

Advanced Web Development

by Sophie Novak

FILE	2769_SOPHIE_NOVAK_ADVANCED_WEB_DEVELOPMENT_162370_1688174505.DOCX		
TIME SUBMITTED	18-JAN-2019 10:40AM (UTC+0000)	WORD COUNT	2903
SUBMISSION ID	99392683	CHARACTER COUNT	18275

Advanced Web Development

M6X01086

Sophie Novak (P149298)
UWTSD

Table of Contents

Introduction	2
Design patterns in the development of web applications.....	2
Singleton Pattern	3
Strategy design pattern.....	5
References	7
Appendix	9

Introduction

Design patterns in the development of web applications

There are numerous different design patterns within the development of a web application, each contain various concepts. But before a design pattern can be used, it is essential for a developer to know and understand object-oriented programming (OOP) to be able to apprehend each of the design patterns [1]. The ultimate objective of using a design pattern is to aid with designing an answer which could be positively repeated and reused. Furthermore, this suggests that the specific design pattern used does not have to adapt only to one web application, but it will have a larger code sustainability and understanding [2]. The 'Gang of Four' which is created by four authors; Eric Gamma, Richard Helm, Ralph Johnson and John Vlissides, wrote a book called "Design Patterns: Elements of Reusable Object-Oriented Software". It consists of 23 design patterns which are divided into three categories which are:

- Creational, which is the process of object creation, it also deals with object creation systems are is used in applications when a simple form of object creation may result in design issues or complexity of code. These design patterns consist of two main features. The first feature includes, encapsulating which of the concrete class the application applies. The second feature is, concealing how the instances of the concrete class is produced and integrated together. Object-creational patterns change a section the object creation into an additional object, and class-creational patterns change the creation of the objects into subclasses. Abstract, factory and singleton are all types of creation pattern designs.

- Behavioural, which is seen as the interaction between objects and classes and it also administers responsibility. These patterns are concerned with recognising interactions between objects, they also impact how state and behaviour go throughout the application. Finally, through optimising how states and behaviours are taken and changed, the developer can simplify, optimise and expand the maintainability of an application. Command and strategy are both types of behavioural design patterns.

- Structural, which deals with the class and object composition. They are concerned with how classes and objects are collected to create larger structures. Decorator is a type of structural design pattern.

Singleton Pattern

The singleton design pattern has been applied to many systems including PHP, Java and .NET [3]. Unlike other patterns, the singleton class pattern can only access itself, other classes are unable to access the singleton class pattern. All the attributes, properties and methods within the class are marked as static, this suggests that they are able to be accessed through the class name, there is no need to instantiate an object from the class. As they are static there is only one instance of the attributes. Numerous Object-Oriented Programmers view static classes as a bad thing for a variety of reasons, including, they are therefore global data which can be used anywhere in the code. They also result in high coupling, meaning if the class name is changed all the code using it must be altered too. Static classes should only be used for “pure” functions meaning it operates only on input arguments which provides a result based on those alone, it does not depend on any global state. For the limitation to be accomplished, the ‘__construct()’ method has to be defined as private [4]. The class which is based on the singleton pattern only initialises one instance of the class and issues access to the particular object every time it is obtained, usually through a ‘static’ method known as ‘getInstance()’ [5]. The singleton pattern must ensure that the class only has one instance, but an instance is able to be replaced with another instance, it does not have to be consistent throughout the application. Singleton classes are able to have subclasses, which is helpful when the developer wishes to alter the behaviour of the class [6].

The singleton design pattern works to apply the mathematical concept, it does this though limiting the expression of a class to one object [7]. This pattern is usually applied to applications which only have one existing object, using the singleton pattern in this instance would make the application perform more efficiently. It is mainly applied to an application to make the process of passing data through the application much easier [8]. A singleton design pattern must provide a global point of access to it also, using a global variable will make an object available and easy to access. However, a global variable does not prevent the user from creating various objects [9]. An improved resolution to this is to ensure that the class itself is responsible to supervise its instances, this way the class can ensure that no other instances are being created [9]. The singleton design pattern has three attributes, which are [9];

- A singleton class must only have one instance.
- It must create the instance on its own.
- It must allow the system to access the instance.

Checking that the class only has one instance and whether it can create an instance on its own is simpler than checking if it allows the system to access its instance. It is not simple to check whether various calls for the method returns the exact same instance of the singleton class [10]. It should be understood that applications which are based exclusively on relationships which are structural are unable to recognise the singleton design pattern instances accurately. It is reported [6] that an application is using a singleton pattern design when the class has a static reference of the singleton class and that it has a public static method containing a return type of the singleton class. But it is understood that this criterion does not guarantee that the specific class is definitely using the singleton pattern [6]. The detection of a singleton pattern design requires discovering a condition which uses the number of singleton objects to just one, the condition is not easily shown in a UML model and could usually be expressed informally or not expressed at all [11].

The singleton design pattern illustrates the ability to reuse designs and reuse a solution to the issue that ensures there must only be one instance of the certain class. It is possible that the

developers can use alternative versions of the singleton design pattern, which then makes it more difficult for the developers who maintain applications to change and adapt to the classes [12]. A reason why singleton classes are not used is because they contain global data which can be used anywhere in the code. Furthermore, numerous Object-Oriented Programmers view singleton classes as bad for various reasons, such as, they make testing more difficult as they do not commonly pass the singleton object as a method argument, instead they get the instance from the class itself. Finally, the argument that there is only ever need for one instance is often found incorrect, so they must leave the constructor public in case they need more instances than originally thought.  5

Below is a code example of a singleton pattern:

```
class Singleton
{
    private static readonly Singleton _instance = new Singleton ();

    private OtherClass _attributes;

    private Singleton ()
    {
        _attributes = new OtherClass();    }

    public static Singleton Instance
    {
        get
        {
            return _instance;
        }
    }

    public OtherClass
    {
        get
        {
            return _attributes;
        }
    }
}
```

Strategy design pattern

The strategy pattern is known as a typical behavioural pattern, as it specifies a selection of algorithms, condenses them and then makes them unable to change, this pattern allows the algorithm to be varied and cover a wide range of users who can use it [13]. Its aim is to apply the algorithm so that it can vary independently from users which use the pattern, it achieves adaptability by implementing composition instead of inheritance [10]. The strategy design pattern does not use inheritance, instead it uses composition, this is because the roles within the pattern are described as interfaces and are viewed separately from the certain classes which carry out the role of the interface [13]. This gives the user more freedom when having to decouple the roles of the interface and the class that adopts the interface. The roles can get changed without disturbing the classes which use it, the classes are also able to alternate between the roles through modifying the performance without the need of notable code changes [13]. The strategy design pattern allows the operation and exchange of different algorithms or implementations for a specific policy, it formulates the definition of an abstract type [14] (known as an abstract class or interface). Then it represents the strategy and a chain of various concrete subtypes which correlate with the other implementations of the policy. The algorithm selection in the strategy design pattern is commonly carried out during the initialisation of a context instance and can not be altered at all [14].

The strategy design pattern is the most frequent pattern to use when the developers wish for ease when managing numerous analytical approaches to resolve one problem at a time and compare their performance [15]. This design shows the consistency within the data and groups together the numerical methods in singular classes.

The main attributes of the strategy design pattern are; encapsulation and/or classification, and execution of numerous algorithms [10]. The advantages of using a strategy design pattern is that it gives the developer the ability to be able to replace and extend as they wish. Furthermore, it avoids using multiple branch statements which makes it easier to expand [16], also the strategy design pattern increases object cohesion which limits the coupling strategy [7]. Moreover, the algorithms are loosely coupled with context entity, which gives the ability to be changed or replaced without changing the context entity. However, it should be highlighted that the strategy design pattern must not be misused even though it is helpful, this is because the result which is caused by the data disconnection alongside the algorithms could influence the execution of the application [7]. Also, the users must be aware of the other design patterns available and must they must understand how they differ from each other. As the strategy design pattern increased the number of objects within an application which could potentially slow down the application. The strategy design pattern is commonly applied within the following situations [7];

- When the sole difference between various classes is the behaviour performance, so each behaviour is specifically selected through using the strategy design pattern.
- When the algorithms are contrasting in different conditions.
- When the details of the algorithm are unable to view, from a client's point of view, which achieves ultimate independence.

A study [17] which focussed mainly on the strategy design pattern, discovered the reason the strategy pattern allows the restrictions connected to inheritance to be overpowered. The study concluded that 'BAST' is the only environment (that the study knows about) which provides a selection of protocols which can be applied for creating a fault free application. Alongside a fully

working framework which is built solely on design patterns for formulating new protocols from former ones.

An example of code which demonstrates the strategy design pattern is shown below:

```
Public void Bark()
{
    //Does nothing
}

Public void Trick()
{
    //Does nothing
}

void PerformBark()
{
    _dogBark.Bark
}
void PerformTrick()
{
    _dogTrick.Trick
}

Public Spaniel()
{
    _dogBark = new Bark ();
    _dogTrick = new Trick();
}
```

12

13

14

References

- [1] A. Paikens and G. Arnicans, "Use of Design Patterns in PHP-Based Web Application Frameworks," *LATVIJAS UNIVERSITĀTES RAKSTI. DATORZINĀTNE UN INFORMĀCIJAS TEHNOLOĢIJAS*, vol. 733, pp. 53-71, 2008.
- [2] A. Saray, *Professional PHP Design Patterns*, Birmingham, UK: Wrox Press Ltd, 2009.
- [3] S. Dascalu, N. Hao and N. Debnath, "Design patterns automation with template library," in *Proceedings of the Fifth IEEE International Symposium on Signal Processing and Information Technology*, 2005.
- [4] K. McArthur, *Pro PHP: Patterns, Frameworks, Testing and More*, Apress, 2008.
- [5] J. E. Sweat, *PHP Architect's Guide to PHP Design Patterns*, Marco Tabini & Associates, Inc, 2005.
- [6] K. J. Stencel and P. Wegrzynowicz, "Implementation Variants of the Singleton Design Pattern," in *OTM Confederated International Conferences "On the Move to Meaningful Internet Systems"*, Warsaw, Poland, 2008.
- [7] L. Q. W. Q. X. G. K. W. W. X. W. Y. C. S. Huang Bibin, "Application of design patterns in power system transient simulator," in *IEEE International Conference on Computer Science and Automation Engineering (CSAE)*, Zhangjiajie, China, 2012.
- [8] B. S. Panca, S. Mardiyanto and B. Hendradjaya, "Evaluation of software design pattern on mobile application based service development related to the value of maintainability and modularity," in *International Conference on Data and Software Engineering (ICoDSE)*, Denpasar, Indonesia, 2016.
- [9] G.-I. He, S. Wu and J.-P. Yao, "Application of design pattern in the JDBC programming," in *8th International Conference on Computer Science & Education*, Colombo, Sri Lanka, 2013.
- [10] Y. Zhao, L. Song, G. Zhang and J. Boyland, "Permission Assurance for Semantic Aspects of Design Patterns," in *19th Asia-Pacific Software Engineering Conference*, Hong Kong, China, 2012.
- [11] F. Bergenti and A. Poggi, "Improving UML Designs Using Automatic Design Pattern Detection," in *Handbook of Software Engineering and Knowledge Engineering*, Parma, Italy, 2002, pp. 771-784.
- [12] J. Bieman, D. Jain and H. Yang, "OO design patterns, design structure, and program changes: an industrial case study," in *Proceedings IEEE International Conference on Software Maintenance. ICSM 2001*, Florence, Italy, 2001.
- [13] S. Jiang and H. Mu, "Design patterns in object oriented analysis and design," in *IEEE 2nd International Conference on Software Engineering and Service Science*, Beijing, China, 2011.

- [14] A. Christopouloua, E. Giakoumakisa, V. E. Zafeirisa and V. Soukarab, "Automated refactoring to the Strategy design pattern," *Information and Software Technology*, vol. 54, no. 11, pp. 1202-1214, 2012.
- [15] G. K. Zhu, "Applying Software Design Patterns in Electromagnetic Field Simulators [EM Programmer's Notebook]," *IEEE Antennas and Propagation Magazine*, vol. 54, no. 2, pp. 174-179, 2012.
- [16] C. Jiang, "Design and implementation of the flexible form based on strategy pattern," Shanghai, China, 2012.
- [17] B. Garbinato and R. Guerraoui, "Using the Strategy Design Pattern to Compose Reliable Distributed Protocols," in *Proceedings of the Third USENIX Conference on Object-Oriented Technologies and Systems*, Portland, Oregon, 1997.

Appendix

Assignmentdb.php

```
<?php
header("Context-Type:text/html");
echo"<html><head></head><body>";

$dsn = 'mysql:host=localhost;dbname=university';
//$dbname='university';
$dbUName = "root";
$dbPassword = "";

//Careful if an exception is thrown the full connection is displayed including
//the username and password

$dbh = new PDO($dsn, $dbUName, $dbPassword);

foreach($dbh->query('SELECT * FROM students') as $record)
{
    echo "row: id: {$record["Id"]} firstname: {$record["FirstName"]} surname: {$record["Surname"]}
    dob: {$record["Dob"]} course: {$record["Course"]}<br/>";
}

//$searchName = "Bob"; //Pretend user types this in
//
//
////Create prepared statement and specify value for placeholder
//$stmt = $dbh->prepare("SELECT * FROM students WHERE FirstName = :name");
//$stmt->bindParam(':name', $searchName);
//
//if($stmt->execute())
```

```

//{
//  while($record = $stmt->fetch())
//  {
//    echo "row: id: {$record["Id"]} firstname: {$record["FirstName"]} surname:
{$record["Surname"]} dob: {$record["Dob"]} course: {$record["Course"]}<br/>";
//  }
//}

//to use LIKE this is the approach
//$stmt = $dbh->prepare("SELECT * FROM Student WHERE NAME LIKE ?");
//if($stmt->execute(["{$searchName}%"])) //% is a wildcard
//{
//  while($record = $stmt->fetch())
//  {
//    echo "row: id: {$record["Id"]} name: {$record["Name"]} age: {$record["Age"]} <br/>";
//  }
//}

//$EnteredName = $_POST("namedTextBox");

$insertId = "12026";
$insertFirstName = "James";
$insertSurname = "24";
$insertDob = "07/12/1996";
$insertCourse = "Web Development";
$stmt = $dbh->prepare("INSERT INTO students ('Id', 'FirstName', 'Surname', 'Dob', 'Course') VALUES
(':id', ':firstname', ':surname', ':dob', ':course')");
$stmt->bindParam(':id', $insertId);
$stmt->bindParam(':name', $insertFirstName);
$stmt->bindParam(':age', $insertSurname);
$stmt->bindParam(':age', $insertDob);

```

```

$stmt->bindParam(':age', $insertCourse);
if ($stmt->execute(["{$insertId}"]))
{
    echo ("Id added");
}
if ($stmt->execute(["{$insertFirstName}"]))
{
    echo ("First name added");
}
if ($stmt->execute(["{$insertSurname}"]))
{
    echo ("Surname added");
}
if ($stmt->execute(["{$insertDob}"]))
{
    echo ("Dob added");
}
if ($stmt->execute(["{$insertCourse}"]))
{
    echo ("Course added");
}
echo "</body></html>";

```

counter.php

```

<?php

namespace Main\ObserverPattern;

require_once 'Object.php';
require_once 'Login.php';

```

```
use \Main\Object;
```

```
class Counter extends Object implements ILogin
```

```
{
```

```
    // Attributes
```

```
    private $_observers = [];
```

```
    private $_count = 0;
```

```
    private $_limit;
```

```
    // Constructor
```

```
    public function __construct($limit)
```

```
    {
```

```
        $this->_limit = $limit;
```

```
    }
```

```
    // Methods for properties
```

```
    // Getters and Setters
```

```
    // Methods
```

```
    public function Increment()
```

```
    {
```

```
        $this->_count++;
```

```
        echo "Counter incremented to {$this->_count}<br/>";
```

```
        if($this->_count > $this->_limit)
```

```
        {
```

```
            $this->Notify();
```

```
        }
```

```
    }
```

```
    public function Decrement()
```

```

    {
        $this->_count--;
        echo "Counter decremented to {$this->_count}<br/>";
    }

    public function Attach(ILogin $observer)
    {
        $this->_observers[] = $observer;
    }

    public function Detach(ILogin $observer)
    {
        $index = array_search($observer, $this->_observers, true);
        if($index !== false)
        {
            array_splice($this->_observers, $index, 1);
        }
    }

    public function Notify()
    {
        foreach($this->_observers as $observer)
        {
            $observer->Update($this);
        }
    }
}
?>

```

CountLogin.php

```
<?php
namespace Main\ObserverPattern;

require_once 'Object.php';
require_once 'ILogin.php';

use \Main\Object;

class CountListener extends Object implements ILogin
{
    // Attributes
    private $_id;

    // Constructor
    public function __construct($id)
    {
        $this->_id = $id;
    }

    // Methods for properties
    // Getters and Setters

    // Methods

    public function Update(ILoginAble $loginable)
    {
        echo "Count listener {$this->_id} just got updated<br/>";
    }
}
?>
```

ILogin.php

```
<?php

namespace Main\ObserverPattern;

require_once 'ILoginAble.php';

interface ILogin
{
    public function Update(IObservable $observable);
}

?>
```

ILoginAble.php

```
<?php

namespace Main\ObserverPattern;

require_once "ILogin.php";

interface ILoginAble
{
    public function Attach(IObserver $observer);
    public function Detach(IObserver $observer);
    public function Notify();
}

?>
```

LoginPattern.php

```
<?php

require_once 'Counter.php';
```

```

require_once 'CountListener.php';

use \Main\ObserverPattern\Counter;
use \Main\ObserverPattern\CountLogin;

$counter = new Counter(5);
$observer = new CountLogin(2);

$counter->Attach(new CountLogin(1));
$counter->Attach($observer);
$counter->Attach(new CountLogin(3));
$counter->Increment();
$counter->Increment();
$counter->Increment();
$counter->Increment();
$counter->Increment();
$counter->Increment(); // Notify
$counter->Increment(); // Notify
$counter->Decrement();
$counter->Decrement();
$counter->Decrement();
$counter->Detach($observer);
$counter->Increment();
$counter->Increment(); // Notify

/*
 * Alternative: Using the Standard PHP Library (SPL)
 */

require_once 'CounterUsingSpl.php';
require_once 'CountListenerUsingSpl.php';

```

```
use \Main\ObserverPattern\CounterUsingSpl;  
use \Main\ObserverPattern\CountLoginUsingSpl;
```

```
$counter = new CounterUsingSpl(5);
```

```
$counter->Attach(new CountLoginUsingSpl(1));
```

```
$counter->Attach(new CountLoginUsingSpl(2));
```

```
$counter->Increment();
```

```
$counter->Increment();
```

```
$counter->Increment();
```

```
$counter->Increment();
```

```
$counter->Increment();
```

```
$counter->Increment(); // Notify
```

```
?>
```

FINAL GRADE

31 / 100

GENERAL COMMENTS

Instructor

Knowledge & Understanding (50) 26

Quality (20) 2

Login Observer (10) 2

Database Handling (15) 1

Software Testing (5) 0

On the one hand you have provided quite a lot of detail about the Singleton and Strategy pattern but there are a number of instances where I'm not convinced that you understand what you have written.

Alternatively, you may have failed to explain yourself clearly and just need to expand and reword a number of sections of the report.

A small amount of code provided which consists of examples provided in class with small modifications.. Database handling is related to a student table??

Insert values are hard coded ??

Basically there is little to mark here.

I understand that you have missed a number of classes through illness, if you need more support please come and see me. We are here to help you.



Comment 1

What mathematical concept?



Comment 2

English needs improvement



Comment 3

Not sure why this is important or relevant



Comment 4

I think this could have been written more clearly. Are you just saying there should be a requirement for only one object instance of the singleton class?



Comment 5

Not sure this argument always holds true. How about configuration data for an application?



Comment 6

This really needs to be explained in more detail. Does it make sense?



Comment 7

What algorithm?



Comment 8

Ok, I can kind of make sense of this because I know the pattern but it is not well written. Although, I'm not convinced that you understand this.



Comment 9

This really does need to be expanded upon, it sounds like an interesting point but it is not very clear what this actually means.



Comment 10

Why?



Comment 11

English needs improving here!!

PAGE 7



Comment 12

Actually, this example doesn't explain anything about the strategy pattern.



Comment 13

`_dogBark.Bark()`



Comment 14

`_dogTrick.Trick();`

PAGE 8

PAGE 9

PAGE 10

PAGE 11

PAGE 12

PAGE 13

PAGE 14

PAGE 15

PAGE 16

PAGE 17

PAGE 18
