Charles Yoon

IS 217

Keith Williams

Final Exam

1. **Software Design Pattern -** is a general reusable solution to a commonly occurring problem within a given context in software design. It is not a finished design that can be transformed directly into source or machine code, but it is a description or a template for how to solve a problem that can be used in many different situations.

Design patterns are important because, in short hey help us recurring design problems. It helps us solve problems when programmers are having trouble with their coding during web development. They are very good for a programmer who does not want to rewrite the code and it can reuse solution to fix any problem in software design.

1. **Unit Testing** - is a software development process in which the smallest testable parts of an application, called units, are individually and independently scrutinized for proper operation. Unit testing is often automated but it can also be done manually. This testing mode is a component of Extreme Programming (XP), a pragmatic method of software development that takes a meticulous approach to building a product by means of continual testing and revision.

Unit testing involves only those characteristics that are vital to the performance of the unit under test. This encourages developers to modify the source code without immediate concerns about how such changes might affect the functioning of other units or the program as a whole. Once all of the units in a program have been found to be working in the most efficient and error-free manner possible, larger components of the program can be evaluated by means of integration testing.

1. The relationship between HTML, CSS and JavaScript are: HTML is the structure, CSS is the colorful icing, and JavaScript is the trick candles you can’t blow out that is the relationship. HTML is the script and it just contains the words the actors are supposed to say. CSS places the actors in their starting positions and dresses them in various costumes. JS moves the actors from their original positions to new positions, alters the costumes so that they look different from the way they looked when the play started. It can even retroactively alter the script. Basically, it can take all the “chess pieces” laid out by the HTML and CSS and play with them, shuffling then around, changing the, etc.
2. **The Singleton Pattern** - The purpose of The Singleton Pattern is to create a class with method that creates a new instance of the class if one does not exist. The Singleton Design Pattern is thus known because traditionally, it restricts instantiation of a class to a single object. With JavaScript, singletons serve as a namespace provider which isolates implementation code from the global namespace so-as provide a single point of access for functions.

The Singleton pattern is a creational design pattern that can be implemented by creating a class with a method that creates a new instance of the class if one doesn’t’ exist. For those that already exist, it can simply return a reference to that object. In JavaScript, it is use to serve as a shared resource namespace which isolate implementation code from the global namespace so as to provide a single point of access for functions.

1. **The Factory Pattern -** is similar to other creational patterns. This patterns deals with the problem of creating objects without the need to specify the exact class of object being created. Factory Pattern is an object-oriented creational design pattern that implements the concept of factories and deals with the problem if creating objects without specifying the exact class of an object that is to be created. The essence of this pattern is to define an interface for creating an object, but let the classes that implement the interface decide which class to instantiate. The method for the factory pattern us the let a class defer from instantiation to subclasses.

The purpose of this pattern is to define an interface for creating an object where you allow the subclasses to decide which class to instantiate. This pattern handles the problem by defining a completely separate method for the creation of objects and which sub-classes are able to override so they can specify the ‘type’ of factory product that will be created.

1. **Pub Sub Pattern** – or publish-subscribe is a messaging pattern where the publisher which is the senders of messages, do not program the messages to subscriber which is a receiver. Published messages are characterized into classes without having any knowledge of what or if any if there subscribers. For subscribers, it expresses its interest in one or more classes and it only receive any message that is interesting even without the knowledge of what or in any of the publisher.
2. **Decorator Pattern** – is a structural design pattern that allows behavior to be added to an individual object that either statically or dynamically other object from the same class without affecting the behavior.

The decorator pattern can be used to extend (decorate) the functionality of a certain object statically, or in some cases at run-time, independly of other instances of the same class, provided some groundwork is done at design time. This is achieved by designing a new decorator class that wraps the original class. This wrapping could be achieved by the following sequence of steps:

* Subclass the original "Component" class into a "Decorator" class (see UML diagram);
* In the Decorator class, add a Component pointer as a field;
* Pass a Component to the Decorator constructor to initialize the Component pointer;
* In the Decorator class, redirect all "Component" methods to the "Component" pointer; and
* In the Concrete Decorator class, override any Component method(s) whose behavior needs to be modified.

1. **Decorator Pattern Example:**

**function** vehicle( vehicleType ){

*// properties and defaults*

**this**.vehicleType = vehicleType || 'car',

**this**.model = 'default',

**this**.license = '00000-000'

}

*// Test instance for a basic vehicle*

**var** testInstance = **new** vehicle('car');

console.log(testInstance);

*// vehicle: car, model:default, license: 00000-000*

*// Lets create a new instance of vehicle, to be decorated\*/*

**var** truck = **new** vehicle('truck');

*// New functionality we're decorating vehicle with*

truck.setModel = **function**( modelName ){

**this**.model = modelName;

}

truck.setColor = **function**( color ){

**this**.color = color;

}

*// Test the value setters and value assignment works correctly*

truck.setModel('CAT');

truck.setColor('blue');

console.log(truck);

*// vehicle:truck, model:CAT, color: blue*

*// Demonstrate 'vehicle' is still unaltered*

**var** secondInstance = **new** vehicle('car');

console.log(secondInstance);

*// as before, vehicle: car, model:default, license: 00000-000*

1. **The Factory Pattern:**

**function** VehicleFactory() {}

VehicleFactory.prototype.vehicleClass = Car;

VehicleFactory.prototype.getVehicle = **function** (options) {

**return new this**.vehicleClass(options);

};

**var** carFactory = **new** VehicleFactory();

**var** car = carFactory.getVehicle({ color: "yellow", turbo: **true** });

console.log(car **instanceof** Car); *// => true*

*// approach #1: Modify a VehicleFactory instance to use the Truck class*

carFactory.vehicleClass = Truck;

**var** mover = carFactory.getVehicle({ enclosedCargo: **true**, length: 26 });

counsole.log(mover **instanceof** Truck); *// => true*

*// approach #2: Subclass VehicleFactory to create a factory class that*

*// builds Trucks*

**function** TruckFactory () {}

TruckFactory.prototype = **new** VehicleFactory();

TruckFactory.prototype.vehicleClass = Truck;

**var** truckFactory = **new** TruckFactory();

**var** bigfoot = truckFactory.getVehicle({ monster: **true**, cylinders: 12 });

console.log(bigfoot **instanceof** Truck); *// => true*

1. **Singleton Pattern pseudo code:**

class MySingleton

{

public:

static MySingleton \* GetInstance()

//static method that returns only instance of MySingletone

{

if (m\_pOnlyOneInstance == NULL) // if not yet instantiated

{

m\_pOnlyOneInstance = new MySingleton();

//create one and only object

}

return m\_pOnlyOneInstance;

}

private:

static MySingleton \* m\_pOnlyOneInstance;

//holds one and only object of MySingleton

MySingleton(); // private constructor

public:

// MySingleton functionalities

void foo();

bool goo();

int zoo();

};

1. **jQuery** is a multi-browser JS library designed to simplify the client-side scripting of HTML. I would use the jQuery for animating my website to style it up and to make it really nice for the user to browser. What I wouldn’t use is probably remove element since there is nothing that I want or remove or destroy my website.
2. **Backbone.js** is a JavaScript library with a RESTful JSON interface, and is based on the model-view-presenter (MVP) application design paradigm.

The difference between the backbone.js and jQuery is that jQuery is a multi-browser JS library that is designed to simplify the client-side scripting of HTML. Backbone.js is a small library which is design for developing single-page applications, and for keeping various parts of web applications synchronized. jQuery library is bigger and its syntax is to designed to make it easier to navigate a document, select DOM elements, create animations, handle events, and develop Ajax applications.

1. **getElementByTagName:**

<!DOCTYPE html>

<html>

<head>

<script src="loadxmldoc.js"></script>

</head>

<body>

<script>

xmlDoc=loadXMLDoc("books.xml");

x=xmlDoc.getElementsByTagName("title")[0]

y=x.childNodes[0];

document.write(y.nodeValue);

</script>

</body>

</html>

1. **getElementById:**

**var** myVariable = document.getElementById("my\_element");

1. **innerHTML:**

var myContentHolder = document.getElementById("content");

myContentHolder.innerHTML = "<p>This is the dynamic content created by the innerHTML property</p>";

1. **createElement:**

**var myNewListItem = document.createElement("li");**

**var myNewLink = document.createElement("a");**

1. Node.js is a development platform built on top of Google’s V8 JavaScript virtual machine. While JavaScript engines (including V8) are traditionally run in Web browsers to form the client side of a client/server application, the Node.js libraries are focused on building server-side applications in JavaScript.

Node.js is intended to run on a dedicated HTTP server and to employ a single thread with one process at a time. Node.js applications are event-based and run asynchronously. Code built on the Node platform does not follow the traditional model of receive, process, send, wait, receive. Instead, Node processes incoming requests in a constant event stack and sends small requests one after the other without waiting for responses.

1. **Unit Tests:** testing an individual unit, such as a method (function) in a class, with all dependencies mocked up.

**Functional Test**: Aka Integration Test, testing a slice of functionality in a system. This will test many methods and may interact with dependencies like Databases or Web Services.

1. Design Pattern is a general reusable solution to a commonly occurring problem within a given context in software design. It is not a finished design that can be transformed directly into source or machine code, but it is a description or a template for how to solve a problem that can be used in many different situations.

I would use the factory pattern to because it helps to deal with problems and create an object that will be created. It also helps to let the classes to be implementing the interface to decide which class to instantiate. The factory pattern also lets a class to defer instantiation to subclasses.

1. I would explain that unit testing is a software development process in which the smallest testable parts of an application, called units, are individually and independently scrutinized for proper operation. Unit testing is often automated but it can also be done manually. This testing mode is a component of Extreme Programming (XP), a pragmatic method of software development that takes a meticulous approach to building a product by means of continual testing and revision.

Unit Testing is important because:

* Instant Gratification: Unit testing allows you to execute your code right away and know that it works. This is a very powerful tool. One of the reasons I enjoy programming is seeing the results of what I built. Unit testing allows you to instantly try out your code and see it working.
* Code Against Your API While or Before it is Built: Unit testing gives you the chance to code against your API/class while you build it, or in the case of TDD, before you build it. Your tests then become a template for future classes that use the API. They will interact with it the same way that a client class will.
* Leads to a Better Design: Since unit testing should start very early in the process of writing code, it can help you to design an API. Since the tests will become templates as to how to use the class under test, you can get a feel for how natural the API feels. This ultimately leads to a better design and helps you to choose good names for methods and also to determine which methods are really needed.
* Understand How Your Code Works: Unit tests really work out your code and tell you how your code will act under many different scenarios. Whether the tests pass or fail, you will learn something about how your code executes.
* Confidence in Your Code: You may have heard the phrase “Refactor without fear” and unit testing gives you this capability. This means that you can make changes to your code and instantly know whether those changes have broken it. When you have a good baseline of tests, you know your code works and you don’t have to be afraid of changes to it in the future.

<https://github.com/cjy2/IS-217-Final-Exam.git>