Department of Computer Science

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Tackling Design Patterns
Chapter 8: Strategy Design Pattern

Contents

8.1 Introduction

In this lecture you will learn all about the Strategy Design Pattern. This pattern has proven to be a very useful pattern. Owing to its elegance it is often used.

We discuss the concept of controlled coupling and why it is considered good practice to control coupling. Most of the design patterns are aimed at controlling coupling in order to enhance maintainability of the code. We illustrate how the Strategy design pattern solves a specific class of problems by applying a standard design solution aimed at controlling coupling.

8.2 Programming preliminaries

8.2.1 Controlling coupling

[3] identify coupling by stating:

> *If changing one module in a program requires changing another module, then coupling exists.*

Coupling impacts negative on the maintainability of a system. The larger the number of modules that needs to be changed to extend or adapt a system, the more difficult it becomes to implement these changes without breaking some code. Coupling occurs when code in one module uses code from another, either by calling a function or accessing some data. It is practically impossible to avoid coupling. It can, however, be controlled. It is important to organise coupling using familiar patterns of dependencies between modules – this is mostly what design patterns are about; i.e. prescribing patterns of dependencies between modules that contributes to the maintainability of the code.

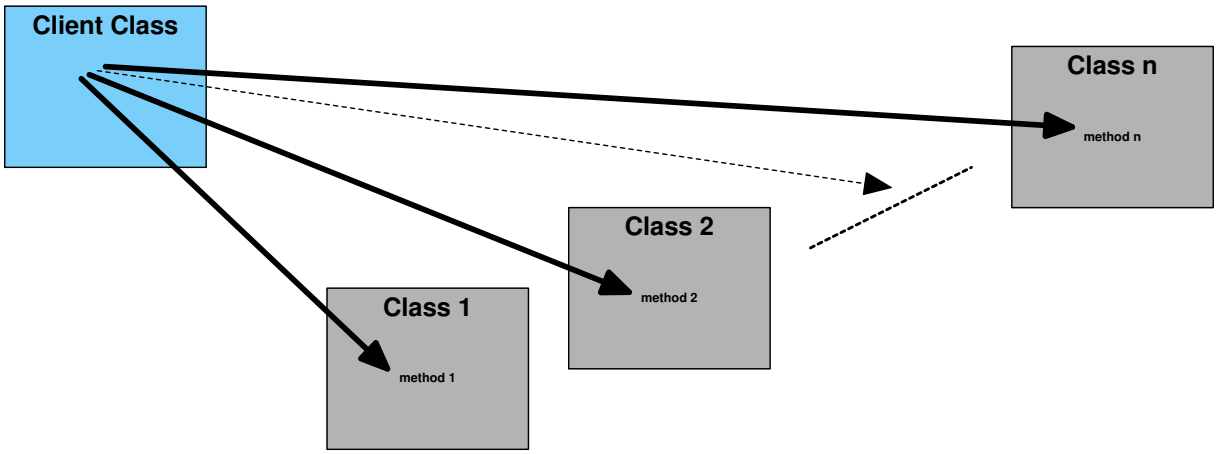

Figure 1: Uncontrolled relations of a client with multiple classes

Defining an interface to serve as a communication hub is a fundamental way to break dependencies and reduce coupling. This strategy is applied in many of the design patterns of which the Strategy design pattern is a very nice example. Figure 1 shows a number

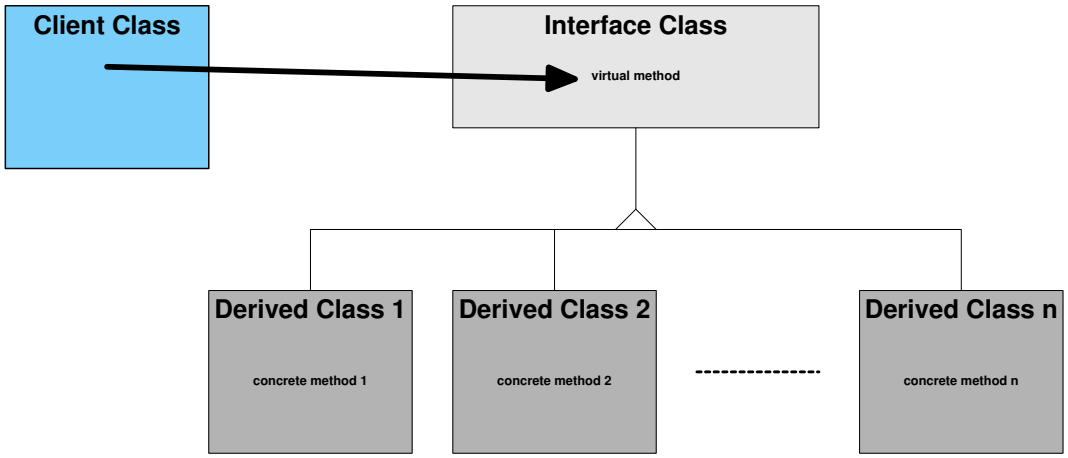

Figure 2: Controlled relation of a client with multiple classes through an interface

of modules who have uncontrolled relations with a client application. Figure 2 shows how this should be organised into a familiar pattern that you will often see in the design patterns.

8.3 Strategy Design Pattern

8.3.1 Identification

Name	Classification	Strategy
Strategy	Behavioural	Delegation
Intent		
Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it ([4]:315)		

8.3.2 Structure

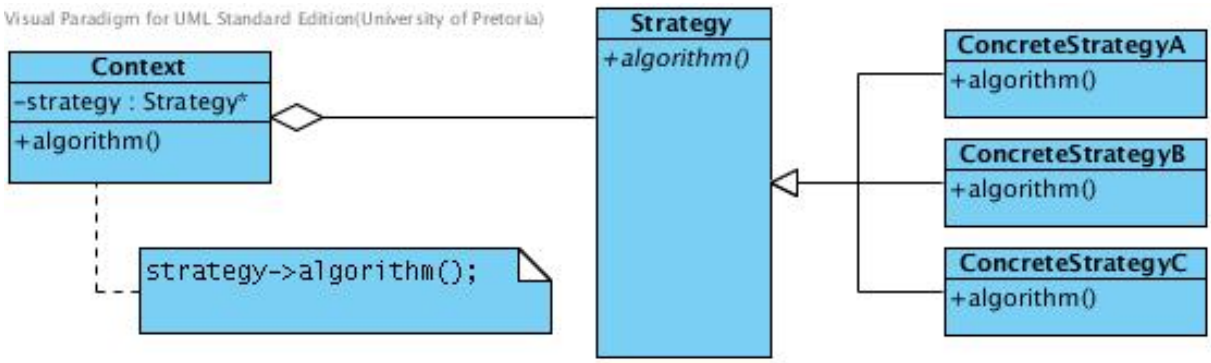

Figure 3: The structure of the Strategy Design Pattern

8.3.3 Problem

An implementation implementing different algorithms to solve the same problem requires uncontrolled coupling with its clients that can be avoided. That is, the implementation consists of a variation of classes for implementing different algorithms to solve a given problem and clients need to couple individually with these classes.

8.3.4 Participants

Strategy

- Declares an interface common to all supported algorithms.
- Context uses this interface to call the algorithm defined by a ConcreteStrategy.

ConcreteStrategy

- Implements the algorithm defined by the Strategy interface.

Context

- Is configured with a ConcreteStrategy object.
- Maintains a reference to a Strategy object.
- May define an interface that lets Strategy access its data.

8.4 Strategy Pattern Explained

8.4.1 Improvements achieved

- Where various classes provide different implementations of the same routine, interface details is encapsulated in a base class, while the implementation details are provided in derived classes. Clients can now couple themselves to the interface, and will be shielded from changes if algorithms need to be changed: no impact when the number of derived classes changes, and no impact when the implementation of a derived class changes.
- Usually the implementation of the Strategy pattern eliminates the need for a complicated conditional statement to select the desired strategy.
- When encapsulating an algorithm in its own class it is possible to define complex algorithm-specific data structures in this class and consequently avoid exposing them to the clients that use the algorithm. This way the implementation of the algorithm is more robust.
- When heuristics are applied to programmatically select an appropriate strategy, this selection my involve the implementation of complex selection structures such as a `switch`-statement or cascading `if`-statements. Encapsulating these in a Context class relieves clients from having to implement them.

8.4.2 Disadvantages

- The coupling between the Strategy interface and the Context class might be wider than always needed. The interface must cater for everything that is needed by each of the strategies. This is the union of the needs of all the strategies. Not all strategies use the whole interface resulting in parameters initialised and never used.

8.4.3 Practical examples

A classic example often used to illustrate the strategy design pattern is to implement a system that offers a method to choose between different algorithms for sorting [1]. This is probably a popular example because most programmers are familiar with the various sorting algorithms and their respective time-space tradeoffs.

There are many other problems for which there are known alternative algorithms to solve the problem, each with their own situational benefits and drawbacks. When a problem requires the system or the user to decide which algorithm should be used to solve the current instance of the problem, the application of the Strategy design pattern is ideal for the implementation of such system. The following are a few practical examples:

- Save files in different formats or compress files using different compression algorithms.
- Provide different routing algorithms in GPS navigational software.
- Provide different validating strategies for input fields.
- Apply different line-breaking strategies to display textual data.
- Apply different routing algorithms to manage network traffic to select to the most effective algorithm based on current traffic patterns.
- Apply different register allocation schemes when compiling code to provide flexibility in targeting code optimisation for different machine architectures.

8.4.4 Implementation Issues

When implementing the pattern one have to provide a way to choose the most appropriate strategy [2]. This can be implemented programmatically or may be implemented to be user-driven. If it is implemented programmatically, the context should implement heuristics to select the most appropriate strategy based on specified criteria. If it is implemented user-driven, the system has to provide an interface for the user to select one of the available strategies. This interface should preferably be implemented by the client. In this case the context only need to provide methods that can be called by the client to change the strategy.

Another thing that needs to be considered is how implemented algorithm is granted access to the data it has to manipulate. The context may pass the data to the algorithm through parameters. If the data structures involved are large, it is desirable to pass a pointer to the data rather than duplicating data. In some cases it may be useful to pass a pointer to the

Context to the algorithm and allow the algorithm to operate directly on the data stored in the Context using a call back mechanism. Alternatively, the strategy can store a reference to its Context, eliminating the need to pass anything a all [4]. This technique couples the Strategy and the Context more closely. The data requirements of the particular problem and its algorithms will determine the best technique.

8.4.5 Common Misconceptions

- When the different strategies are simple methods implemented without encapsulating them in classes that are polymorphic subclasses of an interface, the system implements the desired functionality but is not an application of the Strategy design pattern.

8.4.6 Related Patterns

Factory Method

Both Strategy and Factory Method use delegation through an abstract interface to concrete implementations. However, Strategy performs an operation while Factory Method create an object.

State

The State and Strategy patterns have the same structure and apply the same techniques to achieve their goals. However, they differ in intent. The Strategy pattern is about having different implementations that accomplishes the same result, so that one implementation can replace the other as the strategy requires while the State pattern is about doing different things based on the state, while relieving the caller from the burden to accommodate every possible state.

Flyweight

Strategy objects often makes good flyweights.

8.5 Example

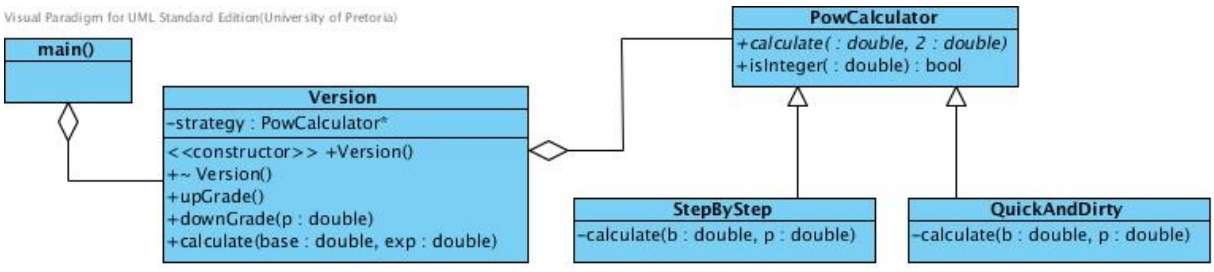

Figure 4: Class Diagram of a system illustrating the implementation of the Strategy design pattern

Figure 4 is a class diagram of an application that implements the strategy design pattern. It implements two different strategies to calculate x^y.

Participant	Entity in application
Context	Version
Strategy	PowCalculator
Concrete Strategies	StepByStep, QuickAndDirty
algorithm()	calculate()
Client	main()

Context

- The `Version` class act as the context.
- The implementation of the `calculate()` method is a redirection to the `calculate()` method of the current concrete strategy. The handle to this method is provided by the `strategy` instance variable of `Version`.
- The `upgrade()` and `downgrade()` methods are provided to enable clients to change the active strategy on the fly.

Stratgey

- `PowCalculator` is an abstract class that implements an interface containing the pure virtual `calculate()` method.
- For the purpose of this application a `isInteger()` method is also defined. This method is used to determine if a given `double` variable holds an integer value. This is used to manipulate the output and also to decide if the step-by-step strategy may be applied.

ConcreteStrategy

- The classes `StepByStep` and `QuickAndDirty` act as concrete strategies.
- The `StepByStep` strategy implements repeated multiplication while the `QuickAndDirty` strategy applies the `math.pow()` function.
- To be able to distinguish between the execution of these strategies, they display their results differently.

Client

- In this application the client has a `Version` object called `application` through which the appropriate implementations of `calculate()` is executed.

8.6 Exercises

1. Write a system that implements the strategy design pattern to allow the user to select different representations of data values that are stored in a text file. Your system should read an unknown number of integer values from the text file and store them in an array in the `Context`. One concrete strategy should write out the values as pairs in set notation. The first coordinate of each pair is an index value while the second coordinate is the value that was read from the file. For example if the file contained the values 5, 7, 2, 12 and 9, the output should be:

{(1,5); (2,7); (3,2); (4,12), (5,9)}

Another strategy should produce a bar chart. For example the above mentioned data should produce the following output:

```
01: XXXXX
02: XXXXXXX
03: XX
04: XXXXXXXXXXXX
05: XXXXXXXXX
```

2. Write a system that can be used for sorting an array of objects using different sorting algorithms. Apply the strategy pattern to allow dynamic switching between algorithms. Implement each algorithm as a template method to enable sorting of different kinds of objects.

References

[1] Judith Bishop. *C# 3.0 design patterns*. O'Reilly, Farnham, 2008.

[2] James W. Cooper. *C# design patterns: a tutorial*. Pearson Education, Inc, Boston, 2003.

[3] Martin Fowler. Reducing coupling. *IEEE Software*, 18(4):102 –104, jul/aug 2001. ISSN 0740-7459.

[4] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns : elements of reusable object-oriented software*. Addison-Wesley, Reading, Mass, 1995.