Design Patterns: From Analysis to Implementation

by

Net Objectives

This is an excerpt from the manuals for
Design Patterns Explained: A New Perspective for Object-Oriented Design

Not all of the Gang of Four design patterns are included because not all of them are covered in the course. Furthermore, we use a variation on the classification of the GoF patterns:

> The Decorator and Proxy patterns are classified as Structural patterns by the GoF. Since they both add instead of just combining functionality, I believe they are more behavioral in nature. In our minds, the Bridge can be considered as either a Structural or Behavioral pattern. As a structural pattern, it allows the use of two different, given, implementation in the same way. As a behaviorally pattern, it helps identify that there are variations of an implementation. We have also reclassified several Behavioral patterns as Decoupling patterns (a new classification of ours). That is because these patterns are more about decoupling than about managing new behavior.

This is a work in progress. Updates will be announced through our e-zine. You can subscribe to this by sending a message to info@netobjectives.com and putting subscribe in the subject.

Contents:

Abstract Factory*	Composite*	Visitor
Builder	Façade*	Chain of Responsibility*
Factory Method*	Proxy – Virtual*	Iterator*
Object Pool*	Decorator*	Mediator*
Prototype	Proxy – adding function*	Memento
Singleton*	State*	Observer*
Adapter*	Strategy*	Proxy – accessibility*
Bridge*	Template Method*	Model-View-Controller

* Along with the Analysis Matrix, these are considered minimal competencies for software developers by Net Objectives

Design Pattern Matrix

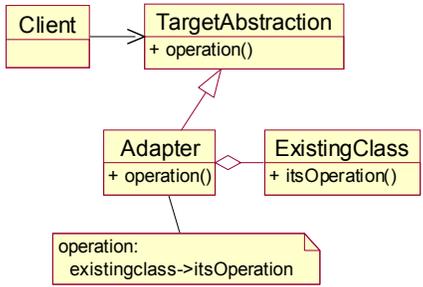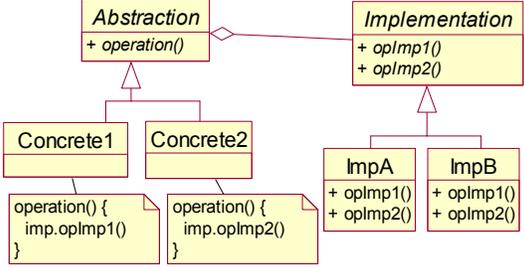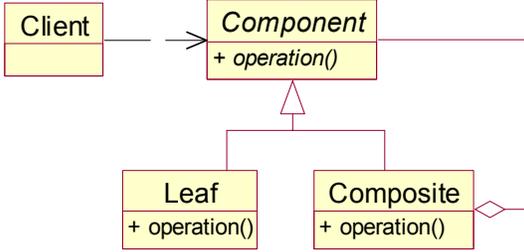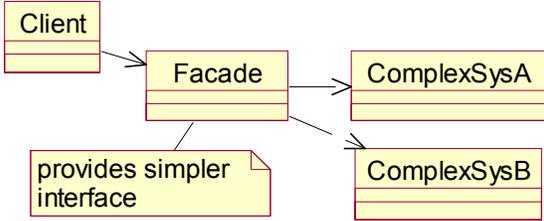|		CREATIONAL PATTERNS
		Notes on the patterns
A b s t r a c t	**F a c t o r y**	**Indicators in analysis:** Different cases exist that require different implementations of sets of rules. **Indicators in design:** Many polymorphic structures exist that are used in pre-defined combinations. These combinations 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.
	B u i l d e r	**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.
F a c t o r y	**M e t h o d**	**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.
O b j e c t	**P o o l**	**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.
P r o t o t y p e		**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.
S i n g l e t o n		**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 should 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.		**A b s t r a c t** **F a c t o r y**
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.		**B u i l d e r**
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.		**F a c t o r y** **M e t h o d**
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.		**O b j e c t** **P o o l**
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.		**P r o t o t y p e**
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.	```\nPSEUDO CODE (Java)\n\nclass Singleton {\n public static Singleton Instance();\n protected Singleton();\n private static _instance= null;\n\n Singleton Instance () {\n if _instance== null)\n _instance= new Singleton;\n return _instance\n} }\n```	**S i n g l e t o n**

Design Pattern Matrix

	STRUCTURAL PATTERNS
	Notes on the patterns
A d a p t e r	**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.
B r i d g e	**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.
C o m p o s i t e	**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
F a ç a d e	**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.
P r o x y V i r t u a l	**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.	**Client** → **TargetAbstraction** + operation() **Adapter** + operation() ◇— **ExistingClass** + itsOperation() operation: existingclass->itsOperation	**A d a p t e r**
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.	*Abstraction* + *operation()* ◇— *Implementation* + *opImp1()* + *opImp2()* **Concrete1** **Concrete2** **ImpA** + opImp1() + opImp2() **ImpB** + opImp1() + opImp2() operation() { imp.opImp1() } operation() { imp.opImp2() }	**B r i d g e**
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.	**Client** → *Component* + *operation()* **Leaf** + operation() **Composite** + operation() ◇	**C o m p o s i t e**
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.	**Client** → **Facade** → **ComplexSysA** provides simpler interface → **ComplexSysB**	**F a ç a d e**
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.	**Client** —to proxy→ *Abstract* + *operation()* **VirtualSubject** + operation() ◇— **RealSubject** + operation() realsubject->operation()	**P r o x y -** **V i r t u a l**
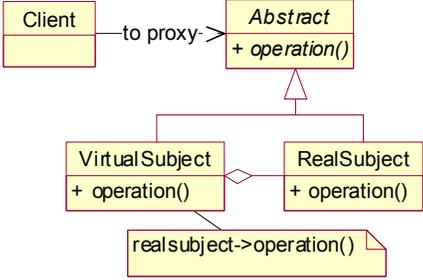
Design Pattern Matrix

	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.		**D e c o r a t o r**
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.		**P r o x y | N e w** **F u n c t i o n**
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.		**S t a t e**
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.		**S t r a t e g y**
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.		**T e m p l a t e** **M e t h o d**
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.		**V i s i t o r**

Design Pattern Matrix

| DECOUPLING PATTERNS	
	Notes on the patterns
Chain of Re-sponsibility	**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	Ac-cessibility**	**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 Re- sponsibility**
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 | Ac- cessibility Access-**

Design Pattern Matrix

Model-View-Controller
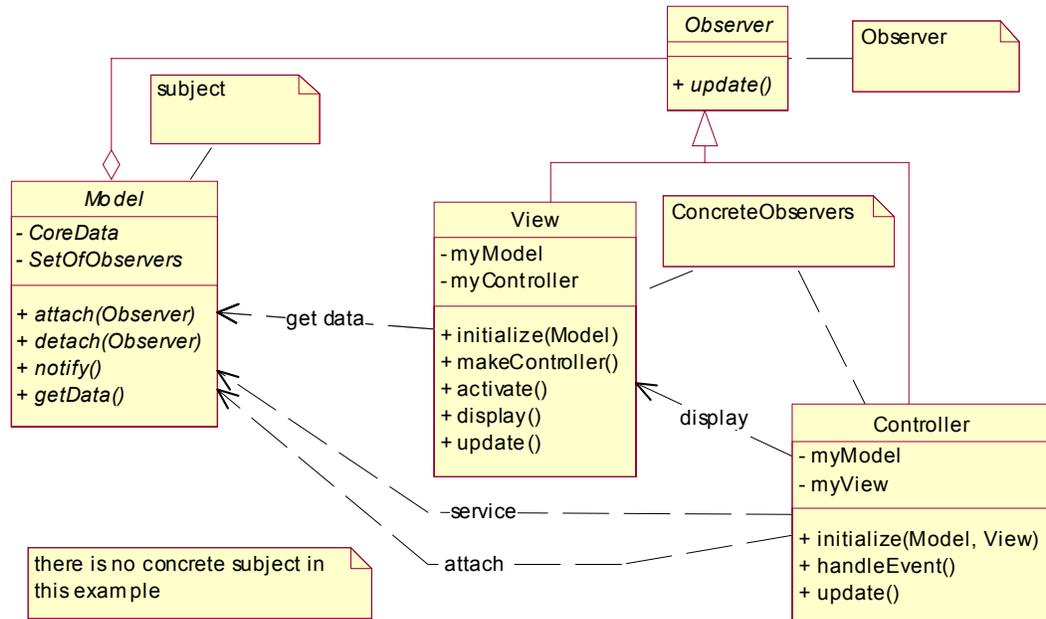
The Model-View-Controller (MVC) is primarily used when building GUIs. However, it can be used anytime you have an interactive type system. It is used to de-couple your data, your presentation of the data and the logic for handling the events from each other.

The Analysis Matrix

Use the Analysis matrix to collect variation between the different cases you have to deal with. Do not try to make designs from it while you are collecting it. However, the consistencies and inconsistencies between the cases will give you clues. Remember, we will implement the rows as Strategies, Proxies, Decorators, Bridges, etc. We will implement the columns with the Abstract Factory.

	Case 1	Case 2	Case 3	Case 4
one thing that is varying	These are the concrete implementations for the ways to whatever is varying that is listed on the left.			
another thing that varies	These are the concrete implementations for the ways to whatever is varying that is listed on the left.			
still another thing that varies	These are the concrete implementations for the ways to whatever is varying that is listed on the left.			
. . .	**. . .**			

	Case 1	Case 2	Case 3	Case 4
one thing that is varying	These implementations are used when have case 1	These implementations are used when have case 2	These implementations are used when have case 3	These implementations are used when have case 4
another thing that varies				
still another thing that varies				
. . .				

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?

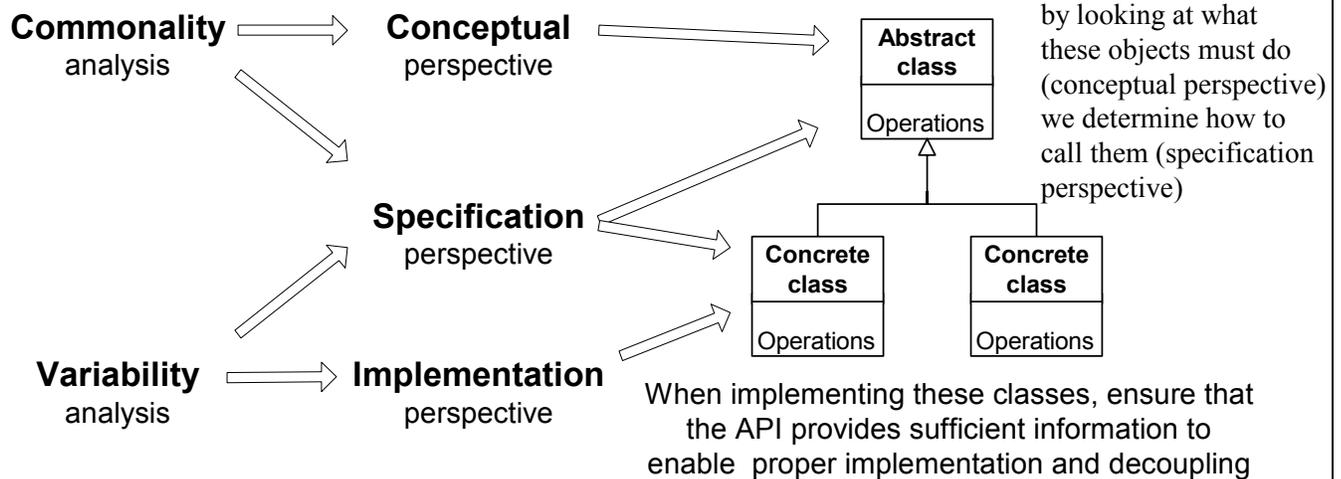|	
Adapter –	do we have the right stuff but the wrong interface? (Used to fit classes into patterns as well)
Composite –	do we have units and groups and want to treat them the same way?
Façade –	do we want to simplify, beautify, or OO-ify an existing class or syb-system?

Are we trying to decouple things?

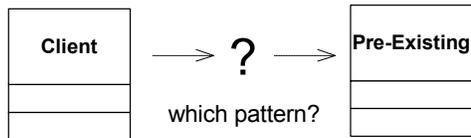|	
Observer –	do various entities need to know about events that have occurred?
Chain of Responsibility –	do we have different objects that can do the job but we don't want the client object know which is actually going to do it?
Iterator –	do we want to separate the collection from the client that is 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.

Commonality analysis → **Conceptual** perspective →

Specification perspective

Variability analysis → **Implementation** perspective

Abstract class / Operations

Concrete class / Operations

Concrete class / Operations

by looking at what these objects must do (conceptual perspective) we determine how to call them (specification perspective)

When implementing these classes, ensure that the API provides sufficient information to enable proper implementation and decoupling

Design Pattern Matrix
Comparing patterns

Comparing Adapter, Façade, Proxy and Decorator

Client	→ ? →	Pre-Existing

which pattern?

	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").
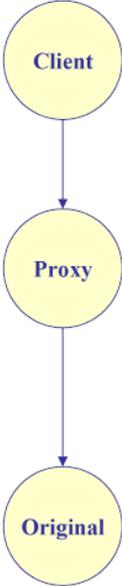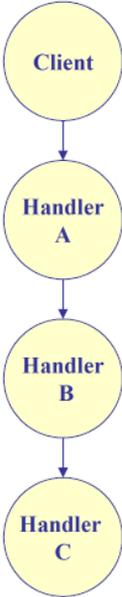
Comparing Decorator, Proxy, and Chain of Responsibility
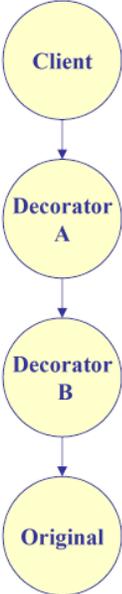
Experts in Agile Training / Coaching

All of our staff are practitioners who know how to teach and coach. This enables us to provide knowledge transfer at all levels: training, coaching and staff supplementation. We provide a wide variety of services as well as approaches to agility.

Quality Training

Design Patterns   Effective ASP.NET   Use Cases   On-Site Courses!

Lean Software Development

ScrumMaster Certification by Net Objectives   **Instructor Led / Web-**

Test-Driven Development   **Based Training for**

Extreme Programming, Effective Programming, RUP   **Languages**

"If I were tasked with bringing in an outside design course, Net Objectives' would be on the top of my list" - John Terrell, Microsoft

"Two things in life are certain: death and taxes" – Ben Franklin

"In the information age, three things in life are certain – death, taxes, and *requirements will change*" – Alan Shalloway

Lean Software Development and ScrumMaster Certification by Net Objectives

We offer training in both Lean Software Development and in Scrum. Our Lean Software Development training is based on Mary and Tom Poppendieck's work. Our Scrum training includes certification in being a ScrumMaster. We also provide training in how to be a Product Champion (playing the role of the customer) in an agile project.

Agile Development Best Practices

In simple terms, an Agile Project is one that is predicated on making sure it is always doing the right thing, not merely following a plan that has since gone out of date. The cornerstone of this approach is getting and adapting to feedback as the project progresses. Most projects can't do this, so they fall further behind and either fail or provide inferior products. Changes are of many types, but the most common (and important) changes are to the system's requirements. This course analyzes what it means to be an agile project, and provides a number of best practices that provide and/or enhance agility. Different agile practices (including RUP, XP and Scrum) are discussed.

Agile Use Case Analysis

This 3-day course provides theory and practice in writing use cases, an understanding of how use cases fit into software development, and the use of the "ever unfolding story" to incrementally refine the use cases into a robust requirements model. The course is largely based on Alistair Cockburn's book "Writing Effective Use Cases" - winner of the Jolt Productivity Award for 2001. As a certified member of Cockburn and Associates, we are one of the few companies authorized to teach it.

Design Patterns Explained: A New Perspective on Object-Oriented Design

This course goes beyond merely teaching several design patterns. It also teaches the principles and strategies that make design patterns good designs. This enables students to use these advanced design techniques in their problems whether design patterns are even present. After teaching several patterns and the principles underneath them, the course goes further by showing how patterns can work together to create robust, flexible, maintainable designs.

Test-Driven Development

The practice of Agile Software Development requires, among other things, a high degree of flexibility in the coding process. As we get feedback from clients, stakeholders, and end users, we want to be able to evolve our design and functionality to meet their needs and expectations. This implies an incremental process, with frequent (almost constant) change to the code we're working on. Refactoring, the discipline of changing code without harming it, is an essential technique to enable this process. Unit testing, which ensures that a given change has not caused an unforeseen ripple effect in the system, is another.

Effective Object-Oriented Programming: Editions for Java, C++, C# and VB.NET

These courses take programmers who understand the syntax of the language but who aren't taking advantage of object-oriented development methods into the object-oriented world.

Get more info! Call Mike Shalloway at 404-593-8375	275 118th Ave SE Ste. 115 Bellevue, WA 98005	mike.shalloway@netobjectives.com www.netobjectives.com