

## Design Pattern Matrix

CREATIONAL PATTERNS	
Pattern	Notes on the patterns
<b>Abstract factory</b>	<p><b>Indicators in analysis:</b> Different cases exist that require different implementations of common rules.</p> <p><b>Indicators in design:</b> 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.</p> <p><b>Indication pattern is not being used when it should be:</b> A variable is used in several places to determine which object to instantiate.</p> <p><b>Relationships involved:</b> 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.</p>
<b>Builder</b>	<p><b>Indicators in analysis:</b> 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.</p> <p><b>Indicators in design:</b> You want to hide the implementation of instantiating complex object, or you want to bring together all of the rules for instantiating complex objects.</p>
<b>Factory Method</b>	<p><b>Indicators in analysis:</b> There are different commonalities whose implementations are coordinated with each other.</p> <p><b>Indicators in design:</b> 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.</p> <p><b>Field notes:</b> 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.</p>
<b>Prototype</b>	<p><b>Indicators in analysis:</b> There are prototypical instances of things.</p> <p><b>Indicators in design:</b> When objects being instantiated need to look like a copy of a particular object. Allows for dynamically specifying what our instantiated objects look like.</p>
<b>Singleton</b>	<p><b>Indicators in analysis:</b> There exists only one entity of something in the problem domain that is used by several different things.</p> <p><b>Indicators in design:</b> 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.</p> <p><b>Field notes:</b> You can get much the same function as Singletons with static methods. Therefore, the Singleton should be used only when statics don't work well. This occurs when you need to control when the class is instantiated (that is, static members are allocated). Another case is if you want to use polymorphism on the Singleton.</p>

## Design Pattern Matrix

CREATIONAL PATTERNS		
How it is implemented	Class Diagram/Implementation	Pattern
<p>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 run-time type identification (RTTI) can be used to instantiate the correct objects.</p>	<pre> classDiagram     class Client     class Window     class WidgetFactory {         +createWindow()         +createScrollBar()     }     class MotifFactory {         +createWindow()         +createScrollBar()     }     class PMFactory {         +createWindow()         +createScrollBar()     }     class PMWindow     class MotifWindow     class ScrollBar     class PMScrollBar     class MotifScrollBar      Client --&gt; Window     Client --&gt; WidgetFactory     Client --&gt; MotifFactory     Window &lt; -- PMWindow     Window &lt; -- MotifWindow     ScrollBar &lt; -- PMScrollBar     ScrollBar &lt; -- MotifScrollBar     WidgetFactory &lt; -- PMFactory     MotifFactory &lt; -- PMFactory     </pre>	<b>Abstract factory</b>
<p>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.</p>	<pre> classDiagram     class Director {         +construct()     }     class Builder {         +buildStep1()         +buildStep2()         +getObject()     }     class ConcreteBuilder1 {         +buildStep1()         +buildStep2()         +getObject()     }     class ConcreteBuilder2 {         +buildStep1()         +buildStep2()         +getObject()     }      Director --&gt; Builder     Builder &lt; -- ConcreteBuilder1     Builder &lt; -- ConcreteBuilder2     </pre>	<b>Builder</b>
<p>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 <i>new</i> command to instantiate the proper object.</p>	<pre> classDiagram     class Document     class Application {         +createDocument()     }     class MyDoc     class MyAp {         +createDocument()     }      Document &lt; -- MyDoc     Application &lt; -- MyAp     Application &lt; -- Application     </pre> <p>Note: in this example createDocument is called a factory method. Application is not a factory object.</p>	<b>Factory Method</b>
<p>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.</p>	<pre> classDiagram     class Client     class Prototype {         +clone()     }     class ConcretePrototype1 {         +clone()     }     class ConcretePrototype2 {         +clone()     }      Client --&gt; Prototype     Client --&gt; ConcretePrototype1     Client --&gt; ConcretePrototype2     Prototype &lt; -- ConcretePrototype1     Prototype &lt; -- ConcretePrototype2     </pre>	<b>Prototype</b>
<p>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.</p>	<p>PSEUDO CODE (if C++, _instance should be pointer)</p> <pre> class Singleton {     public static Singleton Instance();     protected Singleton();     private static _instance= null;      Singleton Instance () {         if _instance== null             _instance= new Singleton;         return _instance     } } </pre>	<b>Singleton</b>

## Design Pattern Matrix

STRUCTURAL PATTERNS	
Pattern	Notes on the patterns
<b>Adapter</b>	<p><b>Indicators in analysis:</b> 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.</p> <p><b>Indicator in design:</b> 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.</p> <p><b>Field notes:</b> 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.</p>
<b>Bridge</b>	<p><b>Indicators in analysis:</b> There are a set of related objects using another set of objects. This second set represents an implementation of the first set.</p> <p><b>Indicators in design:</b> There is a set of derivations that use a common set of objects to get implemented.</p> <p><b>Indication pattern is not being used when it should be:</b> There is a class hierarchy that has redundancy in it, in particular, 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.</p> <p><b>Relationships involved:</b> 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.</p> <p><b>Field notes:</b> 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.</p>
<b>Composite</b>	<p><b>Indicators in analysis:</b> 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).</p> <p><b>Indicators in design:</b> Some objects are comprised of collections of other objects, yet we want to handle all of these objects in the same way.</p> <p><b>Indication pattern is not being used when it should be:</b> The code is distinguishing between whether a single object is present or a collection of objects is present.</p>
<b>Façade</b>	<p><b>Indicators in analysis:</b> A complex system will be used which will likely not be utilized to its full extent.</p> <p><b>Indicators in design:</b> 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.</p> <p><b>Indication pattern is not being used when it should be:</b> Many people on a team have to learn a new system although each person is only using a small aspect of it.</p> <p><b>Field notes:</b> Not usually used for encapsulating variation, but different facades derived from the same abstract class can encapsulate different sub-systems. This is called an <i>encapsulating façade</i>.</p>
<b>Proxy – virtual</b>	<p><b>Indicators in analysis and design:</b> Performance issues (speed or memory) can be foreseen because of the cost of having objects around before they are actually used.</p> <p><b>Indication pattern is not being used when it should be:</b> Objects are being instantiated before they are actually used and the extent of this is causing performance problems.</p> <p><b>Field notes:</b> This pattern often comes up to solve scalability issues or performance issues that arise after a system is working.</p>

## Design Pattern Matrix

STRUCTURAL PATTERNS		
How it is implemented	Class Diagram	Pattern
Contain the existing class in another class. Have the containing class match the required interface and call the contained class's methods	<pre> classDiagram     class Client     class TargetAbstraction {         + operation()     }     class Adapter {         + operation()     }     class ExistingClass {         + itsOperation()     }     Client --&gt; TargetAbstraction     Adapter -- &gt; TargetAbstraction     Adapter --&gt; ExistingClass     note for Adapter "operation: existingclass-&gt;itsOperation"         </pre>	<b>Adapter</b>
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.	<pre> classDiagram     class Abstraction {         + operation()     }     class RefinedAbstraction     class Implementation {         + opImp()     }     class Imp_A {         + opImp()     }     class Imp_B {         + opImp()     }     Abstraction &lt; -- RefinedAbstraction     Implementation &lt; -- Imp_A     Implementation &lt; -- Imp_B     Abstraction o-- Implementation     note for Abstraction "imp-&gt;opImp"         </pre>	<b>Bridge</b>
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.	<pre> classDiagram     class Client     class Component {         + operation()     }     class Leaf {         + operation()     }     class Composite {         + operation()     }     Client --&gt; Component     Component &lt; -- Leaf     Component &lt; -- Composite     Composite o-- Component         </pre>	<b>Composite</b>
Define a new class (or classes) that has the required interface. Have this new class use the existing system.	<pre> classDiagram     class Client     class Façade     class ComplexSysA     class ComplexSysB     Client --&gt; Façade     Façade --&gt; ComplexSysA     Façade --&gt; ComplexSysB     note for Façade "provides simpler interface"         </pre>	<b>Façade</b>
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.	<pre> classDiagram     class Client     class Proxy_Virtual {         + operation()     }     class Abstract {         + operation()     }     class RealSubject {         + operation()     }     Client --&gt; Proxy_Virtual     note for Client "to proxy"     Proxy_Virtual &lt; -- Abstract     Proxy_Virtual --&gt; RealSubject         </pre>	<b>Proxy - virtual</b>

## Design Pattern Matrix

BEHAVIORAL PATTERNS	
Pattern	Notes on the patterns
<b>Decorator</b>	<p><b>Indicators in analysis:</b> There is some action that is always done, there are other actions that may need to be done.</p> <p><b>Indicators in design:</b> 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.</p> <p><b>Indication pattern is not being used when it should be:</b> There are switches that determine if some optional function should be called before some existing function.</p> <p><b>Variation Encapsulated:</b> The functionality to be added before or after an existing function.</p> <p><b>Field notes:</b> This pattern is used extensively in the JFC for file handling.</p>
<b>Proxy – adding function</b>	<p><b>Indicators in design:</b> We need some particular action to occur before some object we already have is called.</p> <p><b>Indication pattern is not being used when it should be:</b> 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.</p> <p><b>Field notes:</b> Proxies are useful to encapsulate a special function that is sometimes used prior to calling an existing object.</p>
<b>State</b>	<p><b>Indicators in analysis and design:</b> We have behaviors that change, depending upon the state we are in.</p> <p><b>Indication pattern is not being used when it should be:</b> 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.</p> <p><b>Field notes:</b> We define our classes by looking at the following questions:</p> <ol style="list-style-type: none"> <li>1. What are our states?</li> <li>2. What are the events we must handle?</li> <li>3. How do we handle the transitions between states?</li> </ol>
<b>Strategy</b>	<p><b>Indicators in analysis:</b> There are different implementations of a business rule.</p> <p><b>Indicators in design:</b> You have a place where a business rule (or algorithm) changes.</p> <p><b>Indication pattern is not being used when it should be:</b> 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.</p> <p><b>Relationships involved:</b> 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).</p> <p><b>Variation encapsulated:</b> The different implementations of the business rules.</p> <p><b>Field notes:</b> 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 Algorithm to use or the Context asks a factory (or configuration type) object for the correct algorithm object to use.</p>
<b>Template</b>	<p><b>Indicators in analysis:</b> There are different procedures that are followed that are essentially the same, except that each step does things differently.</p> <p><b>Indicators in design:</b> You have a consistent set of steps to follow but individual steps may have different implementations.</p> <p><b>Indication pattern is not being used when it should be:</b> Different classes implement essentially the same process flow.</p> <p><b>Field notes:</b> The template pattern is most useful when it is used to abstract out a common flow between two similar processes.</p>
<b>Visitor</b>	<p><b>Indicators in analysis and design:</b> 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.</p> <p><b>Variation encapsulated:</b> A set of tasks to run against a set of derivations.</p> <p><b>Field notes:</b> This is a useful pattern for writing sets of tests that you can run when needed.</p>

NOTE: The Decorator and Proxy patterns are classified as Structural patterns by the GoF. Since they both add functionality, however, instead of simply

## 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 <i>operation</i> method. Optionally, call the contained object's <i>operation</i> method first, then do your own special function.	
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.	
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 <i>get</i> routines, of course).	
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.	
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.	
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.	

## Design Pattern Matrix

DECOUPLING PATTERNS	
Pattern	Notes on the patterns
<b>Chain of responsibility</b>	<p><b>Indicators in analysis:</b> We have the several actions that may be done by different things.</p> <p><b>Indicators in design:</b> 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.</p> <p><b>Field notes:</b> 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.</p>
<b>Iterator</b>	<p><b>Indicators in analysis and design:</b> 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.</p> <p><b>Indication pattern is not being used when it should be:</b> Changing the underlying structure of a collection (say from a vector to a composite) will affect the way the collection is iterated over.</p> <p><b>Variations encapsulated:</b> Type of collection used.</p> <p><b>Field notes:</b> The Iterator pattern enables us to defer a decision on which type of collection structure to use.</p>
<b>Mediator</b>	<p><b>Indicators in analysis and design:</b> Many objects need to communicate with many other objects yet this communication cannot be handled with the observer pattern.</p> <p><b>Indication pattern is not being used when it should be:</b> The system is tightly coupled due to inter-object communication requirements.</p> <p><b>Field notes:</b> When several objects are highly coupled in the way they interact, yet this set of rules can be encapsulated in one place.</p>
<b>Memento</b>	<p><b>Indicators in analysis and design:</b> The state of an object needs to be remembered so we can go back to it (e.g., undo an action).</p> <p><b>Indication pattern is not being used when it should be:</b> 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.</p> <p><b>Field notes:</b> This pattern is useful only when making copies of the object whose state is being remembered would be inefficient.</p>
<b>Observer</b>	<p><b>Indicators in analysis and design:</b> 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.</p> <p><b>Indication pattern is not being used when it should be:</b> When a new object needs to be notified of an event occurring the programmer has to change the object that detects the event.</p> <p><b>Variation encapsulated:</b> The list of objects that need to know about an event occurring.</p> <p><b>Field notes:</b> This pattern is used extensively in the JFC for event handling and is supported with the <i>Observable</i> class and <i>Observer</i> interface.</p>
<b>Proxy – access-ability</b>	<p><b>Indicators in analysis and design:</b> 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).</p> <p><b>Indication pattern is not being used when it should be:</b> The use of an object and the set-up of the connection to the object are found together in more than one place.</p> <p><b>Field notes:</b> 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.</p>

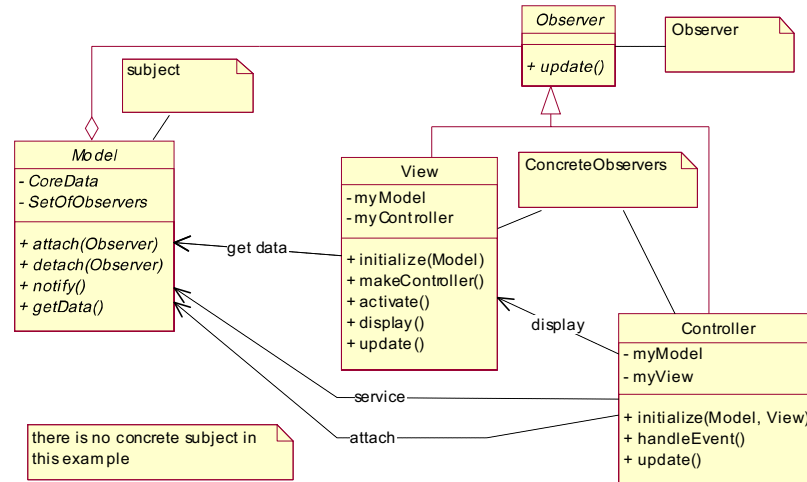
## Design Pattern Matrix

DECOUPLING PATTERNS		
How it is implemented	Class Diagram	Pattern
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.	<pre> classDiagram     class Client     class Handler {         +handleRequest()     }     class Handler_A {         +handleRequest()     }     class Handler_B {         +handleRequest()     }     Client --&gt; Handler     Handler &lt; -- Handler_A     Handler &lt; -- Handler_B     Handler --&gt; Handler </pre>	<b>Chain of responsibility</b>
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.	<pre> classDiagram     class Client     class Collection {         +createIterator()         +append()         +remove()     }     class Iterator {         +first()         +next()         +currentItem()     }     class List     class Vector     class IteratorVector     class IteratorList     Client --&gt; Collection     Client --&gt; Iterator     Collection &lt; -- List     Collection &lt; -- Vector     Iterator &lt; -- IteratorVector     Iterator &lt; -- IteratorList     Collection --&gt; Iterator     Iterator --&gt; Collection </pre>	<b>Iterator</b>
Define a central class that acts as a message routing service to all other classes.	<pre> classDiagram     class aColleague     class aMediator     aColleague --&gt; aMediator     aColleague --&gt; aMediator     aColleague --&gt; aMediator     aColleague --&gt; aMediator </pre>	<b>Mediator</b>
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.	<pre> classDiagram     class Originator {         +setMemento(m : Memento)         +createMemento()     }     class Caretaker     class Memento {         +getState()     }     Originator --&gt; Memento     Caretaker --&gt; Memento </pre>	<b>Memento</b>
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.	<pre> classDiagram     class Subject {         +attach()         +detach()         +notify()     }     class Observer {         +update()     }     class ObserverA {         +update()     }     class ObserverB {         +update()     }     Subject --&gt; Observer : attach/detach     Subject --&gt; Observer : notify     Observer &lt; -- ObserverA     Observer &lt; -- ObserverB </pre>	<b>Observer</b>
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.	<pre> classDiagram     class Client     class Abstract {         +operation()     }     class Proxy_Remote {         +operation()     }     class RealSubject {         +operation()     }     Client --&gt; Abstract : to proxy     Abstract &lt; -- Proxy_Remote     Abstract &lt; -- RealSubject     Proxy_Remote --&gt; RealSubject : realSubject-&gt;operation() </pre>	<b>Proxy – access-ability</b>

## Design Pattern Matrix

### MODEL VIEW CONTROLLER and ANALYSIS 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				
...				

## 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 varying rules?
- Bridge – do we have multiple implementations?
- Proxy – do we need to always add some new functionality to something that already exists?
- Decorator – do we have additional functionality we may need to apply, but what we add varies?
- Visitor – do we have new tasks that we will need to apply to our existing classes?

Are you concerned with interfaces, either 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
- Facade – do we want to simplify our interfaces?
- Proxy – do we want to incorporate a rule to access something without affecting any other class?

Are we trying to decouple things?

- Observer – do things 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 who 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.

