Description

"Define a family of algorithms, encapsulate each one, and make them interchangeable" [GoF]. In other words, this pattern must provide an interface and a number of independent implementations of that interface. Each implementation performs some specific function, can be fully replaced by another implementation and is independent of others and from the enclosing design. A good real world example of a strategy pattern would be a credit card purchase. When you buy something with a credit card, the type of credit card doesn't matter. All credit cards have a magnetic strip that has encoded information in it. The strip, and what it contains, represent the 'interface' and the type of card would be the 'implementation'. Each credit card can be replaced by any other and all are fully independent of each other.   
  
This pattern is one of my favorite patterns and is in almost every design I have ever done. In fact, I once took this to an extreme and designed a whole system using the strategy pattern for every class. I combined this with a specialized classloader and was able to create an extremely flexible system where I could replace any part of it with another implementation at runtime and could dynamically adjust the system depending on real time factors. As an example, I can insert debugging statements or change caching strategies and inject those classes into the system as a special strategy. While the system was flexible in the extreme, it did suffer from various subtle bugs that were introduced by implementations where the associated contract was underspecified (see the consideration section).   
  
When to use (and when not to!)

In applications, this pattern is rarely employed because application designers generally know the scope of the business rules (ie logic) and don't really need the ability to swap in new or unknown rules after the system has been built. Any changes to those rules will generally be part of a larger project change and the overhead of switching to newer rules is fairly minimal. I disagree with this and think the strategy pattern can bring advantages to an application, but that's traditionally the thinking in application design.   
  
For a component designer however, the situation changes radically because the component designer has no idea in what context the component will be used. The component designer needs to be aware that the logic they build into the component may or may not be applicable to the application. The component must supply a number of pre-built modules that will make the component attractive to the application designer but also allow the application designer to 'plug-in' their own logic to provide the flexibility required by the application designer.   
  
The easiest way to determine whether a strategy pattern will be needed is during the reading of the requirements. If the requirements ask for some logic or some other logic ('or' being the important part), you probably have a very good candidate for the strategy pattern. Likewise, if you find yourself thinking that there are two or more ways of accomplishing some task, a strategy pattern is likely needed.   
  
I have a very simple test to determine when a strategy pattern is required. If I abstracted the logic into method calls defined in some interface, could I create a design that still meets the requirements without including that logic? If that's the case, then I know the logic can be fully replaced with other logic and not affect the component at all - in other words, I have a strategy pattern.   
  
Considerations

The strategy pattern is all about the contract specified by the interface. The biggest mistake in designs using the strategy pattern is in not providing a solid contract for implementing classes. Please note that the method signatures are only a very small part of the contract and that the documentation (both class and method) provides the majority of the contract. The contract needs to specify, in as much detail as possible, the exact scope of what implementation can do, what it should not do, and how it should react to various situations. Without those details, the implementations can vary too widely in the logic they performed and how they react to situations. This variation will cause subtle bugs to start occurring throughout the system because one implementation did or didn't do something. These bugs are generally difficult to tract down because they vary with the implementation used.   
  
Another consideration would be how the implementation is initialized. Initialization of the strategy generally falls into two categories: application or factory-based.   
  
An application-based initialization depends on the application to create the implementation itself and cast it to the interface (which then is presumably used by the application thereafter). This has the advantage of allowing the application to finely tune how implementations are created and can provide a lot of contextual information to the implementation. The downside being that it's harder to swap out the implementation to a different implementation since it's generally hardcoded within the application (and potentially in multiple spots). This generally appears to the application like:

WidgetInterface widget = new ACMEWidget();

widget.doSomething();

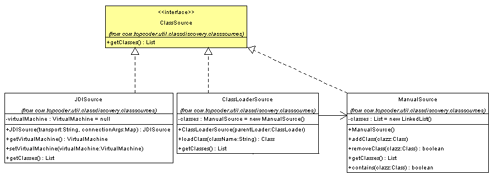
A factory-based initialization defines a factory that contains the different implementations and then the appropriate implementation is returned. The factory can either be managed by the application or potentially managed by some configuration file. The advantage to this method is that is makes switching implementations very easy and provides a central class to manage those implementations. The downside is that it may limit the type of implementations that can be used by the application (if they have a very complex construction that requires application context). This generally looks like:

WidgetInterface widget = WidgetFactory.getWidget();

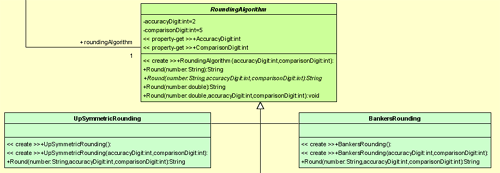
widget.doSomething();

Unfortunately there really are no good guidelines as to when an application-based initialization is preferred over a factory-based one. Which is more appropriate is largely dependant upon the component and the requirements for that component.

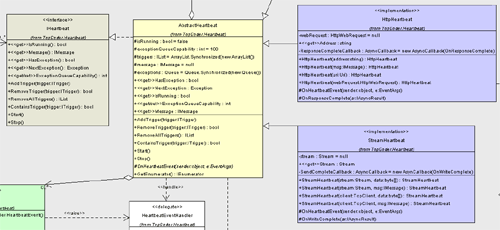
Implementation details  
**Strategy Pattern (interface based)**  
The strategy pattern is very easy to implement. Simply create an interface that fully defines the contract for the implementation to follow. Implementations can then be provided for the interface as specified by the design or requirements.

[](http://www.topcoder.com/i/education/strategy1_lg.gif)  
**Java Class Discovery Component**

The Java Class Discovery component tries to discover the classes that have been loaded by the JVM. This component implements a strategy pattern to allow the application to pick the strategy used for discovering classes. As you see above, the component works with the interface ClassSource and has three implementing classes used to discover classes - each of which implement their own strategy for returning classes from the getClasses() method call. The power of the strategy pattern can be shown looking forward to JDK1.5. A new strategy called "InstrumentSource" could be implemented that uses the new 1.5 Instrument class to discover loaded classes. This new source can then be plugged into the component with no changes to the application.   
  
The downside to this implementation is if the contract for ClassSource ever changes, all implementations would need those changes reflected in them.   
  
**Strategy Pattern (abstract class based)**  
This version of the strategy pattern uses an abstract base rather than an interface to define the contract for implementations. Implementations must then inherit from this class.

[](http://www.topcoder.com/i/education/strategy2_lg.gif)  
**.NET Rounding Factory**

The .NET Rounding Factory component provides various rounding strategies to an application. As shown above, an abstract RoundingAlogrithm class has specified the contract that implementing classes inherit from.   
  
While this variation addresses the issue about changes to the contract (mentioned above), this variation ties the component into a rigid hierarchy structure. So, if an application wishes to create its own rounding algorithm, it cannot use its own class hierarchy to do so.   
  
**Interface/Abstract Base Variation**  
This variation of the strategy pattern provides an abstract base class that implements an interface, which defines the contract. All strategy implementations would then inherit from this base class.   
  
This is a powerful addition to the strategy pattern because it allows the contract of the interface to mutate (in future version of the contract) with very minimal affects on any implementing classes (either those included with the component or more importantly, any provided by the end using application). In fact, utility and/or overloaded functions can easily be added to the interface and implemented in the abstract base without affecting those implementations. Likewise, it allows the designer to minimize the effort in writing an implementation by putting common functionality, which would otherwise be duplicated across many implementations, into the abstract base.   
  
This variation has the benefits of both variations above but minimizes the downsides by allowing the application to choose how it wants to implement. If the application wants to use its own hierarchy, it can by simply implementing the interface. If it wants to be 'future protected' and write simpler implementations, it can implement by inheriting from the abstract base class.

[](http://www.topcoder.com/i/education/strategy3_lg.gif)  
**.NET Heartbeat**

The .NET Heartbeat component provides a 'heartbeat' message to keep connections alive. The strategy pattern is used to allow the component to work with different types of connections - HTTP and a socket stream in this case. Here we have the IHeartbeat interface that defines the contract for implementations. An abstract class, called AbstractHeartbeat that implements the interface, defines many common functions and methods that are generic to all implementations. Finally, we have two implementations that inherit from the abstract class and override the OnHeartbeatEvent method to implement their specific strategy.