MedTech – Mediterranean Institute of Technology

CS-Web and Mobile Development

Chp4- JavaScript Design Patterns

Design Patterns

MedTech

What is a Pattern?

JavaScript Design Patterns

- A pattern is a reusable solution that can be applied to commonly occurring problems in software design
  - In our case, in writing JavaScript web applications.
- Sorts of templates for how we solve problems - ones which can be used in quite a few different situations.
- Three main benefits
  - Patterns are proven solutions: They provide solid approaches to solving issues in software development using proven techniques that reflect the experience and insights the developers that helped define them bring to the pattern.
  - Patterns can be easily reused: A pattern usually reflects an out of the box solution that can be adapted to suit our own needs. This feature makes them quite robust.
  - Patterns can be expressive: When we look at a pattern there's generally a set structure and vocabulary to the solution presented that can help express rather large solutions quite elegantly.

Summary of Design Patterns

JavaScript Design Patterns

Creational: Based on the concept of creating an object

Class	
Factory Method	This makes an instance of several derived classes based on interfaced data or events.
Object	
Abstract Factory	Creates an instance of several families of classes without detailing concrete classes.
Builder	Separates object construction from its representation, always creates the same type of object.
Prototype	A fully initialized instance used for copying or cloning.
Singleton	A class with only a single instance with global access points.

Summary of Design Patterns

JavaScript Design Patterns

Structural: Based on the idea of building blocks of objects.

Class	
Adapter	Match interfaces of different classes therefore classes can work together despite incompatible interfaces. (uses inheritance)
Object	
Adapter	Match interfaces of different classes therefore classes can work together despite incompatible interfaces. (uses composition)
Bridge	Separates an object's interface from its implementation so the two can vary independently.
Composite	A structure of simple and composite objects which makes the total object more than just the sum of its parts.
Decorator	Dynamically add alternate processing to objects.
Façade	A single class that hides the complexity of an entire subsystem.
Flyweight	A fine-grained instance used for efficient sharing of information that is contained elsewhere.
Proxy	A place holder object representing the true object.

Summary of Design Patterns

JavaScript Design Patterns

Behavioral: Based on the way objects play and work together

Class	
Interpreter	A way to include language elements in an application to match the grammar of the intended language.
Template Method	Creates the shell of an algorithm in a method, then defer the exact steps to a subclass.
Object	
Chain of Responsibility	A way of passing a request between a chain of objects to find the object that can handle the request.
Command	Encapsulate a command request as an object to enable, logging and/or queuing of requests, and provides error-handling for unhandled requests.
Iterator	Sequentially access the elements of a collection without knowing the inner workings of the collection.
Mediator	Defines simplified communication between classes to prevent a group of classes from referring explicitly to each other.
Memento	Capture an object's internal state to be able to restore it later.
Observer	A way of notifying change to a number of classes to ensure consistency between the classes.
State	Alter an object's behavior when its state changes.
Strategy	Encapsulates an algorithm inside a class separating the selection from the implementation.
Visitor	Adds a new operation to a class without changing the class.

JavaScript Design Patterns

JS PATTERNS

Creational Pattern

JavaScript Design Patterns

- Deals with the creation of new objects
- Basically three ways to create objects:
  - `var newObject = {};`
  - `var newObject = Object.create(null);`
  - `var newObject = new Object();`
- And four ways to assign a value to an object
  1. Dot Syntax
     - `newObject.someKey = 'Hello World';  // Write properties`
     - `var key = newObject.someKey;        // Access properties`
  2. Square Brackets Syntax
     - `newObject['someKey'] = 'Hello World';// Write properties`
     - `var key = newObject['someKey'];          // Access properties`

Creational Pattern

JavaScript Design Patterns

3. definePropery

```
Object.defineProperty(newObject, "someKey", {
     value: "for more control of the property's behavior",
     writable: true,
     enumerable: true,
     configurable: true
});
```

5. defineProperties

```
Object.defineProperties(newObject, {
     "someKey": {
          value: "Hello World",
          writable: true
     },
     "anotherKey": {
          value: "Foo bar",
          writable: false
     }
});
```

Constructor Pattern

JavaScript Design Patterns

- Basic Constructors

```
function Car(model, year, miles) {
     this.model = model;
     this.year = year;
     this.miles = miles;
     this.toString = function () {
          return this.model + " has done " +
               this.miles + " miles";
     };
}

var civic = new Car("Honda Civic", 2009, 20000);
var mondeo = new Car("Ford Mondeo", 2010, 5000);

console.log(civic.toString());
console.log(mondeo.toString());
```

Constructor Pattern

JavaScript Design Patterns

- Constructors with Prototype

```
function Car(model, year, miles) {
    this.model = model;
    this.year = year;
    this.miles = miles;
}

Car.prototype.toString = function () {
    return this.model + " has done " +
        this.miles + " miles";
};

var civic = new Car("Honda Civic", 2009, 20000);
var mondeo = new Car("Ford Mondeo", 2010, 5000);

console.log(civic.toString());
console.log(mondeo.toString());
```

Module Pattern

JavaScript Design Patterns

- Module
  - Interchangeable single-part of a larger system that can be easily reused
- Using the notion of IIFE: Immediately Invoked Function Expressions

```
(function() {
    // code to be immediately invoked
}());
```

- Problem: there is no real privacy in JavaScript
- The typical module pattern is where immediately invoked function expressions (IIFEs) use execution context to create 'privacy'
  - Here, objects are returned instead of functions.
- In the pattern
  - Variables declared are only available inside the module.
  - Variables defined within the returning object are available to everyone

Module Pattern

JavaScript Design Patterns

```
var basketModule = (function() {
    var basket = [];            //private
    return {                    //exposed to public
        addItem: function(values) {
            basket.push(values);
        },
        getItemCount: function() {
            return basket.length;
        },
        getTotal: function(){
            var q = this.getItemCount(), p=0;
            while(q--){
                p+= basket[q].price;
            }
            return p;
        }
    }
}());
```

Module Pattern

JavaScript Design Patterns

```
//basketModule is an object with properties which can
also be methods

basketModule.addItem({item:'bread',price:0.5});
basketModule.addItem({item:'butter',price:0.3});
console.log(basketModule.getItemCount());
console.log(basketModule.getTotal());

//however, the following will not work:
// (undefined as not inside the returned object)

console.log(basketModule.basket);      //error!

//(only possible within the module scope)

console.log(basket);      //error
```

Façade Pattern

JavaScript Design Patterns

- Structural Pattern
- Convenient, high-level interfaces to larger bodies of code that hide underlying complexity
    - Aims to simplify the presented API to other developers
- The facade pattern :
    - Simplifies the interface of a class
    - Decouples the class from the code that uses it
- Facades can be used with the Module pattern in order to hide its methods
    - It differs from the Module pattern as the limited public API differs greatly from the reality of the implementation.

Façade Pattern

JavaScript Design Patterns

```
var module = (function() {
   var _private = {
      i:5,
      get : function() {
         console.log('current value:' + this.i);
      },
      set : function( val ) {
         this.i = val;
      },
      run : function() {
         console.log('running');
      },
      jump: function(){
         console.log('jumping');
      }
   };
   return {
      facade : function( args ) {
         _private.set(args.val);
         _private.get();
         if ( args.run ) {
            _private.run();
         }
      }
   }
}());
module.facade({run: true, val:10});
//outputs current value: 10, running
```

Mediator Pattern

JavaScript Design Patterns

- A mediator:
    - A neutral party who assists in negotiations and conflict resolution.
- Behavioural design pattern
- Encapsulates how disparate modules interact with each other by acting as an intermediary
    - If a system seems to have too many direct relationships between its modules (colleagues), it may be time to have a central point of control that modules communicate through instead.
- A mediator:
    - Promotes loose coupling
    - Modules can broadcast or listen for notifications without worrying about the rest of the system

Mediator Pattern
JavaScript Design Patterns

Mediator Pattern

JavaScript Design Patterns

```
var Participant = function(name) {
    this.name = name;
    this.chatroom = null;
};

Participant.prototype = {
    send: function(message, to) {
        this.chatroom.send(message, this, to);
    },
    receive: function(message, from) {
        log.add(from.name + " to " + this.name + ": " + message);
    }
};

var Chatroom = function() {
    var participants = {};
    return {
        register: function(participant) {
            participants[participant.name] = participant;
            participant.chatroom = this;
        },
        send: function(message, from, to) {
            if (to) {                      // single message
                to.receive(message, from);
            } else {                       // broadcast message
                for (key in participants) {
                    if (participants[key] !== from) {
                        participants[key].receive(message, from);
                    }
                }
            }
        }
    };
};
```

Mediator Pattern

JavaScript Design Patterns

```
// log helper

var log = (function() {
    var log = "";

    return {
        add: function(msg) { log += msg + "\n"; },
        show: function() { alert(log); log = ""; }
    }
})();

function run() {
    var yoko = new Participant("Yoko");
    var john = new Participant("John");
    var paul = new Participant("Paul");
    var ringo = new Participant("Ringo");

    var chatroom = new Chatroom();
    chatroom.register(yoko);
    chatroom.register(john);
    chatroom.register(paul);
    chatroom.register(ringo);

    yoko.send("All you need is love.");
    yoko.send("I love you John.");
    john.send("Hey, no need to broadcast", yoko);
    paul.send("Ha, I heard that!");
    ringo.send("Paul, what do you think?", paul);

    log.show();
}
```

PUTTING IT ALL TOGETHER

Building Large and Complex Applications

JavaScript Design Patterns

- How do Design Patterns help in building a large and complex application?
- Large-scale JavaScript apps are **non-trivial** applications requiring **significant** developer effort to maintain, where most heavy lifting of data manipulation and display falls to the browser
  - You should dedicate sufficient time to planning the underlying architecture
- Ask yourself these questions about your application:
  - How much of your modules are instantly re-usable?
  - Can single modules exist on their own independently?
  - Can single modules be tested independently?
  - How much are your modules coupled?
  - If specific parts of your application fail, will it still function?

Building Large and Complex Applications

JavaScript Design Patterns

« The secret to building large apps is to **never** build large apps. Break your applications into small pieces. Then, assemble those testable, bite-sized pieces into your big application » - Justin Meyer

« The **more tied** components are to each other, the **less reusable** they will be, and the more **difficult** it becomes to make changes to one without accidentally affecting another" - Rebecca Murphey

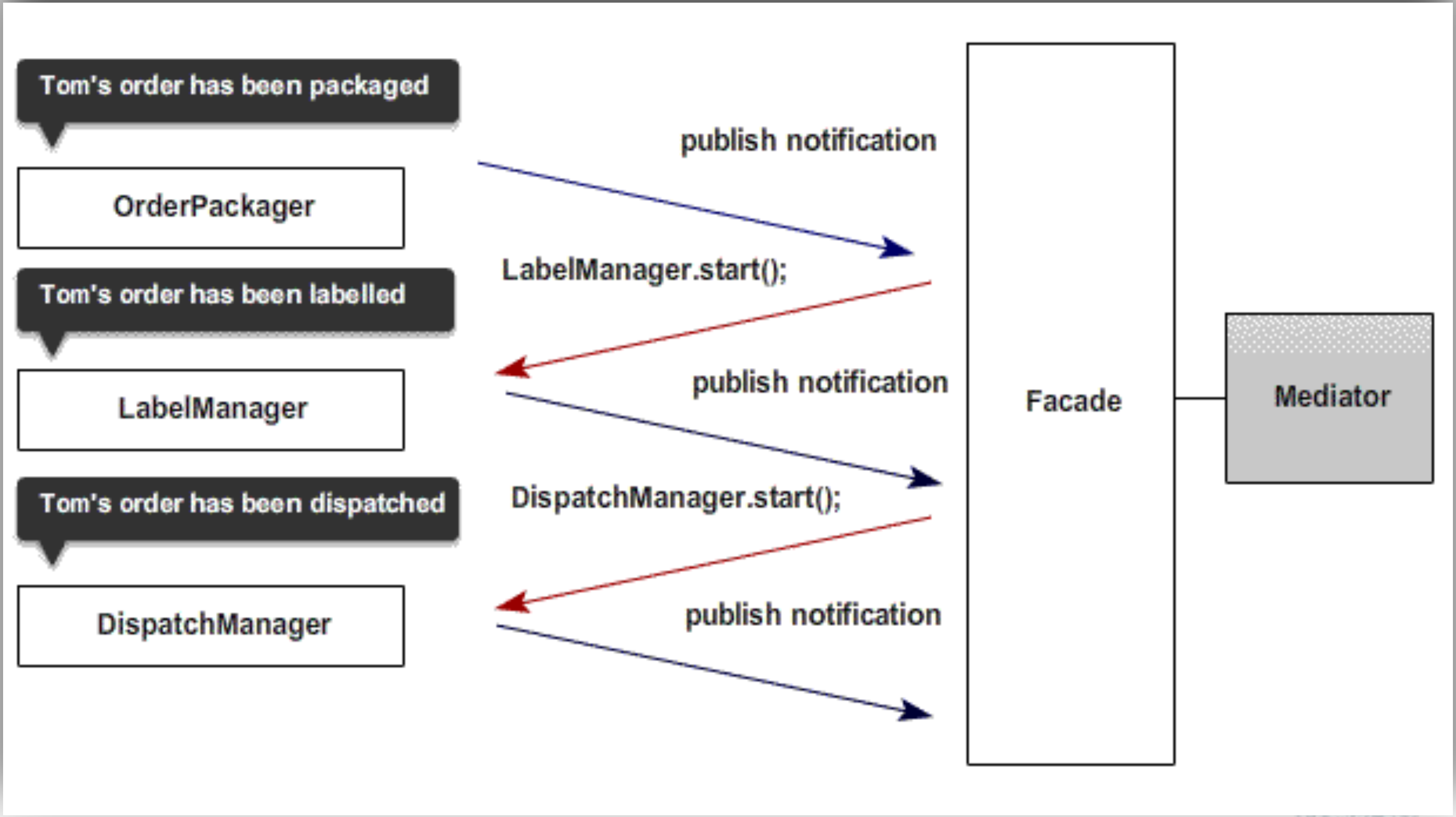
Building Large and Complex Applications

JavaScript Design Patterns

- Our objectives
  - Loosely coupled architecture
  - Functionality broken down into smaller independent modules
  - Framework and library agnostic, flexibility to change in the future
- But also:
  - Single modules speak to the app when something interesting happens
  - An intermediate layer interprets the requests
  - Modules can't access the core directly
  - The app shouldn't fall over due to an error in a specific module

Building Large and Complex Applications

JavaScript Design Patterns



Building Large and Complex Applications

JavaScript Design Patterns

- Facade
  - Abstraction of the core that sits in the middle between it and the modules
  - Ensures a consistent interface to our modules is available at all times
  - Should be the only thing modules are aware of
    - They shouldn't know about each other, or about the core application
  - Acts as a security guard, determining which part of the application a module can access
- Application core : Manages the modules lifecycle:
  - Starts and stops them, restarts them if they fail
  - Adds and removes them without breaking anything
  - Detects and manages errors of the system
- Modules
  - Inform the application when something interesting happens to publish events of interest
  - Don't care about when, where and to whom the notifications are issued
  - Aren't concerned if other modules fail

References

- M. Haverbeke, Eloquent JavaScript: A modern introduction to programming, 2014

- Textbook
  - A. Osmany, Learning JavaScript Design Patterns, O'Reilly, 2012