**BStrategy Design Pattern**

**Motivation**

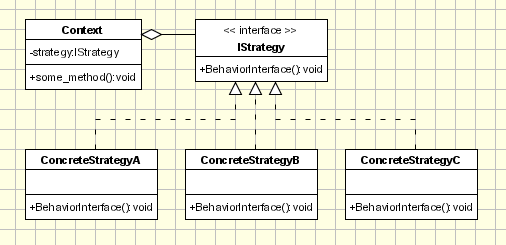
There are common situations when classes differ only in their behavior.

For these cases it is a good idea to isolate the algorithms in separate classes in order to have the ability to select different algorithms at runtime. 

**Intent**

Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

**Implementation**



Participants of this pattern are:

* **Strategy** - defines an interface common to all supported algorithms. Context uses this interface to call the algorithm defined by a **ConcreteStrategy**.
* **ConcreteStrategy** - each concrete strategy implements an algorithm.
* **Context** - contains a reference to a strategy object. It may define an interface that lets strategy accessing its data.

The Context objects contains a reference to the **ConcreteStrategy** that should be used.

When an operation is required then the algorithm is run from the strategy object.

The **Context** is not aware of the strategy implementation.

If necessary, addition objects can be defined to pass data from context object to strategy.   
  
The context object receives requests from the client and delegates them to the strategy object.

Usually the **ConcreteStrategy** is created by the client and passed to the context. From this point the clients interacts only with the context.

**Applicability and Example**

**Switch statements or if-else** chains are candidates for refactoring to a Strategy pattern (as they are for the State pattern).

The difference between the two is their intent: **State** encapsulates data and possibly the transition to other States; a Strategy object usually does not produce other Strategy implementations and hides complex behavior.

The implementation of the Strategy pattern usually follows the classical composition paradigm: Context has a private field reference to a Strategy, while Strategy may be shareable as a **Flyweight** if the Context passes to it the necessary parameters (or even itself) when calling its methods.

The composition of a Strategy object is a valuable alternative to inheritance: Strategy can be think of as a generalization of many patterns that gain their power from favoring composition.

An **Abstract Factory** is in fact a Strategy dedicated to the creation of objects; an **Adapter** allows retrofitting an object as a Strategy for another one; and so on.

The code sample uses hidden Strategy objects for the sorting process of a Collection, in particular for comparing two values.

<?php

/\*\*

\* Using the Strategy design pattern.

\* Defines a behavior for comparing two objects

\* of the Collection.

\*/

**interface** Comparator

{

/\*\*

\* **@return** integer -1 if $a should precede $b

\* 1 if $b should precede $a

\* 0 if considered equal

\*/

**public function** compare($a, $b);

}

/\*\*

\* The Context where the Strategy is employed.

\* Strategy is stored as a private field which can

\* be initialized only one time.

\*/

**class** Collection **implements** Countable

{

**private** $\_elements;

**private** $\_comparator;

**public function** \_\_construct(**array** $elements = **array**())

{

$this->\_elements = $elements;

}

**public function** initComparator(Comparator $comparator)

{

**if** (**isset**($this->\_comparator)) {

**throw new** Exception("A Comparator is already present.");

}

$this->\_comparator = $comparator;

}

**public function** sort()

{

$callback = **array**($this->\_comparator, 'compare');

uasort($this->\_elements, $callback);

}

/\*\*

\* A representation for a clear output.

\*/

**public function** \_\_toString()

{

$elements = **array**();

**foreach** ($this->\_elements **as** $value) {

**if** (is\_array($value)) {

$value = 'Array with ' . count($value) . ' elements';

}

$elements[] = $value;

}

**return** '(' . implode(', ', $elements) . ')';

}

**public function** count()

{

**return** count($this->\_elements);

}

}

/\*\*

\* A ConcreteStrategy that compares via the native operator.

\*/

**class** NumericComparator **implements** Comparator

{

**public function** compare($a, $b)

{

**if** ($a == $b) {

**return** 0;

}

**return** $a < $b ? -1 : 1;

}

}

/\*\*

\* A ConcreteStrategy that compares via the result

\* of the count() function.

\*/

**class** CountableObjectComparator **implements** Comparator

{

**public function** compare($a, $b)

{

**if** (count($a) == count($b)) {

**return** 0;

}

**return** count($a) < count($b) ? -1 : 1;

}

}

// ordering numbers

$numbers = **new** Collection(**array**(4, 6, 1, 7, 3));

$numbers->initComparator(**new** NumericComparator);

$numbers->sort();

**echo** $numbers, "\n";

// ordering Countable objects

$first = **array**(1, 2, 3);

$second = **array**(1, 2, 3, 4);

$third = **new** Collection(**array**(1, 2, 3, 4, 5));

$objects = **new** Collection(**array**($third, $second, $first));

$objects->initComparator(**new** CountableObjectComparator);

$objects->sort();

**echo** $objects, "\n";

**Specific problems and implementation**

* **Passing data to/from Strategy object**

Usually each strategy need data from the context and have to return some processed data to the context. This can be achieved in 2 ways:

* Creating some additional classes to encapsulate the specific data.
* Passing the context object itself to the strategy objects. The strategy object can set returning data directly in the context.

When data should be passed the drawbacks of each method should be analyzed.

For example, if some classes are created to encapsulate additional data, a special care should be paid to what fields are included in the classes.

Maybe in the current implementation all required fields are added, but maybe in the future some new strategy concrete classes require data from context which are not include in additional classes.

Another fact should be specified at this point: it's very likely that some of the strategy concrete classes will not use field passed to the in the additional classes.  
  
On the other side, if the context object is passed to the strategy then we have a tighter coupling between strategy and context.

* **Families of related algorithms**

The strategies can be defined as a hierarchy of classes offering the ability to extend and customize the existing algorithms from an application.

At this point the composite design pattern can be used with a special care. 

* **Optionally Concrete Strategy Objects**

It's possible to implement a context object that carries an implementation for default or a basic algorithm.

While running it, it checks if it contains a strategy object. If not it will run the default code and that's it. If a strategy object is found, it is called instead (or in addition) of the default code.

This is an elegant solution to exposing some customization points to be used only when they are required. Otherwise the clients don't have to deal with Strategy objects.

* **Strategy and Creational Patterns**

In the classic implementation of the pattern the client should be aware of the strategy concrete classes. In order to decouple the client class from strategy classes is possible to use a factory class inside the context object to create the strategy object to be used.

By doing so the client has only to send a parameter (like a string) to the context asking to use a specific algorithm, being totally decoupled of strategy classes.

* **Strategy and Bridge**

Both of the patterns have the same UML diagram. But they differ in their intent since the strategy is related with the behavior and bridge is for structure. Furthermore, the coupling between the context and strategies is tighter that the coupling between the abstraction and implementation in the bring pattern.

**Hot points**

The strategy design pattern splits the behavior (there are many behaviors) of a class from the class itself. This has some advantages, but the main drawback is that a client must understand how the Strategies differ.

Since clients get exposed to implementation issues the strategy design pattern should be used only when the variation in behavior is relevant to them.

**Research Work**

***Investigate about the State Pattern and explore how it differs from the Strategy pattern.***