

Assessment 1

by Gareth Thomas

FILE	12687_GARETH_THOMAS_ASSESSMENT_1_162370_937474997.DOCX (104.9K)		
TIME SUBMITTED	18-JAN-2019 02:49PM (UTC+0000)	WORD COUNT	1916
SUBMISSION ID	99424823	CHARACTER COUNT	12080

Advanced Web Development

Computing Portfolio

Gareth Thomas - 1602705

Table of Contents

1. Introduction	2
2. Object Orientated Patterns.....	2
2.2. Dependency Injection	2
2.3. Singleton Design.....	3
2.3.1. Example of Singleton design pattern	3
2.4. Factory Design.....	4
2.4.1. Example of Factory design pattern	4
2.5. Strategy Design	5
2.6. Observer Design	5
2.7. Iterator Design	5
2.8. Composite Design	5
3. Bibliography	6

1. Introduction

This report will investigate the use design patterns used in the development of web applications. Examples will be provided of some of the design patterns available. An PHP web application will be created using object orientated techniques and patterns. This application will feature a functioning login system which will be created using the observer design pattern. A logging system will be created to highlight any failed login attempts by users. Navigation and items will be filtered and displayed through a database using PHPMyAdmin.

2. Object Orientated Patterns

Design patterns allow developers to choose design alternatives that will make the system more reusable and avoid alternatives that sacrifice reusability. The key to an effective application is the reusability of the code and improving the speed of the development of the application. Design patterns can be used to solve most of the day-to-day problems' programmers may deal with. Patterns are proven solutions that have been thoroughly reviewed and tested throughout the years. [1]

Design patterns can be sorted into three different groups -

- Structural Design Patterns – These patterns are mainly concerned with how classes and objects are composed to form larger structures. Instead of creating interfaces or implementations; structural object patterns define ways to compose each object to realise new functionality. This allows the composition to be changed during run-time; which adds more flexibility as this is impossible to perform using a static class composition. Structural classes mainly operate using inheritance to compose interfaces and implantations. Examples of structural patterns include Bridge, Decorator and Facade.
- Behavioural Design Patterns - Behavioural patterns are mostly concerned with the algorithms and assignment of responsibilities between each object. This pattern also will try to describe and manage the communication between objects and classes. The goal of this pattern is to increase flexibility of an application and allow efficient communication between objects and classes. It also provides a way characterize complex control flow; which can be proven to follow during run-time. Examples of behavioural patterns include Observer, Strategy and Iterator.
- Creational Design Patterns – This pattern tries to abstract the instantiation process; in order to provide an application with more independence of how objects are created and represented. Creational patterns allow a system to evolve to depend on object composition rather than inheritance. The main goal of this pattern is to encapsulate all knowledge about the classes the system uses and try to hide how instances are created. The system will only get knowledge from the abstract class. Examples of creational patterns include Singleton, Factory and Builder.

2.2. Dependency Injection

Dependency injection is a programming technique that will make a class independent from its dependencies. It allows for the development of loosely coupled code. Loosely coupled code allows applications to be developed more efficiently and provide more effective maintainability. Loosely coupled code is isolated which allows for additional changes of code without making changes to the application as a whole. As code will be isolated; it will make testing the application an easier task. For example, fake dependencies can be created to enable effective testing. Late binding is available for loosely coupled code; this means that components can be swapped during run-time instead of compile time. [2]

2.3. Singleton Design

The Singleton design pattern will restrict the instantiation of a class to ensure that only one instance of a class will be created because of this, the pattern is considered to be the simplest to use. Due to the simplicity; it is generally over used and could encourage lazy programming as Singleton may start to impose limitations during the design of an application. This pattern may be found in most programming languages such as Java and .NET. This will be useful if there is only one object used to coordinate the actions of the system. A global access point will create to allow access from aspects of the application. It shall make a resource within the application readily available throughout the application. This pattern is generally used to help test centralized management of internal and external resources.

There many various ways to implement Singleton patterns.

- The single private constructor method is the most commonly used implementation method. The constructor prevents other classes from instantiating it and also prevents sub classing. [3]
- A static variable can be created to hold a reference to the created instance.

Many developers appear to be critical of the singleton pattern and critical of any developer using this pattern. Developers believe it is currently over used and encourage the use of other more effective patterns. It is believed that Singleton may begin to interfere with the unit testing of an application. Singleton creates tightly coupled objects and methods which may begin to cause problems when testing begins. Writing testable code may prove difficult to make. Singleton patterns may start to create some hidden dependencies. As the code will be available throughout the application, it may become difficult to keep track of the areas of code that are using the reference. [4]

2.3.1. Example of Singleton design pattern

This example of the singleton design pattern tries to establish the database connection and will restrict the number of instances to one.

```
<?php
class ConnectDb (
{
    private static $instance = null;
    private $conn;

    private $host = 'localhost';
    private $user = 'root';
    private $pass = '';
    private $name = 'AdvWeb';

    // The db connection is established in the private constructor.
    private function __construct()
    {
        $this->conn = new PDO("mysql:host={$this->host};
        dbname={$this->name}", $this->user,$this->pass,
        array(PDO::MYSQL_ATTR_INIT_COMMAND => "SET NAMES 'utf8'"));
    }

    public static function getInstance()
    {
        if(!self::$instance)
        {
            self::$instance = new ConnectDb();
        }

        return self::$instance;
    }

    public function getConnection()
    {
        return $this->conn;
    }
}
?>
```

2.4. Factory Design

The Factory design pattern is a creational design pattern that will define an interface or a super class that is used for creating an object. It lets subclasses decide which class to instantiate [5]. A Factory can be considered an object that creates objects with the use of a constructor. Constructor will rely heavily on class inheritance. When an object is created it will exist somewhere in a list of inherited classes. A factory method will avoid creating additional classes and will create assign properties to the base class. The factory pattern can make code less coupled and more robust. The factory method may be used to save system resources by reusing existing objects instead of creating new ones.

2.4.1. Example of Factory design pattern

In this example, a factory is used to create an automobile object. The factory pattern will make it easier to make changes to the application in the later stages of development and in order to make these changes; the factory object is the only aspect that needs to be changed.

```
class Automobile
{
    private $vehicleMake;
    private $vehicleModel;

    public function __construct($make, $model)
    {
        $this->vehicleMake = $make;
        $this->vehicleModel = $model;
    }

    public function getMakeAndModel()
    {
        return $this->vehicleMake . ' ' . $this->vehicleModel;
    }
}

class AutomobileFactory
{
    public static function create($make, $model)
    {
        return new Automobile($make, $model);
    }
}

// have the factory create the Automobile object
$veyron = AutomobileFactory::create('Bugatti', 'Veyron');

print_r($veyron->getMakeAndModel()); // outputs "Bugatti Veyron"
```

2.5. Strategy Design

The strategy design pattern defines a set of algorithms that could be used interchangeably and can be changed during the runtime. Strategy design is a behavioural design pattern which enables the selection of different algorithms. Multiple objects are created to represent different strategies. These objects perform similar operations and tasks which form the same interface strategy. This strategy could be used in application which allows an option to save files in different formats, offer file compression and has some form of sorting feature within the application. This strategy pattern can be used when there are several implementations for one feature within an application. [6] Strategy patterns are very extendable and allow for loosely coupled algorithms. However, on larger application, the strategy pattern may become a detriment as there could large number of objects which may begin to confuse the development process.

2.6. Observer Design

The Observer design pattern is a behaviour pattern which is responsible for managing relationships between objects. This pattern will define a one-to-many dependency between objects. This allows all objects to be automatically notified and updated when changes are made. There are two types of key objects in this pattern; the subject and the observer. The subject could have any number of observers and the observers will be notified when the subject changes states at any time. This design pattern is useful for messaging services or subscription services. [7]

The observer pattern offers support for loose coupling between objects that interact with each other. This allows for more flexibility for changing requirements and enables the reusability of subject and object classes. Another benefit for the observer class is that it allows the exchange of data between objects without making changes in the observer and subject classes [8]. The 'Open Closed' principle is used; this states that a class should remain open for extensions without changing the class itself. Model View Controllers patterns can use the observer design to decouple the model from the view. The view represents the observer and the model is the subject.

2.7. Iterator Design

The iterator design pattern is a behavioural design pattern used in Java and .Net environments. The iterator pattern provides access to the elements of an aggregate object without exposing the underlying representation. This pattern is responsible for accessing and passing through objects of a collection and put it into an iterator object that will define a standard traversal protocol. A collection is a group of objects that can share the same type. For example, an array can be a collection. Each collection must be provided an iterator to allow it to iterate through objects while trying not to expose its implementation method. The iterator pattern helps provide a uniform interface used to traverse through different collections.

2.8. Composite Design

The Composite design pattern is used when a group of objects need to be treated the same way as a single instance of the same type of the object. Objects within the composite pattern are represented in a tree structure and are treated as individual objects. A class is created to group these objects to allow a simpler way to modify the group of objects. There are four elements that make up the composite pattern. Components declare the interface for objects in the composition and is used to manage and access its child components. The Leaf represents the leaf objects involved in the composition however, a leaf cannot have any children objects. A Composite defines the behaviour for the components that have children objects and implements operations for children objects in the component interface. The client can manipulate objects through the component interface.

3. Bibliography

- [1] R. C. Martin, "Design Principles and Design Patterns," 2000.
- [2] J. Clark, "Dependency Injection: A Practical Introduction," 2013.
- [3] J. Skeet, C# in Depth, Third Edition, Manning Publications, 2013.
- [4] N. Singh, "Let's examine the pros and cons of the Singleton design pattern," freeCodeCamp, 24 March 2018. [Online]. Available: <https://medium.freecodecamp.org/singleton-design-pattern-pros-and-cons-e10f98e23d63>. [Accessed 09 January 2019].
- [5] E. Gamma, R. Helm, R. Johnson and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Pearson Education, 1994.
- [6] F. Eric, B. Bert and R. Elisabeth, Head First Design Patterns, O'Reilly Media, 2004.
- [7] V. Sarcar, Java Design Patterns, Apress, 2015.
- [8] S. Timms, Mastering JavaScript Design Patterns, Packt Publishing, 2014.
- [9] C. Horstmann, Object-Oriented Design & Patterns, Wiley, 2005.
- [10] S. J. Metsker, Design Patterns in C#, Addison Wesley Professional, 2004.
- [11] A. Shalloway, J. R. Trott and J. Trott, Design Patterns Explained: A New Perspective on Object-oriented Design, Addison-Wesley Professional, 2002.
- [12] M. Seemann and S. v. Deursen, Dependency Injection Principles, Practices, and Patterns, Manning Publications, 2018.
- [13] D. R. Prasanna, Dependency Injection: Design Patterns Using Spring and Guice, Manning, 2009.
- [14] G. Zlobin, Learning Python Design Patterns, Packt Publishing Ltd, 2013.
- [15] L. Vogel, "Observer Design Pattern in Java," Vogella, 2016. [Online]. Available: <http://www.vogella.com/tutorials/DesignPatternObserver/article.html>. [Accessed 10 01 2018].

Assessment 1

GRADEMARK REPORT

FINAL GRADE

50/100

GENERAL COMMENTS

Instructor

Knowledge & Understanding (50) 28

Quality (20) 10

Login Observer (10) 5

Database Handling (15) 7

Software Testing (5) 0

You have described a number of design patterns, but you could have spent more time analysing just two of them in more detail. The discussion of the strategy pattern raised more questions than it answered. I think you have an understanding of the patterns.

Login code provided.

A new user is added to database.

Articles are displayed if user logs in.

Search by title or content.

No delete or edit.

Observer pattern not implemented and no database abstraction provided.

No evidence of testing.

I have capped the implementation which was a late

but the report was submitted on time.

PAGE 1

PAGE 2

PAGE 3



Comment 1

You needed to expand upon this.

What does this actually mean?

PAGE 4

PAGE 5

PAGE 6



Comment 2

What does this mean?



Comment 3

How would it be used?



Comment 4

How?



Comment 5

Why?



Comment 6

You need to explain what an iterator is or does



Comment 7

This really needs to be explained more clearly. Perhaps an example could have been provided.

