting behavior. Another form of the proxy, as shown in Figure B.46 (called the *class Proxy*), uses inheritance instead.

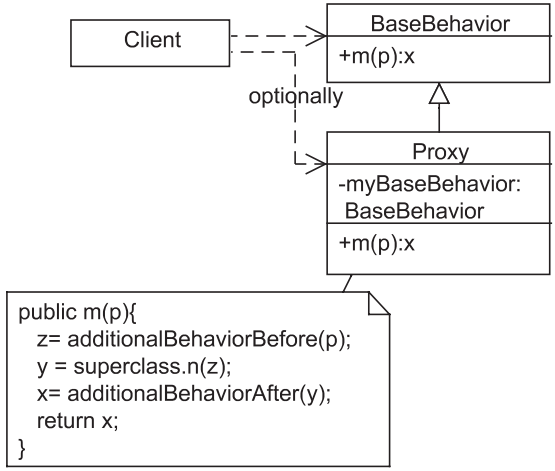


Figure B.46 The class Proxy that uses inheritance instead

There are a number of different kinds of proxies. They are named for the type of behavior that they add to the class being proxied. Only some proxies are possible with the class Proxy form but all of them are possible with the form that uses delegation.

Here are some examples of proxies:

- *Logging Proxy*: Logs all call to the method of the original class, either before or after the base behavior, or both.
- *Protection Proxy*: Block access to one or more methods in the original class. When those methods are called, the Protection Proxy may return a null, or a default value, or throw an exception, and so on.
- *Remote Proxy*: The proxy resides in the same process space as the client object, but the original class does not. Hence, the Proxy contains the networking, piping, or other logic required to access the original object across the barrier. This cannot be accomplished with a class Proxy.
- *Virtual Proxy*: The proxy delays the instantiation of the original object until its behavior is called for. If its behavior is never called for, the original class is never instantiated. This is useful when the original class is heavyweight and/or there are a large number of instances needed. The Virtual Proxy is very lightweight. This cannot be accomplished with a class Proxy.
- *Cache Proxy*: The proxy adds caching behavior to an object that represents a data source.

Consequent Forces

Testing Issues

The proxy is specifically designed to delegate to the original class for all the base behavior, and thus the test should test this delegation as well as the correct addition of behavior to the base behavior. As shown in Figure B.47, this can be accomplished by replacing the base or original class with a mock or fake object.

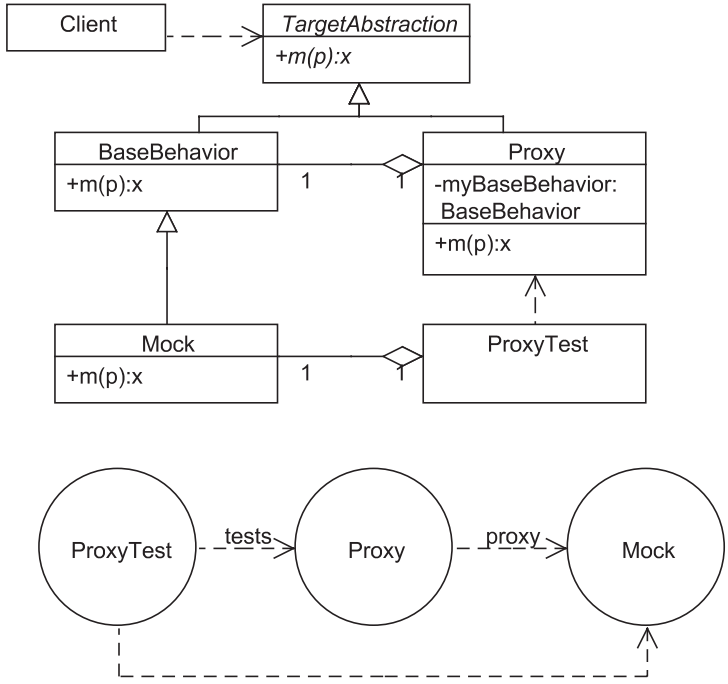


Figure B.47 Proxy designed to delegate to original class for all base behavior

Cost-Benefit (Gain-Loss)

- Proxies promote strong cohesion.
- Proxies simplify the client object and the object being proxied (by hiding complex issues like remoting and caching, and so on).
- If the instantiation of all classes is encapsulated by policy, inserting a proxy at a later time is significantly easier.
- Proxies often evolve into Decorators when multiple additional behaviors are needed. Knowing this, you do not have to introduce the Decorator until it is needed; this helps you to avoid over-design and analysis paralysis.

The Singleton Pattern

Contextual Forces

Motivation

Allow for one and only one instance of a given class to exist. Provide a mechanism to obtain this instance that any entity can access.

Encapsulation

- The construction of the instance
- The fact that the number of instances is constrained
- The mechanism by which the instantiation is made threadsafe, if it is
- Optionally, the destruction of the instance

Procedural Analog

Since the Singleton is concerned with instantiation, which is not an issue in procedural software, a direct analog is difficult to determine. However, the Singleton serves as a similar role to a global variable or function, and in fact can be used to eliminate the need for globals. Therefore, wherever you might consider the need for a global, you might consider the possibility of using a Singleton.

Non-Software Analog

A radio station broadcasts at a specific frequency and within a given context (a market). No other station can broadcast at this same frequency.

Any radio in the same context can receive the program currently being broadcast by tuning to the frequency assigned to the station in question. No matter how many radios are tuned to this station, there is and can be only one broadcast at a given point in time on it.

The number of radios tuned to the station is unconstrained, and thus the station is not designed for any particular number of listeners. The radios in the market can tune in the station by themselves, without any interaction with a controlling entity.

Radio stations and radios interact through an established protocol, which is part of their design.

Implementation Forces

Example

See Figure B.48.

Code

```
public class Singleton{
      private static Singleton _instance;
      private Singleton(){}
      public static Singleton getInstance(){
            if(_instance == null) {
                  _instance = new Singleton();
            }
            return _instance;
      }
}
```

Questions, Concerns, Credibility Checks

The example as shown is not threadsafe. If `Singleton` is not reentrant and is deployed in a multithreaded environment, a threadsafety option must be chosen. See "Options in Implementation" for various ways of accomplishing this.

Does `Singleton` have read-write state? This should be avoided if possible as this is essentially the creation of a global variable in the system. Global variables are to be avoided because they make coupling very difficult to control.

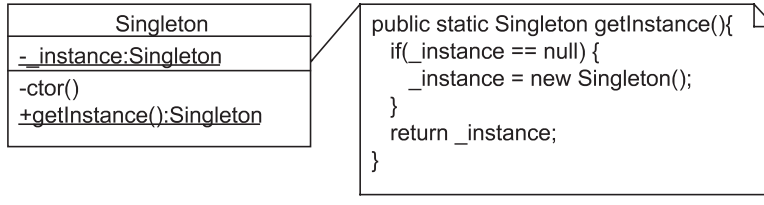


Figure B.48 Implementation forces

Options in Implementation

The Singleton is typically instantiated lazily—that is, the instance is not created until it is needed, and perhaps never (if no other entity ever calls `getInstance()`). However, this introduces a threadsafety problem if it is possible for one thread to be engaged in the creation of the instance while another is checking for null. In this worst-case scenario, it would be possible for two instances to be created. The first thread would have its own private instance; the second thread would have the instance that all other entities will now get. If this is a critical issue, there are other implementations that should be considered.

Deterministic instantiation

If the "lazy" aspect of the Singleton is not important in a given circumstance, the instance can be created as the class loads. This is threadsafe, as it uses the class loader, which must be threadsafe.

```
public class Singleton {
      private static Singleton _instance = new Singleton();
      private Singleton(){}
      public static Singleton getInstance() {
            return _instance;
      }
}
```

This is possibly the simplest solution, and is preferred where lazy instantiation is not necessary. An alternative implementation of this same solution is to use a static constructor, if the language/platform provides one. In either case, the load is atomic and deterministic.

Using a synchronization semaphore

Most languages have a way of locking access to a method while a thread is running it. If you require lazy instantiation, you can use a lock, synchronize, or similar mechanism to ensure threadsafety.

```
public class Singleton {
      private static Singleton _instance;
      private Singleton(){}
      //obtain lock
      public static Singleton getInstance() {
            if(_instance == null) {
                  _instance = new Singleton();
            }
```

```
            return _instance;
      }
      //release lock
}
```

The problem, potentially, is performance. Obtaining the lock is typically a fairly costly action, in terms of speed, and yet the lock is really only needed on the first call. Once the instance is created, the check for null never passes. We pay the locking price on every call, however.

If performance is not critical, this is a simple way to achieve threadsafety and lazy instantiation.

Double-Checked, Locking Singleton

A common suggestion for obtaining both high-performance and laziness in a threadsafe Singleton is to use the following implementation, typically called a Double-Checked, Locking Singleton.

```
public class Singleton {
      private static Singleton _instance;
      private Singleton(){}
      public static Singleton getInstance() {
            if(_instance == null) {  // once the instance is
                                     // created, we bypass
                                     // the lock

                  //obtain lock here
                  if(_instance == null) { // a thread which
                                          // waited for the
                                          // lock now checks
                                          // to see if the
                                          // instance is
                                          // still null.

                        _instance = new Singleton();
                  }
                  //release lock
            }
            return _instance;
      }
}
```

This double-check ensures that a thread, which was waiting for the lock, checks again after the thread obtains the lock, in the off chance (worst-case scenario) that the previous thread created the instance while it was waiting. When the instance is created, the first check never passes and so the lock is never taken, improving performance.

The potential problem has to do with compiler optimization and memory management, and the issues are complex and contentious. One example, however, is this: The compiler may notice that there are two identical null checks on instance, and that there is no code between them that could potentially change the state of the variable. Therefore, a compiler may "feel free" to optimize out one of the two calls, which destroys our threadsafety without our knowledge (unless we analyze the generated bytecode or machine language).

In some languages/platforms, this may or may not be a concern, but it is certainly controversial at this point:

- Java discussion: http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html
- .Net discussion: http://blogs.msdn.com/cbrumme/archive/2003/05/B/51445.aspx

Nested class

Another option for high-performance, lazy threadsafety is to use a nested class. Again, we are simply leaning on the threadsafety of the class loader, but the nested class (if it loads only when referenced) makes the `Singleton` lazy as well.

```
public class Singleton {
     private Singleton(){}
     public static Singleton getInstance() {
          return Nested.instance; // causes the nested
                                  // class to load, on
                                  // first reference
     }
     private class Nested { // does not load until a
                            // member is referenced
          public static Singleton instance;
          static       {
               instance = new Singleton();
          }
     }
}
```

There are specific implementations in different languages and on different platforms (in .Net, for instance, the nested class must be marked `BeforeFieldInit` to make sure it does not load when Singleton loads), but the concept is the same.

Refactoring Issue

The Singleton is easily refactored from a simple encapsulated constructor.

```
public class User {
      private user(){}
      public static User getInstance() {
            return new User();
      }
}
```

This is a recommended general practice, making it easy to refactor a simple class into a Singleton without changing the entities that consume its services.

Encapsulated Destruction

In systems that do not manage memory for you (a.k.a. unmanaged code), you may want to encapsulate the potential destruction of the instance as well.

```
public class Singleton {
      private static Singleton _instance;
      private Singleton(){}
      public static Singleton getInstance() {
            if(_instance == null) {
                  _instance = new Singleton();
            }
            return _instance;
      }
      public static void returnInstance(Singleton anInstance){
            //Take whatever action is needed
      }
}
```

All client entities are coded to call `getInstance()` to obtain the instance, and `returnInstance()` when they are done with it.

In a typical Singleton, no action is needed in `returnInstance()`, but this allows for instance counting and the cleanup of memory, if this is desired. Essentially, this introduces a simple form of memory management to an unmanaged environment.

As mentioned in the preceding "Refactoring Issue," this is also a best-practice for encapsulating the destruction of entities in general.

Consequent Forces

Testing Issues

The unit test for the `Singleton` primarily is concerned with its functional aspects, which are not covered here as they vary from case to case. However, the test can assert that two or more calls to the `getInstance()` method returns the same instance, using `AssertSame` or a similar call (depending on the testing framework used).

```
public void TestGetInstance() {
      Singleton 1stInstance = Singleton getInstance();
      Singleton 2ndInstance = Singleton getInstance();
      AssertSame(1stInstance, 2ndInstance);
}
```

Whether this is necessary to test varies from project to project. If this test is in place, it can be used to test-drive a change in `Singleton` for load-balancing (see the next section, "Cost-Benefit (Gain-Loss)").

Cost-Benefit (Gain-Loss)

- A limited-use resource can be protected from erroneous multiple use without complicating the consuming client entities.
- Singleton can easily scale to two, three, or more instances for load-balancing. Since the cardinality issue is encapsulated, this can be freely changed if this is deemed appropriate.
- If the Singleton gains statefullness, care must be taken to avoid creating a global.

The Strategy Pattern

Contextual Forces

Motivation

A single behavior with varying implementation exists and we want to decouple consumers of this behavior from any particular implementation. We may also want to decouple them from the fact that the implementation is varying at all.

Encapsulation

The various versions of the behavior, how many variations there are, which variation will be used under a given circumstance, and the design of each varying implementation are all hidden from the client. We may also encapsulate the fact that the behavior is varying at all, depending on implementation.

Procedural Analog

A simple switch or other branching logic: where the condition that drives the branch is a concern of the client or system, but not the algorithm itself (see the Chain of Responsibility pattern).

```
if (condition) {
      // Algorithm 1
} else {
      // Algorithm 2
}
```

Non-Software Analog

When I was a young boy, I used to have a Mattel Superstar airplane that had a little plastic disk inside (see Figure B.49). The grooves and notches on the disk made the airplane fly in a given pattern.

Figure B.49 Mattel Superstar airplane

By changing the disk the plane held, as shown in Figure B.50, I would get a different flight path. Each disk had a different shape, and as it was rotated on a cam, a feeler rod was moved back and forth, turning the rudder as the plane flew.

I did not need to change anything about the plane to get a different flight path; I just gave it a different-shaped disk.

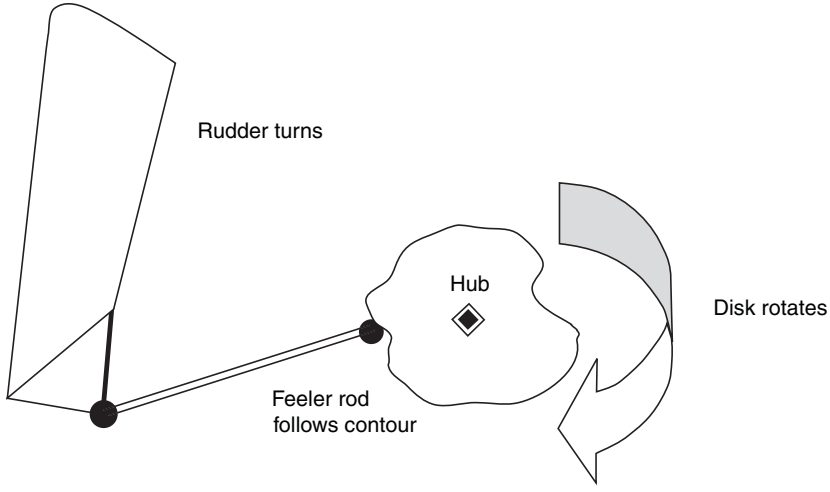


Figure B.50 Different flight paths when different disks are inserted before flight

All the disks fit in the plane because they were under the maximum size and had a particular little diamond-shaped hole in their hub that allowed them to mount on the rotating cam. This was the disk's interface, which the plane was designed to accept. So long as the disk conformed to this, I could make new disks that did not even exist when the plane was created, and they would work, creating new flight behaviors. Only one disk could be mounted at a time, of course.

In this analogy, the plane is the context object and the disks are the implementation of the Strategy Object. The size of the disks and the shape/size of the hole at their center is the interface of the Strategy.

Implementation Forces

Example

See Figure B.51.

Code

```
public class Context {
      public void request(Strategy s) {
            s.operation();
      }
}

public abstract class Strategy {
      public abstract operation();
}

public class Strategy_V1 extends Strategy{
      public void operation() {
            // Implementation V1
      }
}

public class Strategy_V2 extends Strategy{
      public void operation() {
            // Implementation V2
      }
}
```

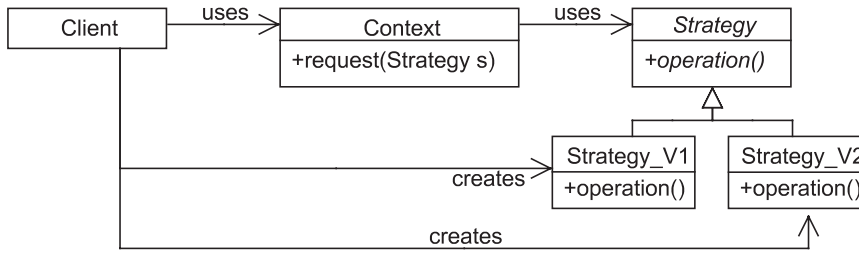


Figure B.51 Implementation forces

Questions, Concerns, Credibility Checks

- Can all the versions of the algorithm (`Strategy`) really share the same interface? What are the differences between them, and how can I resolve them in such a way that they all can be used in the same way?
- How does the `Context` object obtain an instance of a Strategy implementation to delegate to?
- How often and under what circumstances does the particular Strategy implementation the client is using change?
  - Each time the request is called?
  - Each time the `Context` is instantiated?
  - Each time the application is loaded?
  - Each time the application is installed?

Options in Implementation

- Does the `Client` hand in a `Strategy` implementation each time it interacts with the `Context`?
- Is an object factory used?
- If an object factory is used, does the `Client` or the `Context` interact with the factory?
- Is there a factory that builds the `Context` and hands the `Strategy` in via its constructor?

These are illustrated in Figures B.52, B.53, and B.54.

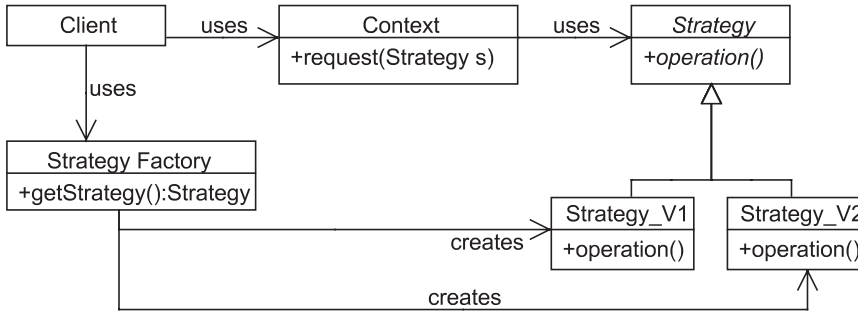


Figure B.52 Strategy with factory, Client uses the factory

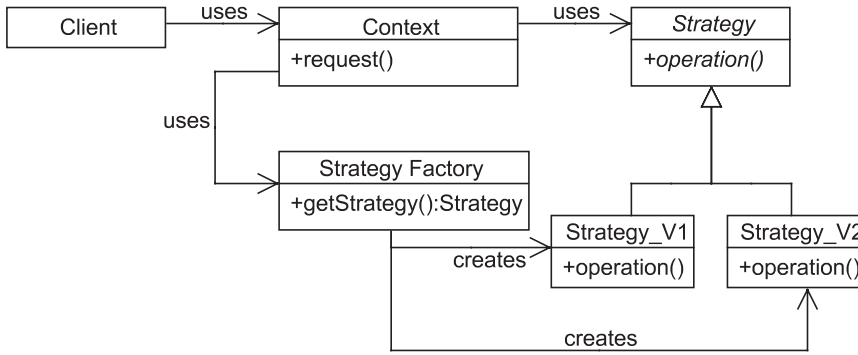


Figure B.53 Strategy with factory, Context uses the factory

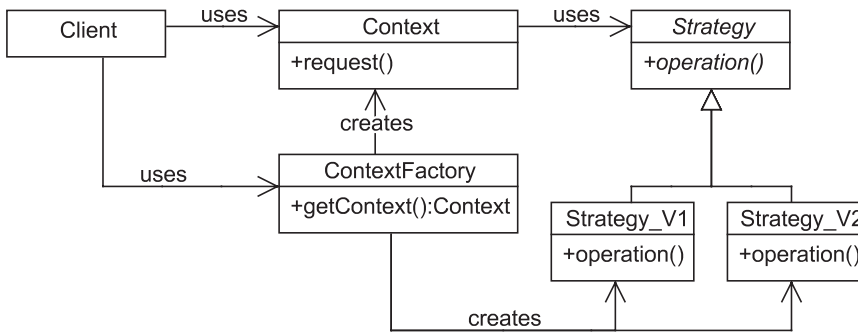


Figure B.54 Strategy with factory, built by a factory

Consequent Forces

Testing Issues

The Strategy implementations can each be tested on its own, but a mock or fake object should be used to test the `Context` object. As shown in Figure B.55, the mock takes the place of a Strategy implementation and allows the test to observe the interaction between the `Context` and the `Strategy` abstraction.

Cost-Benefit (Gain-Loss)

- The `Context` and the `Strategy` objects tend toward strong cohesion.
- The `Context` object is decoupled from the particular `Strategy` implementations, and therefore we are free to add new implementations or remove existing implementations without this affecting the `Context`. The issue is open-closed.
- The algorithm provided by the `Strategy` object must be publicly available, since it is in a separate object. If it were left as code within the `Context` object, it could be private (encapsulated), but after it is "pulled out" into the `Strategy`, this is not possible.

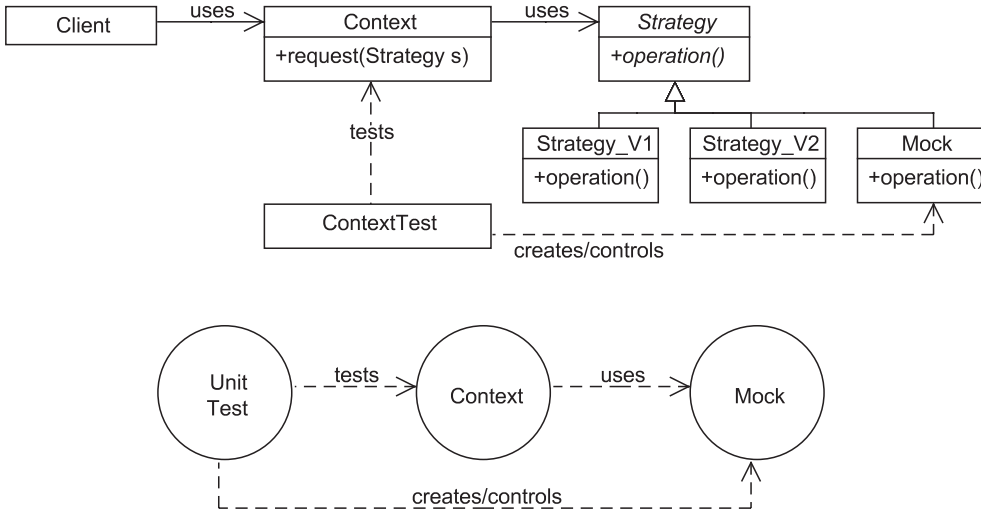


Figure B.55 The mock takes the place of a Strategy implementation

- The algorithm cannot directly access the state of the `Context` object. Any state needed by the algorithm must be "passed in" via parameters. Any effect that should propagate back to the `Context` must be returned. This decouples them and makes testing easier, but it can lead to a very broad interface on the `Strategy` abstraction. This can reduce its reusability.
- More classes are needed when compared to a simple procedural solution within `Context` (branching logic).
- The delegation from the context to the `Strategy` object introduces a small degradation in performance.

The Template Method

Contextual Forces

Motivation

Abstract out the skeletal steps of an algorithm, allowing subclasses to override the specific implementations of each step.

Encapsulation

- The steps of the algorithm
- The implementation of each step
- The variation of the implementations (how many and the fact that they are varying)

Procedural Analog

When a series of steps varies and they all transition from one variation to another at the same time, as a set, you can accomplish this procedurally

with a series of logical branches or switch/case statements. These are all based on the state of a common value.

```
int var = //state set here

//Step one
switch var {
      case 1:
      // first step, variation 1

      case 2:
      // first step, variation 2

      case 3:
      // first step, variation 3
}

//Step two
switch var {
      case 1:
      // second step, variation 1

      case 2:
      // second step, variation 2

      case 3:
      // second step, variation 3
}

//Step three
switch var {
      case 1:
      // third step, variation 1

      case 2:
      // third step, variation 2

      case 3:
      // third step, variation 3
}
```

Non-Software Analog

Recipes are sometimes expressed as a series of steps for preparation, combination, processing (cooking), and finishing for consumption. Sometimes, however, the same recipe can be used to feed a varying number of people. When this is the case, the amounts are sometimes omitted to be filled in based on the number of people to be fed. In these cases, the recipe is the same, but the amounts all vary together, based on the state "number to be fed." The recipe (the non-changing part) is the template method, and the amounts are the delegated methods that are overridden.

Implementation Forces

Example

See Figure B.56.

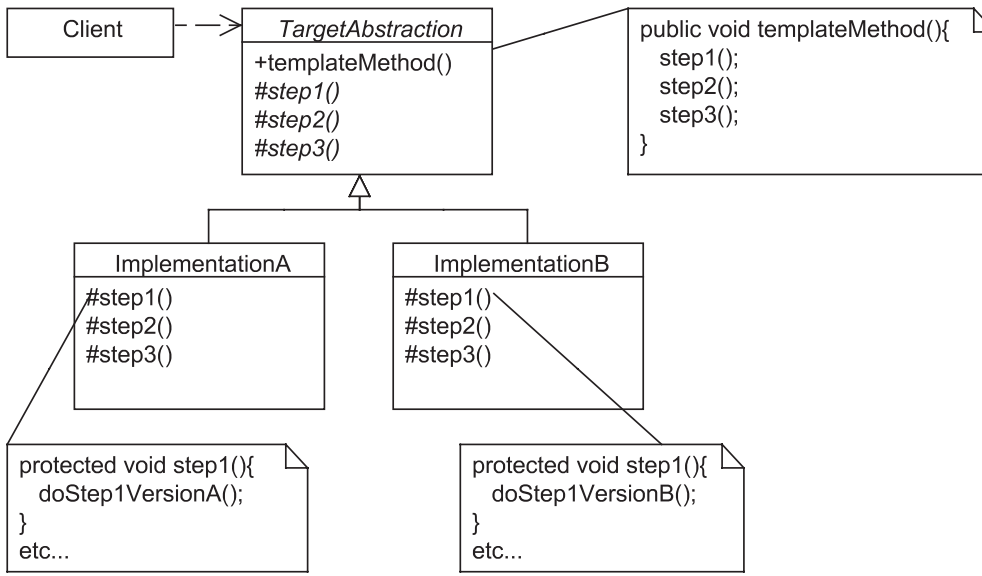


Figure B.56 Implementation forces

Code

```
public abstract class TargetAbstraction {
      public void templateMethod() {
            step1();
            step2();
            step3();
      }

      protected abstract void step1();
      protected abstract void step2();
      protected abstract void step3();
}

public class implementationA extends TargetAbstraction {
      protected void step1() {
            // Implementation of Step1, version A here
      }
      protected void step2() {
            // Implementation of Step2, version A here
      }
      protected void step3() {
            // Implementation of Step3, version A here
      }
}

public class implementationB extends TargetAbstraction {
      protected void step1() {
            // Implementation of Step1, version B here
      }
      protected void step2() {
            // Implementation of Step2, version B here
      }
      protected void step3() {
            // Implementation of Step3, version B here
      }
}
```

Or, to show our non-software analog (see Figure B.57) in UML, see Figure B.57.

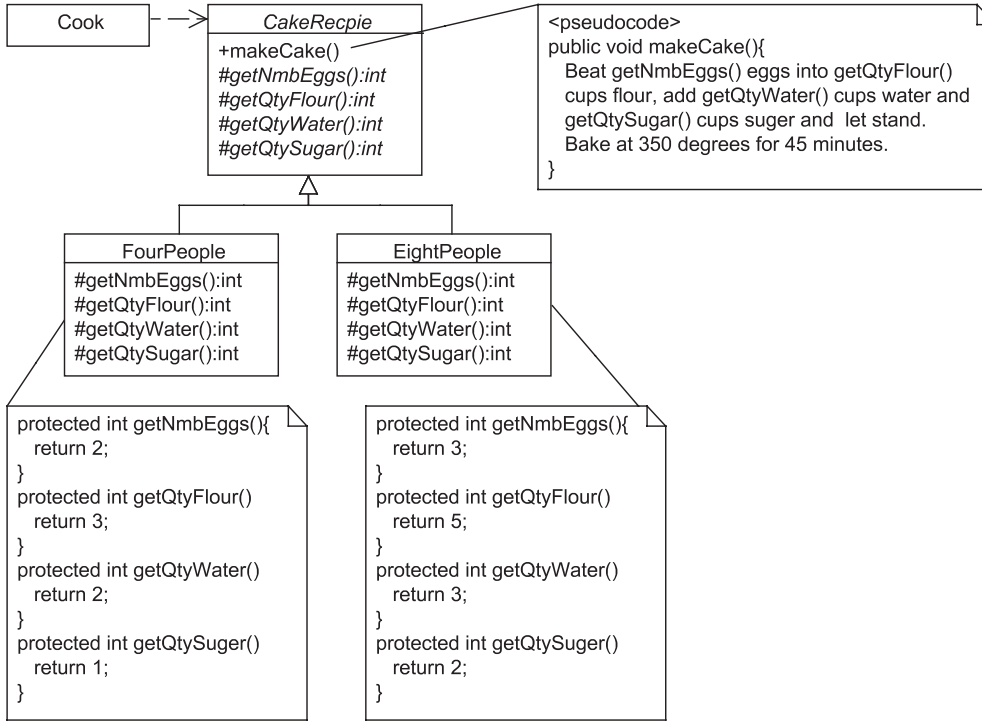


Figure B.57 Non-software analog in UML

Questions, Concerns, Credibility Checks

Can all of the variations of this behavior be resolved to the same set of steps without making them overly generic or without forcing any one of them to give up critical behavior?

Do some subclasses require different/more parameters or returns from the delegated methods? See the discussion of "ignore parameters" in the following "Options in Implementation" section.

The delegated methods, which are overridden in the subclasses, should not be made public to avoid any possibility that other entities will become coupled to them. If the implementation language does not allow abstract methods to be protected (some languages force all abstract methods to be public), you should implement the methods in the base class with default or stubbed-out behavior.

Options in Implementation

Sometimes, the variations of the specific behavior can require different parameters. When this happens, creating a generic interface for these methods can require the use of ignore parameters. For example:

```
public class StarshipWeapon {
     // This is the Template Method
     public int fire(int power){
          int damage = 0;
          if(hasAmmo(power)){
               if(doesHit(range, direction)){
                    damage=calculateDamage(power);
               }
          }
          return damage;
     }

     protected abstract bool hasAmmo(int power);
     protected abstract bool doesHit(long range,
                                     double direction);
     protected abstract int calculateDamage();
}

public class PhotonTorpedos extends StarshipWeapon {
     protected bool hasAmmo(int ignorePower){
          // return true if there are torpedos remaining,
          // else return false
     }
     protected bool doesHit(long range, double direction){
          // Calculation based on range and bearing to the
          // target return true if hit, false if miss
     }
     protected int calculateDamage(){
          // Torpedoes always do full damage if they hit
          return 500;
     }
}

public class Phasers extends StarshipWeapon {
     protected bool hasAmmo(int Power){
          // Store power for later use in calculating damage
          // return true if there is enough energy to fire
          // phasers at this power
     }
```

```
      protected bool doesHit(long range, double ignoreDirection){
            // Store range for later use in calculating damage
            // Phasers always hit
            // Direction is irrelevant for Phasers, so name
            // the parameter accordingly
            return true;

      }
      protected int calculateDamage(){
            // Take power, attenuate for range, and return the
            // reduced amount as the damage to be applied
      }
}
```

The use of the term `ignoreXXX` as the parameter names in subclasses that do not use the parameter makes it clearer during maintenance that this is being done. This allows all subclasses to operate using the same template method.

Consequent Forces

Testing Issues

As shown in Figure B.58, the base class algorithm can be tested by deriving a Fake object just for testing.

Cost-Benefit (Gain-Loss)

The use of a Template Method pushes redundant aspects (skeletal algorithm or template method) to the base class, leaving nonredundant aspects (variation) in derived classes. This separates what changes from what does not and it improves cohesion. In addition, it does the following:

- It can create overly generic designs.
- Implementing new subclasses is simpler because they do not have to implement the skeletal algorithm.
- It improves readability.

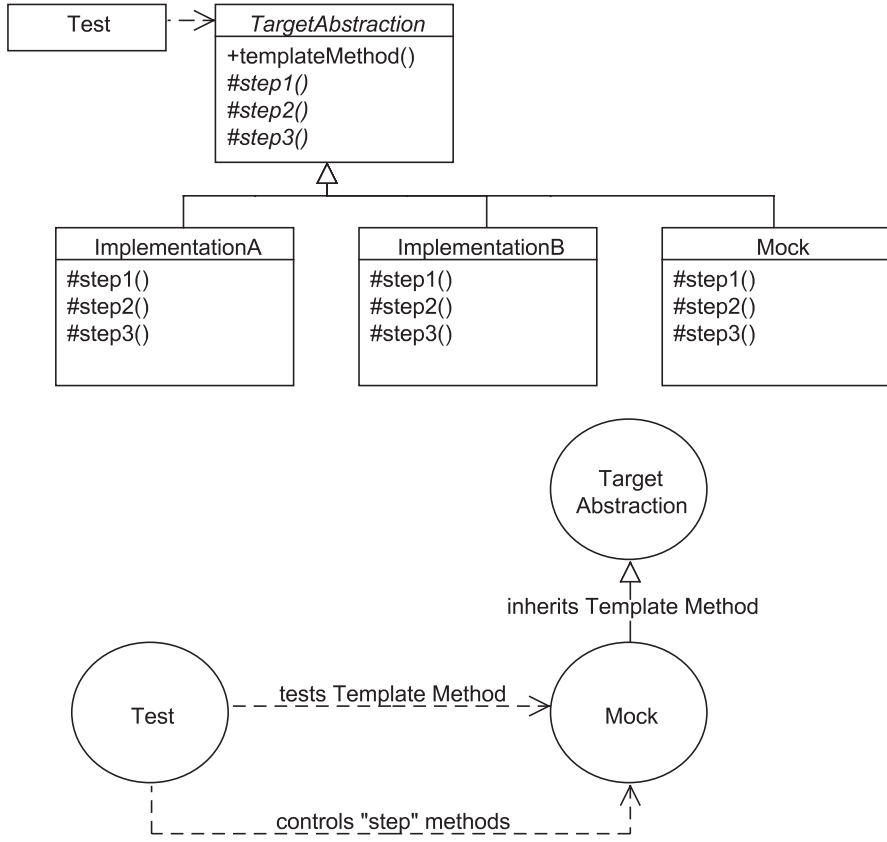


Figure B.58 Testing of a base class algorithm

Design Patterns Exercises

Exercise – Multiple choice

Choose the design pattern that is most suitable for the following cases.

1) We want to construct a complex object part by part
   a) Builder
   b) Factory
   c) Abstract Factory
   d) Prototype

2) There are rules about how to manage a set of objects. These can relate to the number, the lifespan and more.
   a) Singleton
   b) Object Pool
   c) Decorator
   d) Chain of Responsibility

3) When a new object is needed, clone an existing object.
   a) Mediator
   b) Interpreter
   c) Memento
   d) Prototype

4) Encapsulates that another class having the wrong interface.
   a) Proxy
   b) Facade
   c) Adapter
   d) Template

5) Don’t settle for the complex interface – simplify it.
   a) Proxy
   b) Facade
   c) Adapter
   d) Template

6) Multiple wrappers for existing functionality.
   a) Chain of Responsibility
   b) Decorator
   c) Bridge
   d) Visitor

7) Choose from multiple algorithms at runtime.

a) Strategy
b) Template
c) Chain of Responsibility
d) Mediator

8) We have a consistent set of steps to follow but individual steps may have different implementations.
a) Strategy
b) Template
c) Chain of Responsibility
d) Mediator

9) Encapsulates both which services are available and the preference of which service to do the task
a) Strategy
b) Template
c) Chain of Responsibility
d) Mediator

10) Define a central class that acts as a message routing service to all other classes.
a) Strategy
b) Template
c) Chain of Responsibility
d) Mediator

11) Encapsulate the nature of a collection
a) Memento
b) Iterator
c) Observer
d) Visitor

12) Serialization in Java
a) Memento
b) Iterator
c) Observer
d) Visitor

13) Generally used between GUI class and Business logic class.
a) Memento
b) Iterator
c) Observer
d) Visitor

14) It should not be used to store global variables, but it should only be used to limit the number of objects for a class.
a) Bridge
b) Abstract Factory
c) Factory

d) Singleton

15) Java I/O uses this pattern heavily
    a) Prototype
    b) Abstract Factory
    c) Factory
    d) Decorator

16) To localize the creation of a single family of objects, we use
    a) Abstract Factory
    b) Factory
    c) Builder
    d) Prototype

17) AWT in Java for the creation of GUI uses
    a) Abstract Factory
    b) Factory
    c) Builder
    d) Prototype

18) While using JUnit testing, we normally replace heavy-duty time-consuming objects like database, network, etc with Mocks.
    a) Visitor
    b) Command
    c) Proxy
    d) Bridge

19) For undo/redo in an editor
    a) Proxy
    b) Visitor
    c) Bridge
    d) Command

20) To store Files and Directories, where a directory can contain other directories and files.
    a) Singleton
    b) Composite
    c) Chain of Responsibility
    d) Prototype

Exercise – Procedural to OO

Let's say we have an entity in the system that is responsible for monitoring the health of a connection to another machine. On a regular basis, this monitor is polled by another entity in the system, and it then checks

the connection to see if it is still up and running properly. Whether it is or not, it then sends a message elsewhere that indicates this status.

However, let's say where this status should be sent is a varying issue. Sometimes (during business hours, let's say), we want to send this information to a local display object; sometimes (after hours), we want to send this information to a pager, but in that case only if there is a failure. We do not want to wake the systems operator in the middle of the night to say that everything's fine!

The procedural approach is to create a slightly complex conditional as shown below:

```
get the status of the connection
if (it is between 8AM and 5PM)
    send the status to the local display
if (the connection failed or time between 5PM and 8AM)
        send the status to the pager
do whatever cleanup is needed and return
```

Which patterns will help you to improve this code. Assume that the condition checking needs a few lines of code and sending the status also is a significant task.

Exercise – Storage Explorer

The idea of this application came when I desperately needed an application similar to Windows Explorer that can show me the file size composition inside my storage. I want to know, for example, under the Program Files folder, which folder uses what size of space, including the numbers in percentage. I want also to know how big my mp3 total size in drive D compare to my jpg files. The information I want to see should look like this:

```
C:\Program Files:
      Microsoft          445,123 KB        43%
      Adobe              234,744 KB        25%
      Symantec            98,906 KB        10%
      ...
```

And this

```
D:\
      *.mp3              602,456 KB        47%
      *.jpg              305,830 KB        30%
      *.doc              245,355 KB        20%
      ....
```

I need the information is presented in a chart to help me visualize the numbers as well.

The application features are:

- A TreeView containing nodes that represents file system folders. The folders can be selected to see the information about file size structure within that path.
- It shows information about files size grouped by folders
- It shows information about files size grouped by file type
- The information is presented in listview, pie chart and bar chart on the right panel.

Draw the software class diagram and mention the Design patterns used.

Exercise: HR Application

This system consists of a database of the company's employees, and their associated data, such as time cards. The system must pay all employees the correct amount, on time, by the method that they specify. Also, various deductions must be taken from their pay.

- Some employees work by the hour. They are paid an hourly rate that is one of the fields in their employee record. They submit daily time cards that record the date and the number of hours worked. If they work more than 8 hours per day, they are paid 1.5 times their normal rate for those extra hours. They are paid every Friday.
- Some employees are paid a flat salary. They are paid on the last working day of the month. Their monthly salary is one of the fields in their employee record.
- Some of the salaried employees are also paid a commission based on their sales. They submit sales receipts that record the date and the amount of the sale. Their commission rate is a field in their employee record. They are paid every other Friday.
- Employees can select their method of payment. They may have their paychecks mailed to the postal address of their choice, have their paychecks held by the paymaster for pickup, or request that their paychecks be directly deposited into the bank account of their choice.
- Some employees belong to the union. Their employee record has a field for the weekly dues rate. Their dues must be deducted from their pay. Also, the union may access service charges against individual union members from time to time. These service charges are submitted by the union on a weekly basis and must be deducted from the appropriate employee's next pay amount.
- The payroll application will run once each working day and pay the appropriate employees on that day. The system will be told what date the employees are to be paid to, so it will generate payments for records from the last time the employee was paid up to the specified date.

Draw the software class diagram needed for the above application in the business layer. Mention the design patterns used.

Exercise: Game of Life

The Game of Life is not a game in the conventional sense. There are no players, and no winning or losing. Once the "pieces" are placed in the starting position, the rules determine everything that happens later. Nevertheless, Life is full of surprises! In most cases, it is impossible to look at a starting position (or *pattern*) and see what will happen in the future. The only way to find out is to follow the rules of the game.

Rules of the game: Life is played on a grid of square cells--like a chess board but extending infinitely in every direction. A cell can be ***live*** or ***dead***. A live cell is shown by putting a marker on its square. A dead cell is shown by leaving the square empty. Each cell in the grid has a neighborhood consisting of the eight cells in every direction including diagonals.

To apply one step of the rules, we count the number of live neighbors for each cell. What happens next depends on this number.

- A dead cell with exactly three live neighbors becomes a live cell (birth).
- A live cell with two or three live neighbors stays alive (survival).
- In all other cases, a cell dies or remains dead (overcrowding or loneliness).

Note: The number of live neighbors is always based on the cells ***before*** the rule was applied. In other words, we must first find all of the cells that change before changing any of them. Sounds like a job for a computer!

Play this game on the computer. What classes will be needed, to write this game? Which design patterns will be useful?

Exercise: Project – Design, Code and Unit-Test Zip File Extractor

Here we will:

- Design the first version of a ZIP file manager and extractor
- Produce a set of UML class and sequence diagrams to document your design
- Write a covering set of unit tests
- Experience working with Eclipse and JUnit

Requirements

Create an executable named `xzip` which is able to print and extract the contents of ZIP files. The program's usage from the command line should be:

```
xzip [-l | -s | -x] [-nr] <zip file name>
```

Note that the first two parameters are optional, but the file name is not. Parameters must appear in this order.

A zip file can contain files, which may be compressed (but don't have to be). In addition, a zip file can also contain directories, which may by themselves contain files and other directories. When a zip file is extracted, the directories it contains are extracted as well, and files within these directories are extracted to their correct relative location.

For the examples given below, assume that the zip file `my.zip` contains the following: A file called `mysong.mp3` which is an MP3 song, a file called `myicon.gif` which is a GIF image, and a directory called '`web`' which includes two HTML files called `index.html` and `other.html` and one directory called `backup` file which includes two other files with these same names.

The first argument of `xzip` dictates what action will be performed on the zip file:

- `-l` (long print): This action prints the names and details of the files stored in the given zip file. The long format means that in addition to the file names, more details are printed for each file. For example:

```
> java xzip -l my.zip
mysong.mp3      (7901 bytes, MP3 file)
myicon.gif      (910 bytes, 32x32 GIF image)
web             (directory, total 5 files)
   index.html       (5657 bytes, 122 lines of text)
   other.html       (6983 bytes, 208 lines of text)
   backup           (directory, total 2 files)
      index.html       (5622 bytes, 120 lines of text)
      other.html       (7501 bytes, 234 lines of text)
```

  As you can see, the details printed for each file type are different, and follow these rules:
  1. For each text file, the name of the file is printed, followed by a tab, and then the text `(N bytes, M lines of text)` where `N` is the uncompressed size of the file, and `M` is the number of lines (measured by the number of new-line characters) in the file. A text file is defined as any file whose extension is `*.txt`, `*.html` or `*.java`.
  2. For each image file, the name of the file is printed, followed by a tab, and then the text `(N bytes, WxH EXT image)` where `N` is the uncompressed size of the file, `W` is the image's width, `H` is the image's height, and `EXT` is the image file extension in uppercase. An image file is defined as any file whose extension is `*.gif` or `*.jpg`.
  3. For any other file type - any regular file not a text or zip file, such as `mysong.mp3` and `backup.zip` in the above example - print the file name, a tab, and the text `(N bytes, EXT file)` where `N` is the size of the uncompressed file and `EXT` is the file extension in uppercase.
  4. For each directory, the name of the directory is printed, followed by a tab, and then the text `(directory, total N files)` where `N` is the number of files in that directory. Each directory that the directory contains is counted as one (for the directory itself) plus the number of files inside that directory, recursively. In addition, as the example above shows, the contents of the directory are printed in the following lines, under the same rules, with an indent of three spaces relative to the indent of the directory.

- `-s` (short print): This action prints the names of the files and directories in the given zip files, but without the extra details (file size, number of text lines and so forth) which the long format provides.

Printing is equivalent to what the `-l` option outputs, including indentation and recursion into directories and zip files - only the text in parentheses for each file/directory is not printed. For example:

```
> java xzip -s my.zip
mysong.mp3
myicon.gif
web
   index.html
   other.html
   backup
      index.html
      other.html
```

- `-x` (extract): This action actually extracts the contents of the zip file into the file system. That is, for each file/directory in the zip file, a new uncompressed file/directory should be created in the file system. Files should be created in the current directory, but files that are inside directories in the zip file should be created inside these relative directories in the file system. If a file is being extracted and another file with the same name already exists, then the existing file should be overwritten.
  Upon completion, this action prints one line to the console in this format: `Extracted N files, M files failed`. `N` is the total number of files and directories that were successfully created, and `M` is the total number of files and directories that failed. In addition, one line is printed for each failed file or directory, including a detailed error message. Whenever possible, an error should not result in halting the entire program, and the program should output the error message and continue normally. For example:

  ```
  > java xzip -x my.zip
  Error: Failed to extract myicon.gif: Cannot overwrite existing file - file is in
  use
  Extracted 7 files, 1 files failed
  ```

  And the reported 7 successful files will be the following (locations are relative to the current directory):

  ```
  mysong.mp3
  web\
  web\index.html
  web\other.html
  web\backup\
  web\backup\index.html
  web\backup\other.html
  ```

The second command-line argument of `xzip` means "no recursion", and if it appears then all actions should be performed without recursion into directories. This means that only one summary line is printed for every directory in the printing actions, and that the directory is created but not populated in the extract action. For example:

```
> java xzip -l -nr my.zip
mysong.mp3      (7901 bytes, MP3 file)
myicon.gif      (910 bytes, 32x32 GIF image)
web             (directory, total 5 files)
```

```
> java xzip -x -rn my.zip
Extracted 3 files, 0 files failed
```

In this example, the 3 extracted files will be `mysong.mp3, myicon.gif` and `web\.`

The default action is `-l`, meaning that `xzip my.zip` is equivalent to `xzip -l my.zip`. You should print an informative usage message if the program is activated with no or illegal command-line arguments. You should print a detailed error messages and exit gracefully when a critical error occurs (the given zip file does not exist, the given zip file is corrupted and so on).

Design

While this exercise can be programmed within a single class, this won't work since this `xzip` is only a first version, so it is crucial to maintain an open mind with respect to possible future requirements. Consider the following:

1. It may be required to be able to read input format other than ZIP, such as TAR, ARJ, CAB and other archive file formats. The input does not even have to be a file: It can be the set of files of a given directory, a given FTP server address and so forth.
2. It may be required to support other file types that the long print action provides additional details about. For example, printing the image size for image files such as `myicon.gif`, or printing the song duration for music files such as `mysong.mp3`.
3. It may be required to produce the output in formats other than plain text - HTML, PDF, Word or others. It may also be required to write output in several formats at once, for example: `xzip -pdf myzipfiles.pdf -html myzipcontents.html my.zip`.
4. It may be required to modify the input zip file instead of just printing or extracting it. For example, new actions may enable adding files and directories to the zip file, changing the date and time signature of zipped files, and so on.
5. It may be required to support recursion into zip files, and not only into directories. For example, if a zip file contains another zip file, then its contents would have to be printed (recursively) like a directory's contents are printed, and when a file is extracted any zip files it contains would be extracted as well.
6. It may be required to activate all of the program's features not only from the command-line, but also from a graphical user interface, or perhaps even two user interfaces (for example, one that is a custom UI for handling zip files, and another which is fully integrated with the Windows Explorer).

You must design your program so that it is easy to add code that implements the above requirements. Assume that you are the one who will actually have to code it - that's how it usually is in "real life". For each of the above requirements write an explanation in your README file, not more than three sentences long, which explains how it should be coded.

{ If you finish early add this requirement:
It may be required to define filters on which files get printed or extracted, in addition to the -nr switch. For example, new command-line arguments can dictate that only text files should be acted on, only files that match a given pattern (such as *.cpp), and so on.

Solution: Write an Iterator for each kind of filter, whose next() method will move to the next element for which the filter is true. Such iterators are implemented as Decorators of other iterators, which easily enables to dynamically combine different filters and does not require changing or recompiling existing code.}

It is important that each solution you present will be at most three sentences long. The intention is to enforce the use of design patterns vocabulary rather than elaborating specific class and object relationships.

Code & Unit Test

This exercise intends you to divide your time equally between actual coding and between design, writing UML diagrams, and answering the above six design questions. With a proper design, this exercise is quick and simple to code. You are required to write in Java 5.0, and you may use the standard libraries to their full extent - the standard streams, strings and data structures. In particular, working with zip files is done using the `java.util.zip` package, and working with image files is done using the `javax.swing.ImageIcon` class. (While working with C++, the library at www.zlib.net may be used for compression related functions. The library at cimg.sourceforge.net may be used for image related functions.)

You are also required to use the Eclipse environment for this exercise, and are encouraged to take advantage of its editing and debugging features to their full extent.

You are required to provide class diagrams that include all your classes; there may be more than one diagram, if this is visually easier. You are also required to provide at least two sequence diagrams, depicting two interactions in your design which you consider important.

It is also required that you submit unit tests to test your work, and use the JUnit framework to do so. Organize the code such that the program source code is in one package (for example `xzip`), and the tests are in a separate package (for example `xziptest`). Each test should be self-validating - that is, know by itself whether it has passed or failed. Writing unit tests should be an integral part of coding: This is essential when code must be changed in newer versions. We recommend that you try test-first programming and in any case you will lose time if you only write all unit tests after you "finish" coding.

One metric for measuring the usefulness of a set of unit tests is called coverage, which means the percentage of your code that the unit tests actually run. Coverage of 90% or above is considered good, and you should aim to that goal.

The code you submit must be built with no compiler warnings, and pass all unit tests.

This is an advanced course, so there is emphasis is on proper design. However, as usual, you are as always expected to write clear code with a consistent style.

Exercise – Products

Draw Class diagram and mention the design patterns used for the problem below:

Your company sells both Simple Products and Product Bundles. A Simple Product is a stand-alone item, whereas a Product Bundle is a packaged collection of Simple Products. All products have a name. When you request the name of a Product Bundle, you should receive its name followed by the name of each of its constituents.
You must be able to determine the price of any product. Simple Products have a fixed unit price. The price of a Product Bundle is calculated by taking the sum of the prices of the bundle's parts. Some discount rate is applied to this total price. (for example, 10 percent for some, 15 percent for others)

Exercise – Two Problems

Describe one solution strategy that works for both problems.

A.You have a *remote* database server, and you want a *local* object to encapsulate all requests to that remote server, so that the local application programmer can treat the remote server as if it were local. How can we do this easily?

B.You have a document viewer program which may embed very large graphical images which are slow to load; but you need the viewer itself to come up very quickly when the document is selected. Therefore, images in the document should get loaded only as needed (when they come into view), not at document load time. How can we do this without complicating the design of the viewer?

Exercise – File Link

In a File Management System, there is a feature in called a Link that we wish to add to our design (like the Windows shortcut or the Macintosh alias). A Link is simply a navigational shortcut which allows a user to see a virtual copy of a file or a directory, as if it were local to the user's current location in the directory hierarchy. For most operations, the Link behaves exactly like the thing it is linked to, except that it may be deleted without deleting the actual directory or file.
Draw a class diagram for the required design

Exercise – Comany Software

Company X has a large amount of applications software written using a particular class library. Company Y that wrote the class library has now gone out of business. Company X buys a new class library from Company Z that provides the same functionality as the previous library, but unfortunately many of the classes have different interfaces. Company X does not have access to the source code for the old or the new library.
Suggest the appropriate design for the above problem.

Exercise – UI

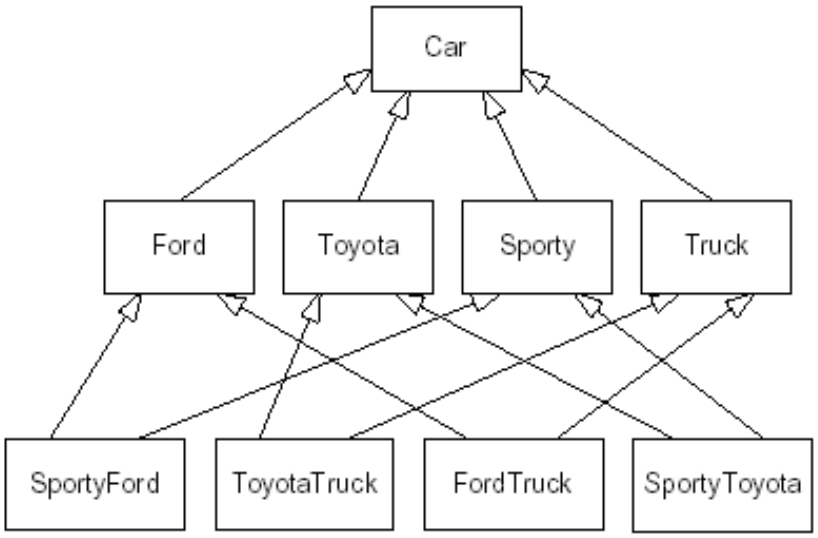
Consider a program which has many buttons, two list boxes and a text entry field.

1. When you select one of the names in the left-hand list box, it is copied into the text field for editing, and the *Copy* button is enabled.
2. When you click on *Copy*, that text is added to the right hand list box, and the *Clear* button is enabled.
3. If you click on the *Clear* button, the right hand list box and the text field are cleared, the list box is deselected and the two buttons are again disabled.

User interfaces such as this one are commonly used to select lists of people or products from longer lists. Further, they are usually even more complicated than this one, involving insert, delete and undo operations as well.

Exercise – Multiple Inheritance

Simplify the following structure. You may introduce new classes.



Exercise – Workflow

Consider a workflow system supporting software developers. The system enables managers to model the process the developers should follow in terms of processes and the work products. The manager can assign specific processes to each developer and set deadlines for the delivery of each work product. The system supports several types of work products., including formatted text, picture and URLs. The manager while editing the workflow, can dynamically set the type of each work product at run time. Assuming one of design goals is to design the system so that more work product types can be added in the future, draw class diagram to represent design.

Exercise – Credit Card

You have a file that contains credit card records. Each record contains a field for the card number, the expiration date, and the name of the card holder. In your system you have the following class structure for the credit cards:

a class CreditCard,
classes VisaCC, MasterCC, AmExCC that are all subclasses of
CreditCard,

You assume more subclasses for other credit card types will be added later on.
You now have to design the structure that reads a record from the file, verifies that the credit card number is a possible account number, and creates an instance of the appropriate credit card class. What design patterns could you use for that?
Important details: Credit card numbers cannot exceed 19 digits, including a single check digit

in the rightmost position. You can also determine the card issuer based on the credit card number:

- For MasterCard, First digit is a 5, second digit is in range 1 through 5 inclusive. Only valid lenght of number is 16 digits.
- For Visa, First digit is a 4. Length is either 13 or 16 digits.
- For AmericanExpress, First digit is a 3 and second digit a 4 or 7. Length is 15 digits.
- For Discover, First four digits are 6011. Length is 16 digits.

Exercise – Book Searches

A CSV (comma separated value) file has the list of books. The first line is the header "Title, Author, Publisher, Price". The remaining lines are book; one book per line.
Now we need to do various searches on this file e.g.

1. Find the costliest 5 books.
2. Find the cheapest 5 books.
3. Find books whose title contain the substring given as input.
4. Find books whose author contain the substring given as input
5. Find books whose publisher contains the substring given as input

Design classes for above functionality. Write unit tests to prove the correctness of your code.

Exercise – Railway Ticket Booking

The cost of railway ticket depends upon the class of travel and the distance.

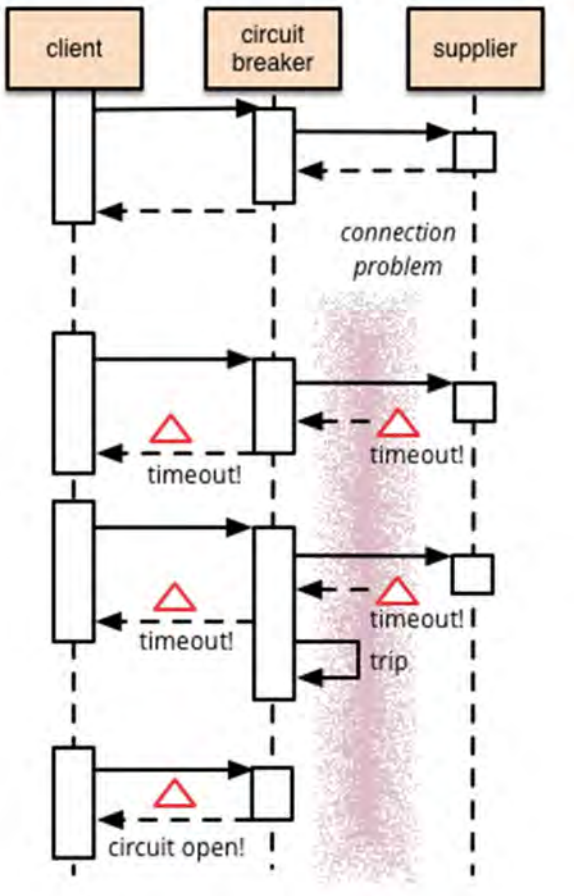
	Sleeper	AC CC	3AC	2AC	1st Class
Minimum charge	50	75	100	200	300
Kms free	20	0	0	0	0
Charge per Km in ₹	0.80	2.00	4.25	6.75	7.80

e.g. The distance from Mumbai to Pune is 180Kms. So a 2nd class Sleeper ticket will cost ₹128 i.e. (180 – 20) * 0.80. Minimum charge is not applicable, because 128 > 50.
If the travel date starts on the day of booking or next day, the ticket is treated as Tatkal. For tatkal tickets, the cost of the ticket is 3 times the normal ticket.
Senior citizen get 50% discount. No discount is given on Tatkal tickets.
The ticket prices are always rounded to nearest rupee.
Design classes for above functionality. Write unit tests to prove the correctness of your code.

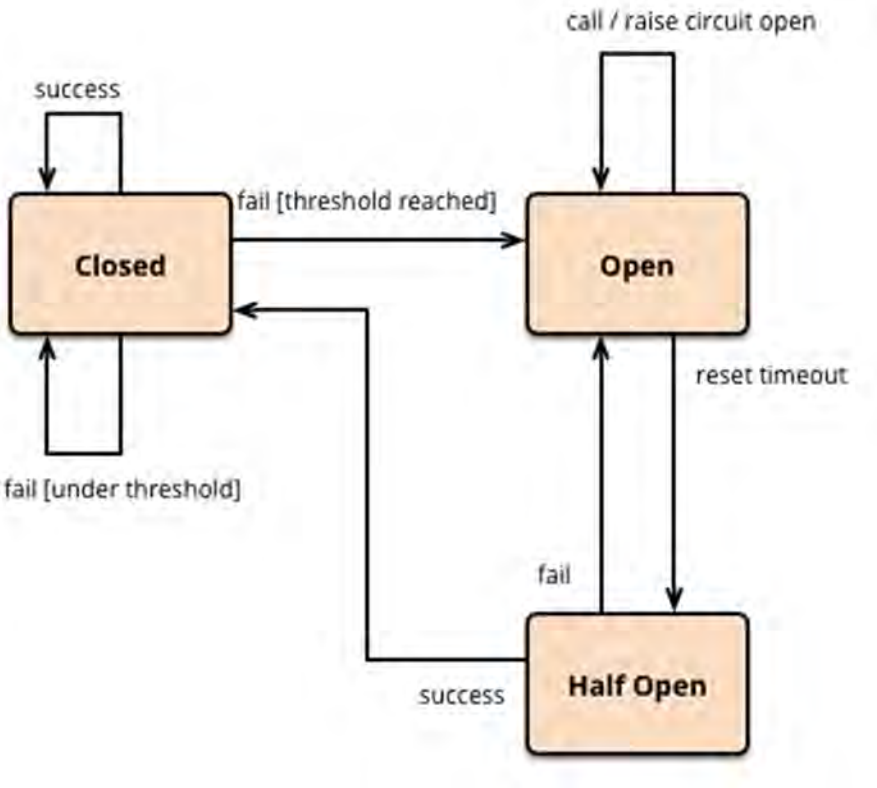
Exercise – CircuitBreaker

It's common for software systems to make remote calls to software running in different processes, probably on different machines across a network. One of the big differences between in-memory calls and remote calls is that remote calls can fail, or hang without a response until some timeout limit is reached. What's worse if you have many callers on an unresponsive supplier, then you can run out of critical resources leading to cascading failures across multiple systems. The Circuit Breaker pattern is used to prevent this kind of catastrophic cascade.

The basic idea behind the circuit breaker is very simple. You wrap a protected function call in a circuit breaker object, which monitors for failures. Once the failures reach a certain threshold, the circuit breaker trips, and all further calls to the circuit breaker return with an error, without the protected call being made at all. Usually you'll also want some kind of monitor alert if the circuit breaker trips.



This simple circuit breaker avoids making the protected call when the circuit is open, but would need an external intervention to reset it when things are well again. This is a reasonable approach with electrical circuit breakers in buildings, but for software circuit breakers we can have the breaker itself detect if the underlying calls are working again. We can implement this self-resetting behavior by trying the protected call again after a suitable interval, and resetting the breaker should it succeed.



There is now a third state present - half open - meaning the circuit is ready to make a real call as trial to see if the problem is fixed. Asked to call in the half-open state results in a trial call, which will either reset the breaker if successful or restart the timeout if not.

You have been given the tests for a working Circuit Breaker with above functionality. The skeleton code of the Circuit Breaker has been given you. You goal is to complete the code such that all tests pass. Please do not change the tests.

The End.