COEN 168/268

Mobile Web Application Development

JavaScript Design Patterns

Peter Bergström (pbergstrom@scu.edu)

Santa Clara University

Many of you probably are familiar with design patterns already

Some of your might have read *Design Patterns: Elements of Reusable Object-Oriented Software* by the 'Gang of Four'

While this class is particularly about mobile web applications…

The design patterns used in other languages still apply

This Lecture Only Covers the Basics

- You can spend the whole course on design patterns and this lecture barely scratches the surface
- This lecture covers patterns that I think you should know
- The goal is to give you some inspiration on which patterns to explore to make your app better
- For more details look at *Learning JavaScript Design Patterns* by Addy Osmani

As most of you know...

one of the most important things about writing maintainable code
is to take advantage of common patterns

What Is A Design Pattern?

- A reusable solution to a commonly occurring problem
- Can be view as templates
- Provides proven a solution
- Should be be easy to reuse

Why Use Design Patterns?

- By using a design pattern you can prevent minor problems cause major architectural issues
- Design patterns can provide general solutions that are well documented
- Certain design patterns can reduce the code that you write by avoiding repetition
- Certain design patterns should be in all developers vocabulary and therefore everyone knows what is going on

Design Patterns Are The Same Across Languages

- A design pattern applied to JavaScript should work in Java, C, Objective-C, Swift, Python, and etc
- Design patterns are timeless. A design pattern from 30 years ago is probably still applicable today
- This is because they are patterns that are meant to solve problems are common across many different types of applications

Before Talking About JS Design Patterns, Let's Talk About Anti-Patterns

- Polluting the global namespace buy defining global variables
- Passing strings rather than functions to `setTimeout` or `setInterval` that use the `eval()` method internally.
- Modifying the `Object` prototype instead of your own object
- Using `document.write` instead of using native DOM manipulation methods

Categories of Design Patterns

- **Creational Design Patterns**
  - Such as `Constructor`, `Module`, `Factory`, `Abstract`, `Prototype`, `Singleton`, `Builder`, and more...
- **Structural Design Patterns**
  - Such as `Decorator`, `Facade`, `Flyweight`, `Adapter`, `Proxy`, and more...

Categories of Design Patterns, Continued

- **Behavioral Design Patterns**
  - Such as `Iterator`, `Mediator`, `Observer`, `Visitor`, and more…
- **Architectural Design Patterns**
  - Such as `Model-View-Controller`, `Model-View-Presenter`, `Model-View-ViewModel`, and more…

Let’s Go Through These Patterns

Hopefully some of these patterns will be useful when developing your applications

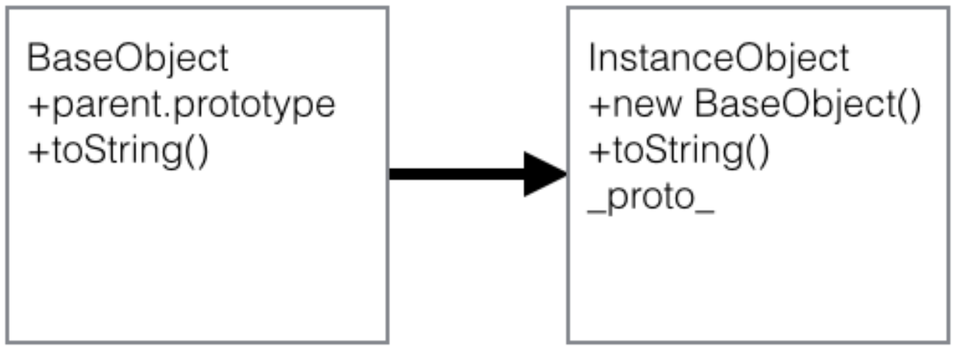
(Most of the graphics are from Wikipedia)

First up:

`Creational` Design Patterns

The `Constructor` Design Pattern

- In object oriented languages, constructors are special methods that are used to initialize a newly allocated instance of a object
- JavaScript is no different (although not class-based)



Without the `Constructor` Design Pattern

You can create an `Object` like this as defined in the JavaScript lecture:

```
var mustang = {};

mustang.make = "Ford";
mustang.model = "Mustang";
mustang.year = 2015;
```

But, it's hard to maintain and extend

The Constructor Pattern In JavaScript

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

  this.description = function() {
    return this.year + " " + this.make + " " + this.model;
  };

  return this;
}

var mustang = new Car('Ford', 'Mustang', 2015);
var camaro  = new Car('Chevrolet', 'Camaro', 2015);
```

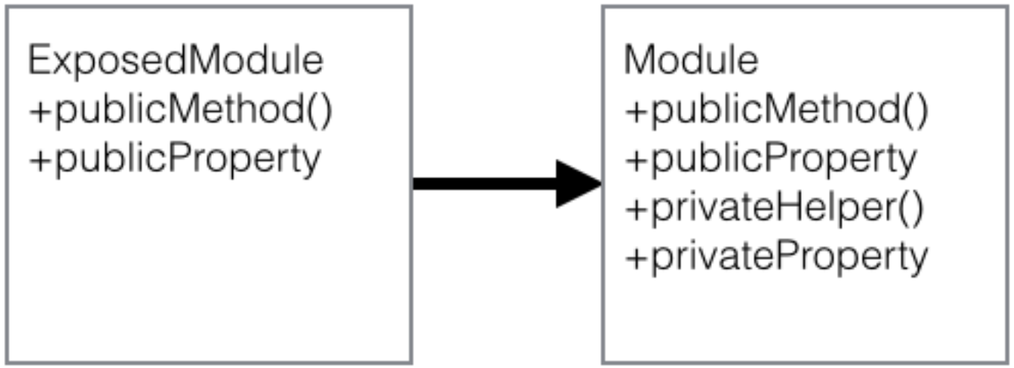
The Constructor Pattern In JavaScript

You could also extend the prototype in conjunction with the Constructor design pattern to make break things out further:

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

  return this;
}
Car.prototype.description = function() {
  return this.year + " " + this.make + " " + this.model;
};
```

The `Module` Design Pattern

- JavaScript doesn't have **private** functions and properties
- But, using the `Module` design pattern, one can hide things with using closures

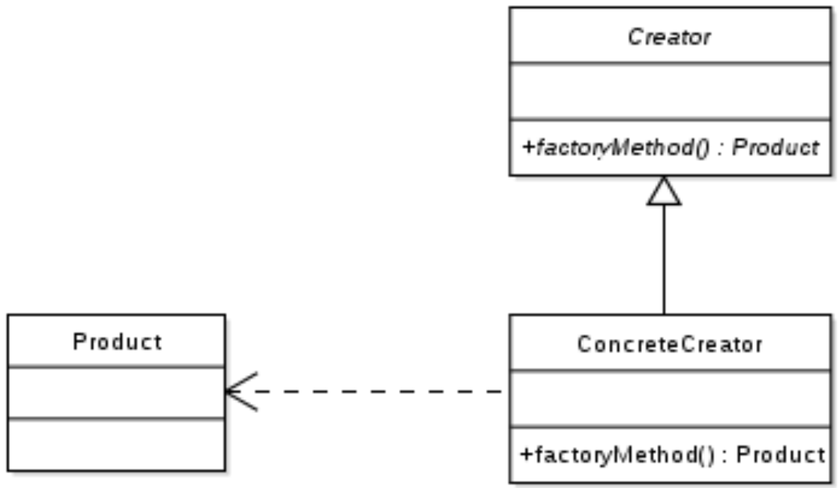


The Module Pattern In JavaScript

```
var dateIncrementor = (function() {
  var date = new Date();
  return {
    incrementDate: function() {
      date.setDate(date.getDate()+1);
      return date;
    },
    decrementDate: function() {
      date.setDate(date.getDate()-1);
      return date;
    }
  };
})();

dateIncrementor.date; // err!
dateIncrementor.incrementDate(); // increment
```

The `Factory` Design Pattern

- Provides a generic interface for creating objects
- Allows you to create objects without having to specify the type
- Use with objects with similar props that are complex to setup



The Factory Pattern In JavaScript

```
function USD() {
  this.name  = 'US Dollar';
  this.valueComparedToUSD = 1;
  return this;
}
function CHF() {
  this.name  = 'Swiss Franc';
  this.valueComparedToUSD = 1.12;
  return this;
}
function CurrencyFactory() {};
CurrencyFactory.prototype.createCurrencyFromCountryCode = function(country) {
  if(country === 'US') {
    return new USD();
  }
  if(country === 'CH') {
    return new CHF();
  }
};
```

The Factory Pattern In JavaScript, Continued

```
var currencyFactory = new CurrencyFactory();

var currency = currencyFactory.createCurrencyFromCountryCode('US');

console.log(currency.name); // US Dollar
```

The `Singleton` Design Pattern

- Restricts the instantiation of a class to one instance of an object
- Useful if you need to create only one object that is shared across the whole application or system
- Can be advantageous if the object creation and state is expensive to set up
- Some people think that it is overused (they are right, but I still use them because they are needed)

The Singleton Pattern In JavaScript

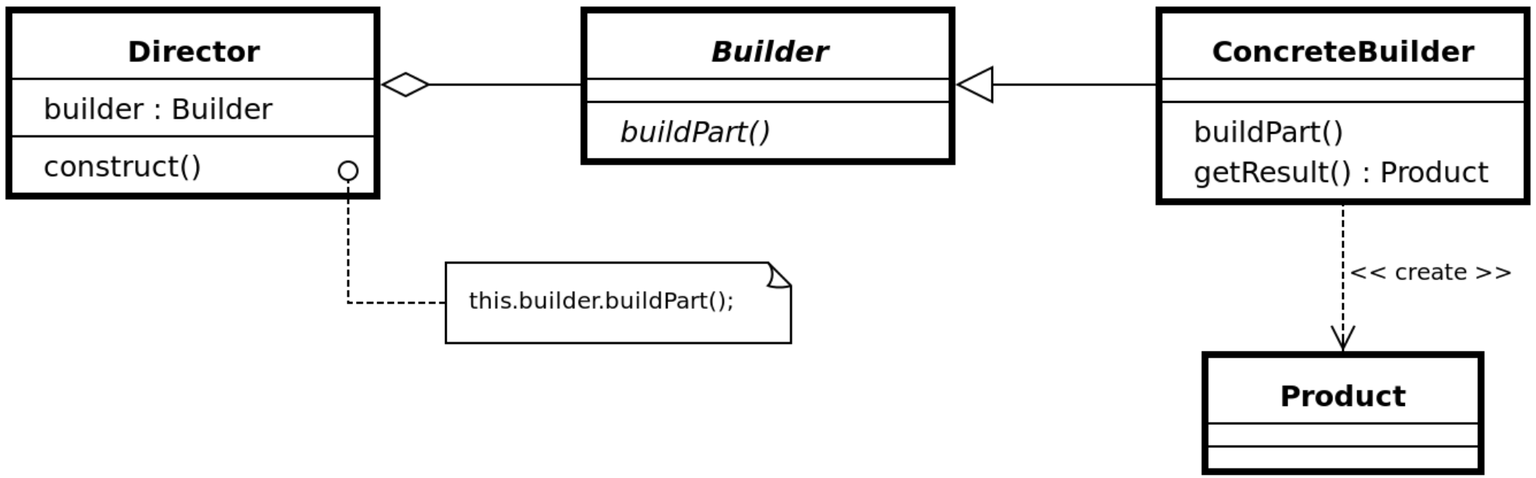
```
var InMemoryDataSource = (function() {
  var instance;
  function init() {
    var myData = [];
    return {
      add: function(obj) {
        myData.push(obj);
      },
      remove: function(obj) {
        myData.splice(myData.indexOf(obj), 1);
      },
      objectAtIndex: function(index) {
        return myData[index];
      }
    };
  }
  return {
    sharedInstance: function() {
      if(!instance) {
        instance = init();
      }
      return instance;
    }
  };
})();
```

The Singleton Pattern In JavaScript, Cont'd

```
InMemoryDataSource.sharedInstance().add({title: 'first'});

var firstObject = InMemoryDataSource.sharedInstance().objectAtIndex(0);

console.log(firstObject.title); // first
```

The `Builder` Design Pattern

- You add properties to the builder object
- Then ask the builder to build a specific object with specific properties



The Builder Design In JavaScript

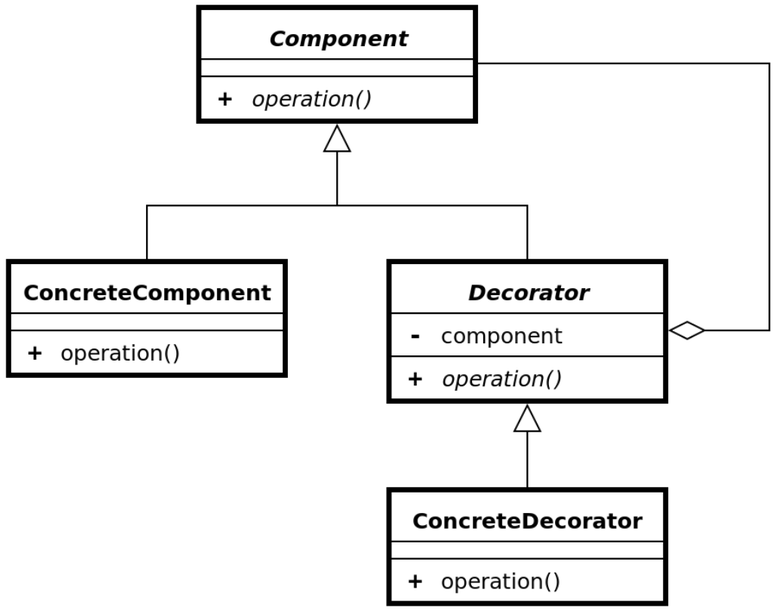
```
var ButtonBuilder = function() {
  this.setColor = function(color) {
    this.color = color;
  };
  this.setText =  function(text) {
    this.text = text;
  };
  this.build = function() {
    return $('<button/>').css({color:this.color}).html(this.text);
  }
  return this;
};

var buttonBuilder = new ButtonBuilder();
buttonBuilder.setColor('black');
buttonBuilder.setText('Build!');
$('body').append(buttonBuilder.build());
// appends <button style="color:black;">Build!</button> to body
```

Let's Shift Gears:

`Structural` Design Patterns

The `Decorator` Design Pattern

- Allows behavior to be added to an individual object
- Does not alter the behavior of other objects of a similar type



The Decorator Pattern in JavaScript

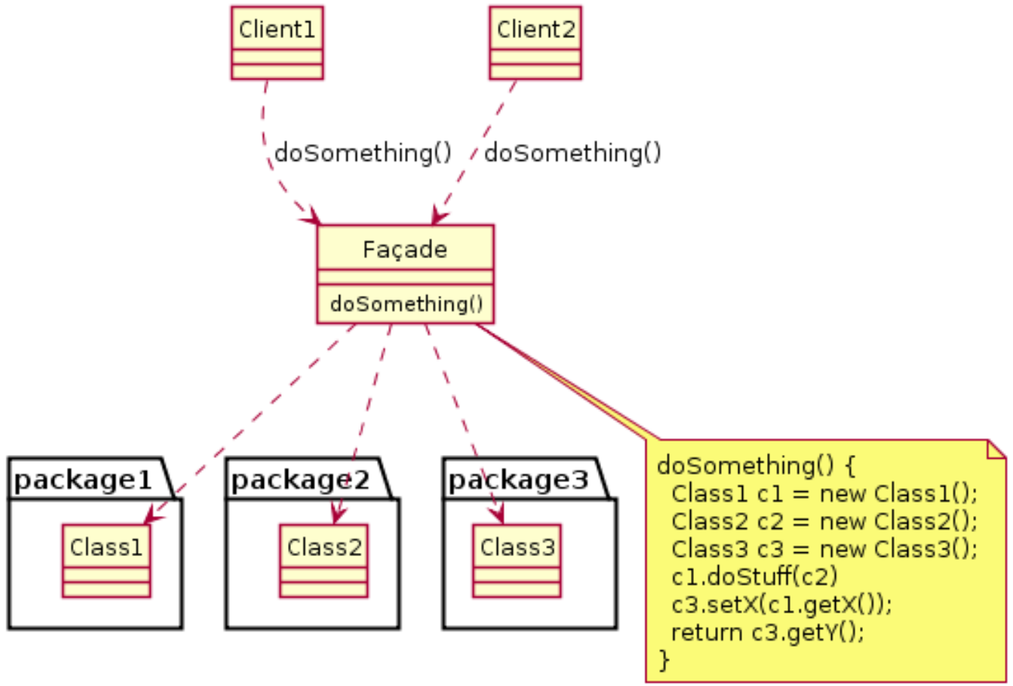
```
function Coffee() {
  this.cost = function() {
    return 1.50;
  }
};
function WithMilk(coffee) {
  var origCost = coffee.cost();
  coffee.cost = function() {
    return 1.00 + origCost;
  }
};
function WithSugar(coffee) {
  var origCost = coffee.cost();
  coffee.cost = function() {
    return 0.15 + origCost;
  }
};
```

The Decorator Pattern in JavaScript, Cont'd

```
var fancyCoffee = new Coffee();
WithMilk(fancyCoffee);
WithSugar(fancyCoffee);
console.log(fancyCoffee.cost) // 2.65

var plainCoffee = new Coffee();
console.log(plainCoffee.cost) // 1.50
```

The Facade Design Pattern

- Use this when you want to expose a simple interface to a more complex underlying implementation



The Facade Pattern in JavaScript

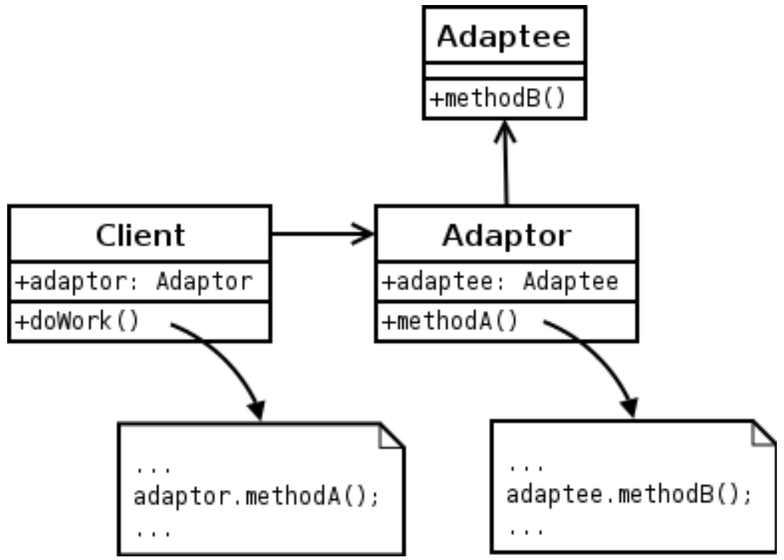
jQuery's `.hasClass()` helper:

```
hasClass: function( selector ) {
    var className = " " + selector + " ",
      i = 0,
      l = this.length;
    for ( ; i < l; i++ ) {
      if ( this[i].nodeType === 1 &&
          (" " + this[i].className + " ").replace(rclass, " ").indexOf( className ) >= 0 ) {
        return true;
      }
    }

    return false;
  }
```

jQuery is full of facades making your life easier, so do take the time to appreciate them :)

The `Adapter` Design Pattern

- Allows the interface of an existing object (or class) to be used from another interface
- Makes existing classes work with others without modifying code



The Adapter Pattern In JavaScript

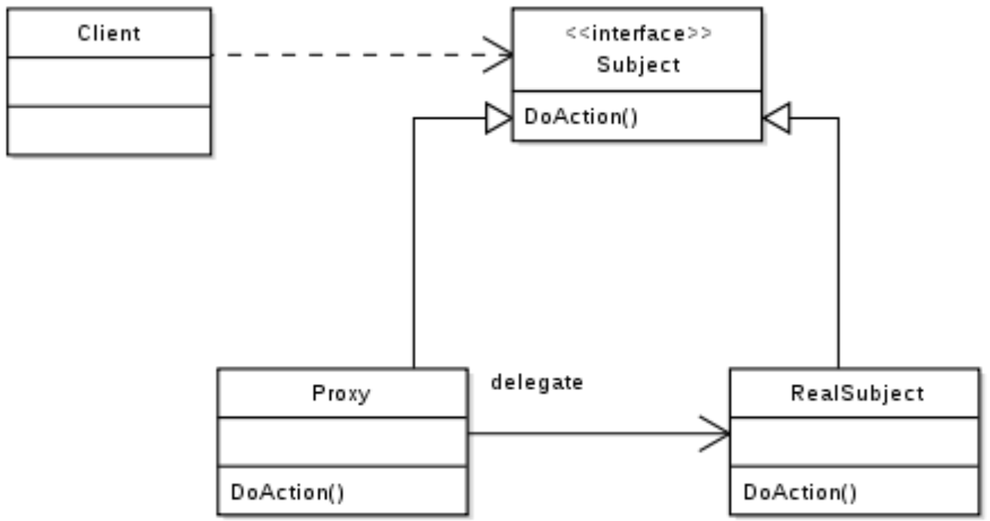
```
var Volt = function(v) {
  var volts = v;

  return {
    getVolts: function() {
      return volts;
    },
    setVolts: function(v) {
      volts = v;
    }
  };
};

var Socket = function() {
  return {
    getVolt: function() {
      return new Volt(120);
    }
  }
};
// inspired by:
// http://www.journaldev.com/1487/adapter-design-pattern-in-java-example-tutorial
```

The Adapter Pattern In JavaScript, Cont'd

```
var SocketAdapter = function() {

  var socket = new Socket();

  socket.get120Volt = function() {
    return this.getVolt();
  };
  socket.get12Volt = function() {
    return this.convertVolt(this.getVolt(), 10);
  };
  socket.get3Volt = function() {
    return this.convertVolt(this.getVolt(), 40);
  };
  socket.convertVolt = function(v,i) {
    return new Volt(v.getVolts()/i);
  }
  return socket;
};

var sa = new SocketAdapter();

console.log(sa.get3Volt().getVolts()); // returns 3
```

The `Proxy` Design Pattern

- A proxy is an object functioning as the interface of another
- Could control when an expensive object is instantiated or provide ways to refer to an object



The `Proxy` Pattern In JavaScript

Solves the issue problem with `var self = that` when doing a `setTimeout`:

```
var MyCounter = {
  counter: 0,

  // Update counter every 50 milliseconds
  updateCounter: function() {
    var self = this;
    setTimeout(function() {
      self.counter++;
      self.updateCounter();
    }, 50);
  }
};
MyCounter.updateCounter();
```

The `Proxy` Pattern In JavaScript, Cont'd

jQuery fixes this with the `$.proxy` which provides a way to bind to a specific context:

```
var MyCounter = {
  counter: 0,

  // Update counter every 50 milliseconds
  updateCounter: function() {
    setTimeout($.proxy(function() {
      this.counter++;
      this.updateCounter();
    }, this), 50);
  }
}

MyCounter.updateCounter();
```

Next Up:

`Behavioral` Design Patterns

The `Iterator` Design Pattern

- Used to traverse a container and access the container's elements
- Decouples algorithms from containers
- `$.each` is an example of a jQuery iterator

```
var designPatternCategories = ['Creational', 'Structural',
                               'Behavioral', 'Architectural'];

// Calls the passed in function for each element in the array
$.each(designPatternCategories, function(index, value) {
  console.log("Part " + (index+1) + ": " + value);
});
```

The `Mediator` Design Pattern

- A `Mediator` is an object that encapsulates how a set of objects (`colleagues`) interact with each other
- Communication between `colleagues` is done through this object only
- Useful for large systems in order to reduce coupling between different components

The Mediator Pattern In JavaScript

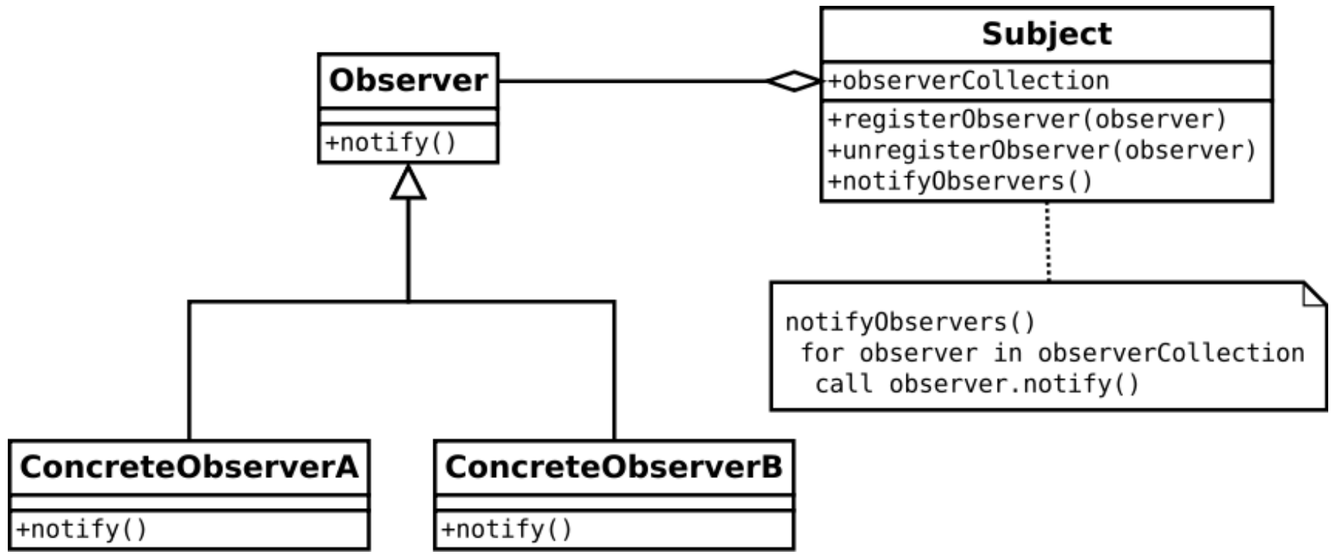
```
var Mediator = function() {
  var colleagues = [];

  this.addColleague = function(colleague) {
    colleagues.push(colleague);
  };
  this.sendMemo = function(message, fromColleague) {
    $.each(colleagues, function(i, colleague) {
      if(fromColleague != colleague) {
        colleague.memo(message);
      }
    });
  };
  return this;
};
```

The Mediator Pattern In JavaScript, Cont'd

```
var Colleague = function(n) {
  this.memo = function(message) {
      console.log(n + " received '" + message + "'");
  }
  return this;
};

var m   = new Mediator();
var c1  = new Colleague('c1');
var c2  = new Colleague('c2');
var c3  = new Colleague('c3');

m.addColleague(c1);
m.addColleague(c2);
m.addColleague(c3);

m.sendMemo("Hello from c2!", c2);

> c1 received 'Hello from c2!'
> c3 received 'Hello from c2!'
```

The `Observer` Design Pattern

- An object, called the `Subject`, maintains a list of dependents, `Observers` that are notified on state changes
- Used in event handling a lot



The Observer Pattern In JavaScript

```
var Subject = function() {
  var observerCollection = [];
  this.registerObserver = function(observer) {
    observerCollection.push(observer);
  };
  this.unregisterObserver = function(observer) {
    observerCollection.splice(observerCollection.indexOf(observer, 1));
  };
  this.notifyObservers = function(message) {
    var context = this;
    $.each(observerCollection, function(i, obs) {
      obs.notify(context, message);
    });
  };
  return this;
};
```

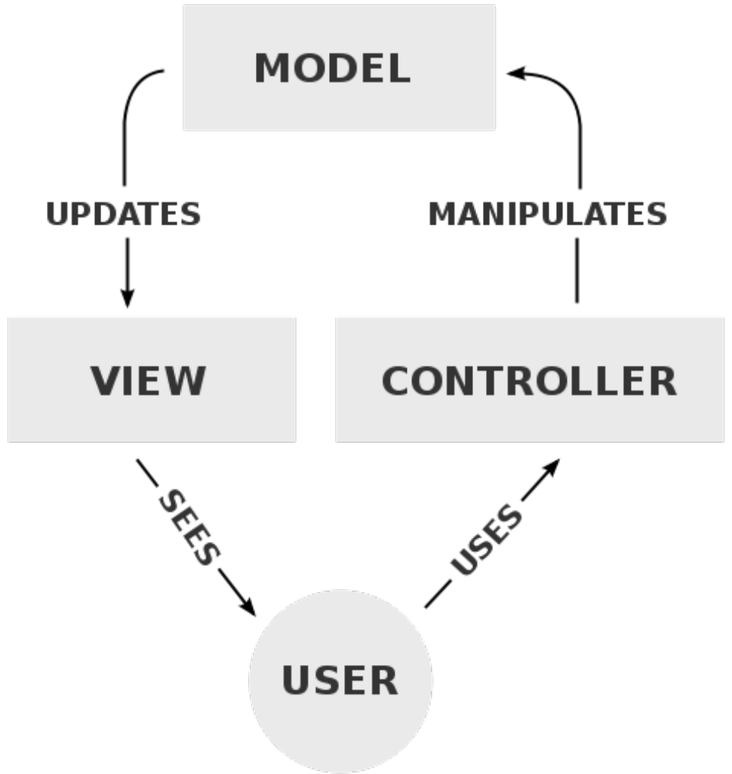
The Observer Pattern In JavaScript, Cont'd

```
var Observer = function() {
  this.notify = function(context, message) {
    console.log(message + " for " + context);
  };
  return this;
};

var s = new Subject();
var obs1 = new Observer();
var obs2 = new Observer();

s.registerObserver(obs1);
s.registerObserver(obs2);
s.notifyObservers('allPropertiesDidChange');

> allPropertiesDidChange for [object Object]
> allPropertiesDidChange for [object Object]
```

Finally:

Architectural Design Patterns

The `Model-View-Controller` (MVC) Design Pattern

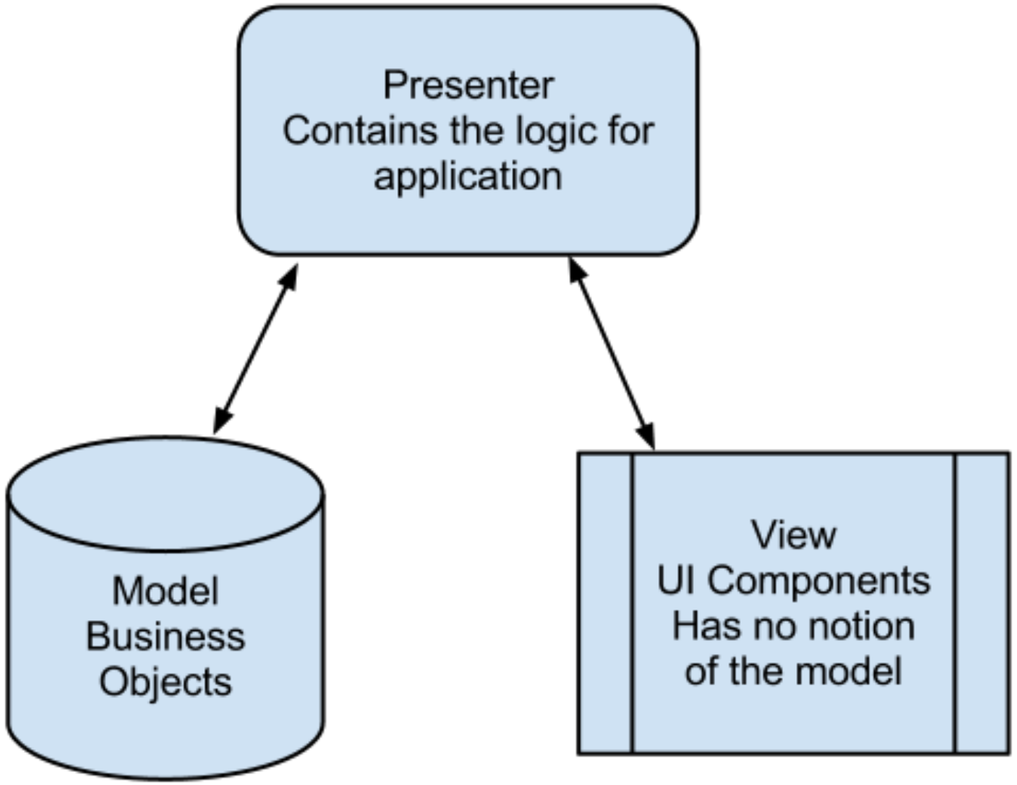


The `Model-View-Controller` (MVC) Design Pattern

- **Controller**: the user *uses* the controller to *change* the model
- **Model**: the controller *manipulates* the model and the model *updates* the view
- **View**: the view *requests* data from model to *generate* the UI displayed to the user

Very important design pattern and is central to most application frameworks. We will talk about **MVC** when it comes to **Ember.js**

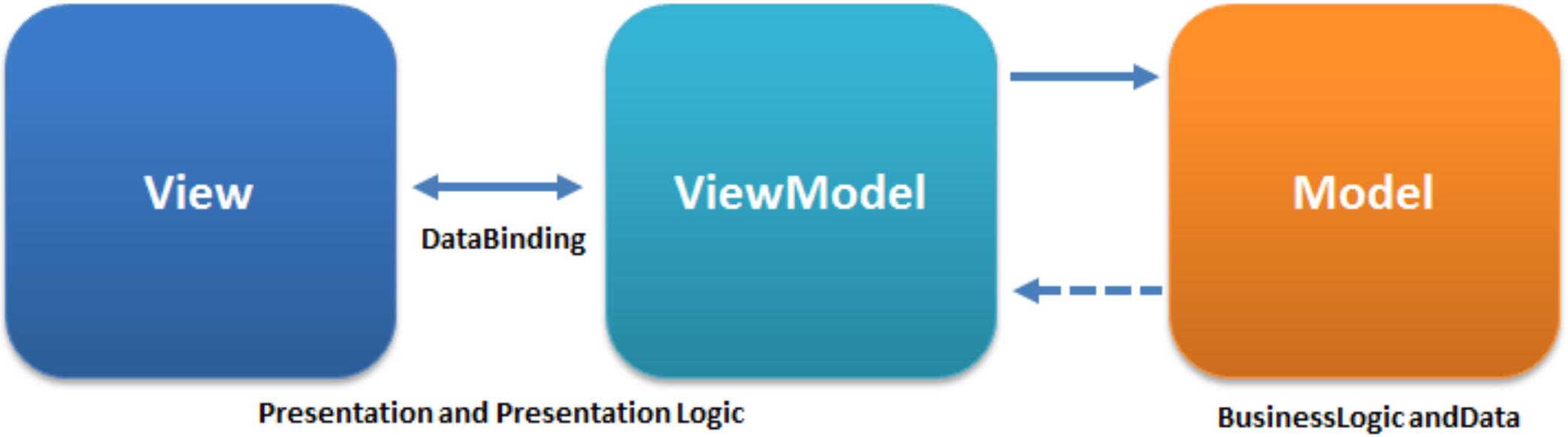
The `Model-View-Presenter` Design Pattern



The `Model-View-Presenter` Design Pattern

- Derivative of MVC pattern
- The `presenter` is the middle man, (instead of the `controller` as in MVC) and all presentation logic is now in the `presenter`
- **Model**: *defines* the data that will be displayed
- **View**: passively *displays* data
- **Presenter**: *act* upon the model and the view

The Model-View-ViewModel (MVVM) Design Pattern



The `Model-View-ViewModel` (MVVM) Design Pattern

- Clearly separates the development of the view and the model
- **Model**: *defines* the data content
- **View**: *displays* data
- **ViewModel**: *mediates* between the model and the view that creates the rules for the view to display the model data without the view needing to know anything about the model itself

MVVM is powerful and makes complex apps easier to manage

We Will Explore the MV* Patterns More As We Get Into Ember.js

Again, for more details look at *Learning JavaScript Design Patterns* by Addy Osmani

The book covers many other design patterns that you can use that I did not cover

COEN 168/268

Mobile Web Application Development

JavaScript Design Patterns

Peter Bergström (pbergstrom@scu.edu)

Santa Clara University