CSWD

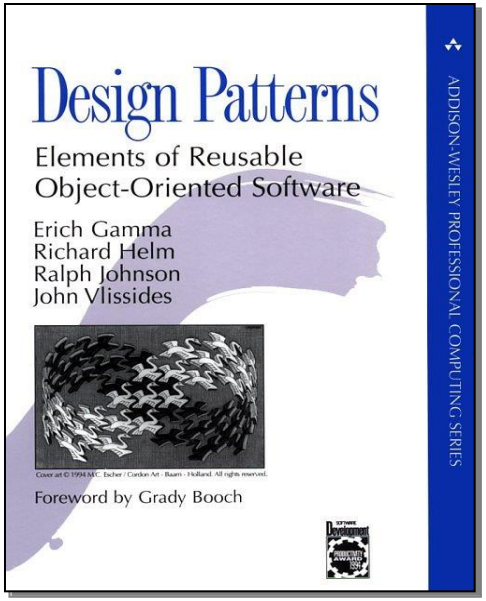
Client Side Web Development

Week 8

Design Patterns … cont.

A pattern in the broader sense of the word is a "theme of recurring events or objects … it can be a template or model which can be used to generate things"

(http://en.wikipedia.org/wiki/Pattern)



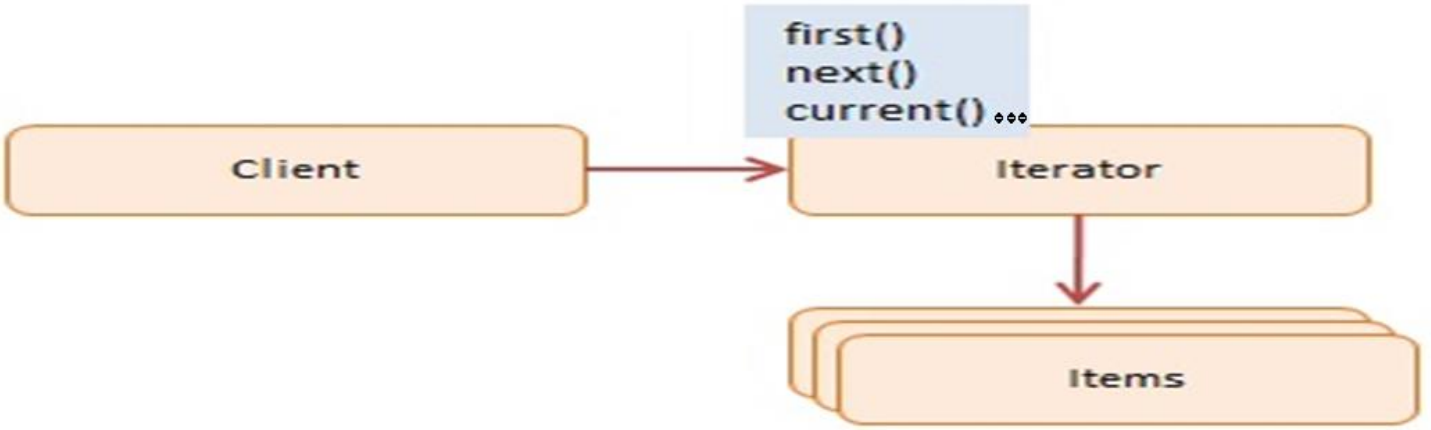
CSWD

Part II: Application cont.

- **Previously looked at:**
  - **Constructor & Factory**
    - » **allows objects to be created in a robust manner**
  - **Module & Revealing Module**
    - » **provides the tools to create self-contained decoupled pieces of code, which can be treated as black boxes of functionality and added, replaced, or removed according to the (ever-changing) requirements of the software you're writing.**
- **Up next:**
  - **Iterator**
    - » **access the elements of an aggregate object sequentially without exposing its underlying representation**
  - **Singleton**
    - » **ensures that not more than one instance of a class/object is created and provides a global point of access to this instance**
  - **Observer**
    - » **One object stores a list of observers that are updated when the state of the object is changed**

Iterator

- **Definition: Provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.**



- **The Iterator object maintains a reference to the collection and the current position.**
- **It also implements the 'standard' Iterator interface with methods like:**
  - **first, next, current, hasNext, reset, and each.**

CSWD

JavaScript Implementation

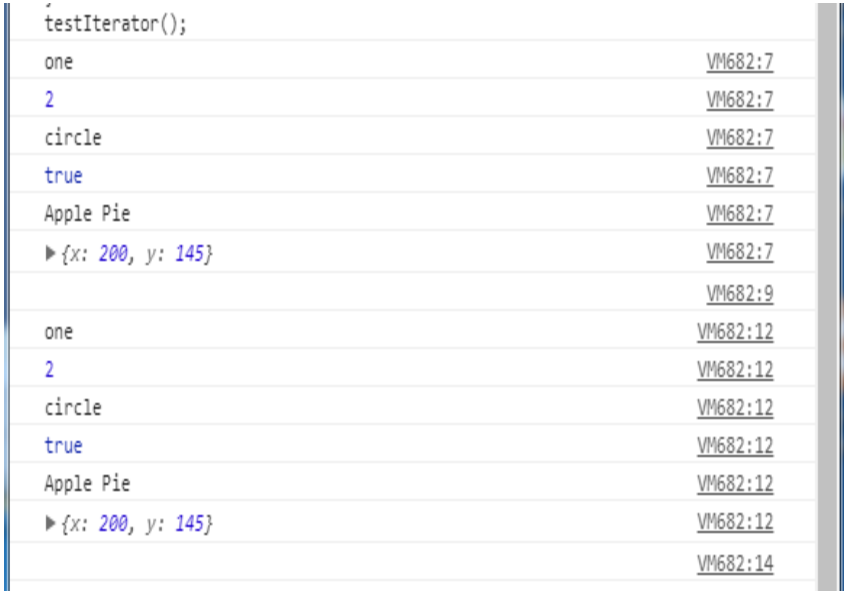
```
function Iterator(items) {
    this.index = -1;
    this.items = items;
}

Iterator.prototype = {
    first: function () {
        this.reset();
        return this.next();
    }, next: function () {
        return this.items[this.index++];
    }, hasNext: function () {
        return this.index <= this.items.length;
    }, reset: function () {
        this.index = 0;
    }, each: function (callback) {
        for (var item = this.first();
             this.hasNext();
             item = this.next()) {
            callback(item);
        }
    }
}
```

Q[2]. Try to implement the *current()* method

CSWD

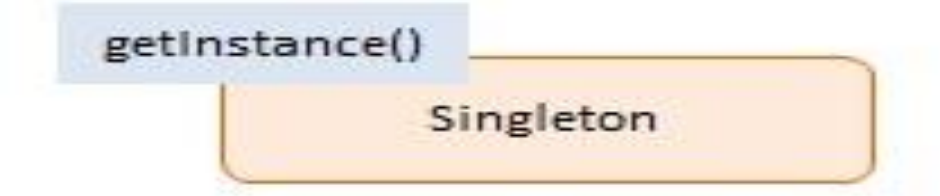
Iterator Test

```
function testIterator() {
    var items = ["one", 2, "circle", true,
        "Apple Pie", {"x": 200, "y": 145}];
    var iter = new Iterator(items);

    // using for loop
    for (var item = iter.first(); iter.hasNext(); item = iter.next()) {
        console.log(item);
    }
    console.log("");

    // using Iterator's each method
    iter.each(function (item) {
        console.log(item);
    });
    console.log("");
}
testIterator();
```



Singleton

- **Definition: Ensures a class has only one instance and provides a global point of access to it.**



- **The Singleton Pattern limits the number of instances of a particular object to just one.**
- **This single instance is called the singleton.**
- **Singletons reduce the need for global variables which is particularly important in JavaScript because it limits namespace pollution and associated risk of name collisions.**
- **The Module pattern is JavaScript's manifestation of the Singleton pattern.**
- **Several other patterns, such as, Factory, Prototype, and Façade are frequently implemented as Singletons when only one instance is needed.**

CSWD

Example code

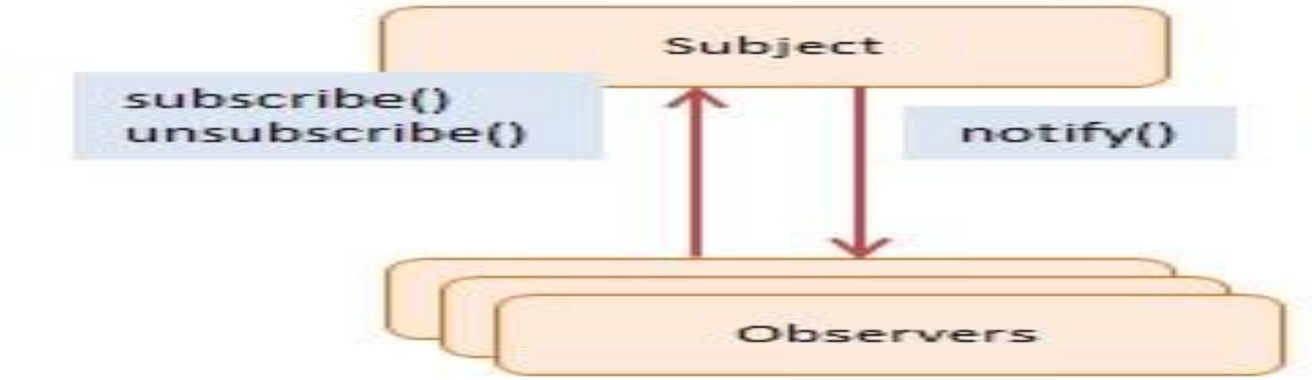
```
var Singleton = ( function () {
    var instance;
    var count = 0;

    function createInstance() {
        count ++;
        var object = {instance:
"instance with count: " + count };
        return object;
    }

    function makeInstance() {
        if (!instance) {
            instance = createInstance();
        }
        return instance;
    }

    return {
        getInstance: makeInstance
    };

} ) ();
```

```
function run() {

    var instance1 =
Singleton.getInstance();
    var instance2 =
Singleton.getInstance();

    alert("Same instance? "
      + (instance1 ===
          instance2));
}

run(); // run the test
```

CSWD

Observer

- **Definition: Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automaticallv.**



- One or more **observers** are interested in the state of a **subject** and **register** their interest with the subject by attaching themselves - **(subscribe)**
- When something changes in our subject that the observer may be interested in, a **notify** message is sent which calls the update method in each observer.
- When the observer is no longer interested in the subject's state, they can simply detach themselves - **(unsubscribe)**

Observer cont.

- The Observer pattern offers a subscription model in which objects subscribe to an event and get notified when the event occurs.
  - this pattern is the cornerstone of event driven programming, including JavaScript.
  - the pattern facilitates good object-oriented design and promotes loose coupling.
- When building web apps you end up writing many event handlers.
  - Event handlers are functions that will be notified when a certain event fires.
  - These notifications optionally receive an event argument with details about the event (for example the x and y position of the mouse at a click event).
- The event and event-handler paradigm in JavaScript is the manifestation of the ***Observer design pattern***.
  - Another name for the Observer pattern is **Pub/Sub**, short for **Publication/Subscription**.

Code → Design → Design Pattern

- **The following will be familiar to anyone who has written a GUI program in *Java*:**

```
public class MyForm extends JFrame implements ActionListener
{
    public MyForm() {
        JButton b = new JButton();
        b.addActionListener(this);
    }

    public void actionPerformed(ActionEvent evt)
    {
        . . .
    }
}
```

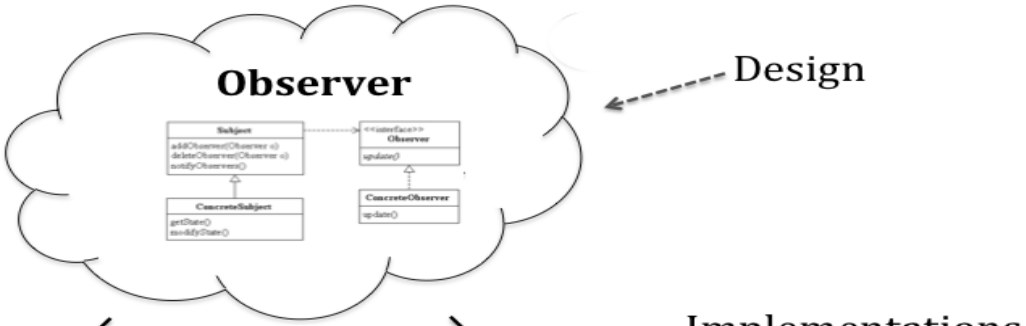
Code → Design → Design Pattern

- **Likewise, the following will be familiar to anyone who has written a GUI program in *C#*:**

```
public class MyForm : Form
{
    public MyForm() {
        Button b = new Button();
        b.Click += new EventHandler(onClick);
    }
}

public void onClick(object sender, System.EventArgs e)
{
        . . .
}
```

Code → Design → Design Pattern

- **In each case, an event handler or callback routine is registered to handle button click events.**
- **Each implementation is unique, but in both cases the design is the same.**



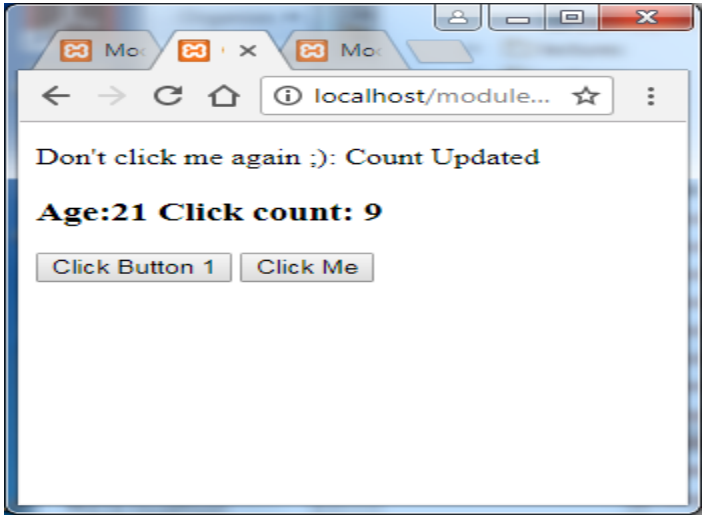
Java

```
public class MyForm extends JFrame implements ActionListener
{
    public MyForm() {
        JButton b = new JButton();
        b.addActionListener(this);
    }

    public void actionPerformed(ActionEvent evt)
    {
        . . .
    }
}
```

C#

```
public class MyForm : Form
{
    public MyForm() {
        Button b = new Button();
        b.Click += new EventHandler(onClick);
    }
}

public void onClick(object sender, System.EventArgs e)
{
        . . .
}
```

Observer design pattern example code
testObserver.html




```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Singleton Test</title>
  <script src="Observer.js"></script>
</head>
<body>
<p id="placeHolder">message goes here</p>

<button onclick="button1Click();">Click Button 1</button>
<button onclick="button2Click();">Click Me</button>
<script src="PageUpdater.js"></script>
<script src="Singleton.js"></script>
<script src="testObserver.js"></script>
</body>
</html>
```

CSWD

Observer design pattern example code

Observer.js

```
function Click() {
    this.handlers = []; // observers
}

Click.prototype = {

    subscribe: function(fn) {
        this.handlers.push(fn);
    },

    unsubscribe: function(fn) {
        this.handlers = this.handlers.filter(
            function(item) {
                if (item !== fn) {
                    return item;
                }
            }
        );
    },

    fire: function(o, thisObj) {
        var scope = thisObj || window;
        this.handlers.forEach(function(item) {
            item.call(scope, o);
        });
    }
};
```

CSWD

Observer design pattern example code `testObserver.js`

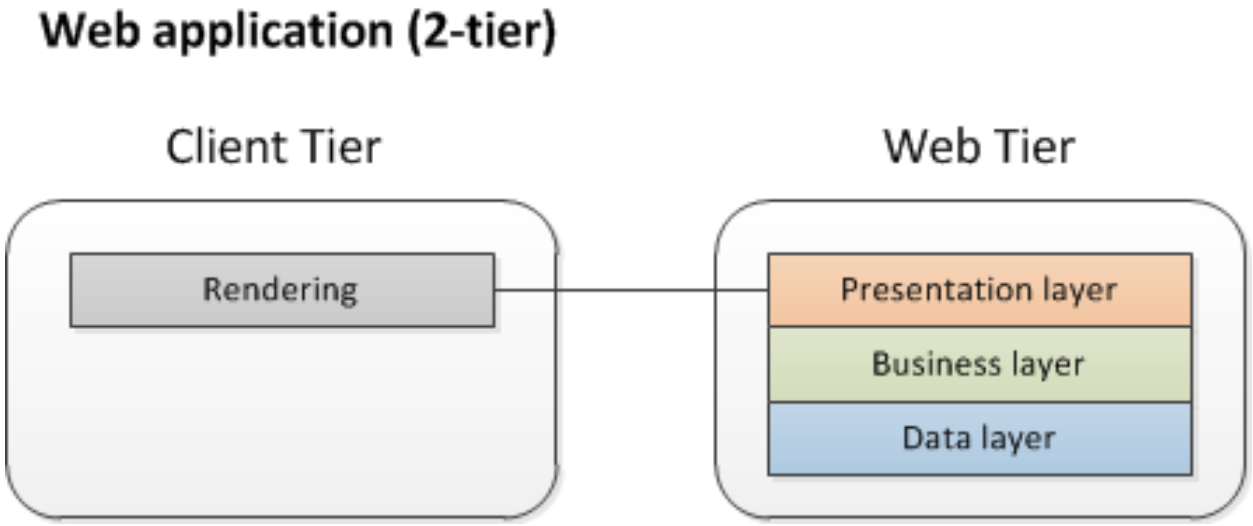
```
function button1Click() {
    Model.data.buttonCounts[0]++;
    button1.fire('Count Updated');
}

function button1Clicked (item) {
    View.firstPage = "<h1>" + Model.data.buttonCounts[0] + "</h1>";
    htmlPage.update( "placeHolder", "You clicked me: " + item + View.firstPage);
    console.log(Model);
}

function button2Click() {
    Model.data.buttonCounts[1]++;
    button2.fire('Count Updated');
}

function button2Clicked(item) {
    View.secondPage = "<h3>Age:" + Model.data.age ;
    View.secondPage += " Click count: " + Model.data.buttonCounts[1] + "</h3>";
    htmlPage.update( "placeHolder", "Don't click me again ;): " + item + View.secondPage);
    console.log(Model);
}
```

testObserver.js cont.

```
// Listen for 'button1Fire - ed' and call button1Clicked when event triggered, similar for button2
function subscribeForEventHandlers() {
    button1.subscribe( button1Clicked );
    button2.subscribe( button2Clicked );
}

var button1 = new Click();
var button2 = new Click();

var Model = Model || Singleton.getInstance();
Model.data = {name:"jim", age: 21, buttonCounts: [0,0] };

var View = View || {};

document.addEventListener( 'DOMContentLoaded', subscribeForEventHandlers );
```

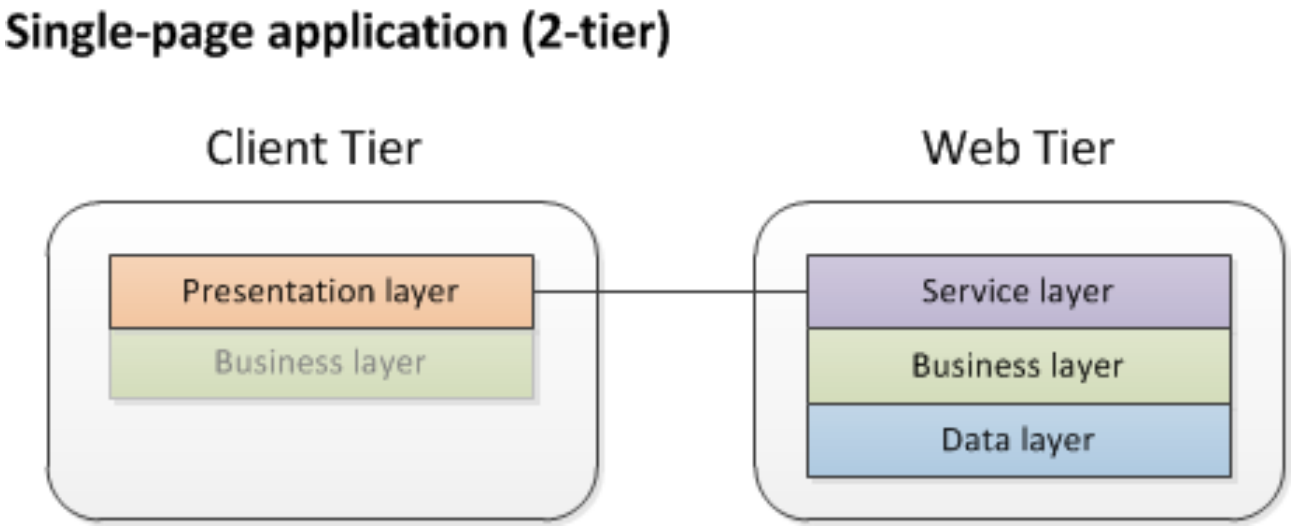
Presentation layer

- **In a "traditional" web application, presentation layer is within web tier, renders output to client tier (browser)**



Presentation layer

- **In a single-page application, presentation layer moves to client tier**
- **May also have an element of business logic in client tier**



Client Tier presentation

- **Presentation layer technologies are HTML, JavaScript, CSS**
- **Easy to end up with spaghetti code – global variables, functions whose purpose is unclear, etc.**
- **Can improve this situation using patterns which have been tried and tested in the presentation layer in the Web Tier**
  - **Eg in the coursework it is the Module and Revealing Module pattern that will deal with this.**

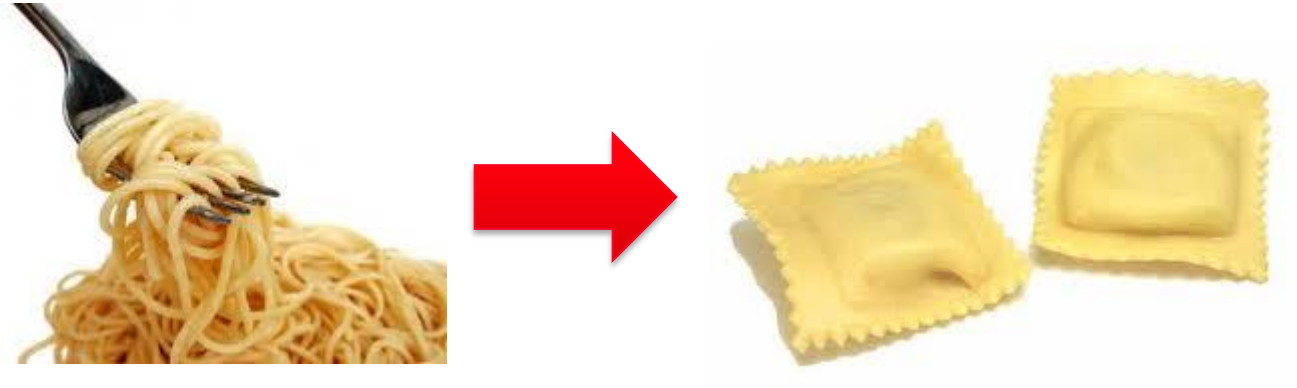
Presentation layer patterns

- **Model View Controller (MVC)**
  - **Widely used and well documented and supported in .NET by ASP.NET MVC framework**
- **Model View Presenter (MVP)**
  - **Largely superseded in ASP.NET by MVC, but basis for…**
- **Model View ViewModel (MVVM)**
  - **Not really a Web Tier pattern, used in desktop (WPF) and web (Silverlight) clients**
  - **Well documented for those applications**

CSWD

Benefits of presentation layer patterns

- Provide structure for presentation layer
- Separation of concerns
- Understandability
- Maintainability
- Testability



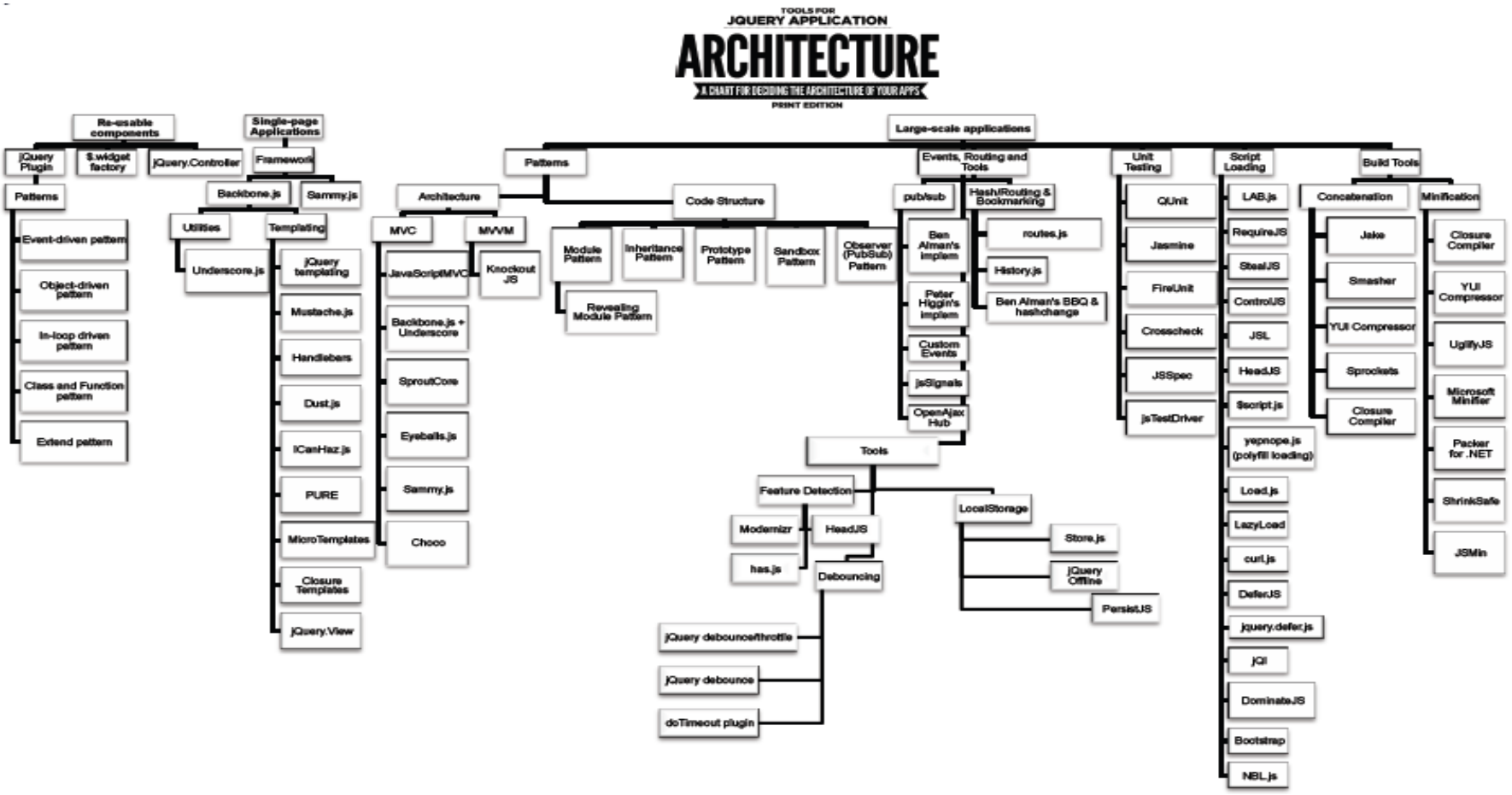
CSWD

Support for client side presentation development

- Patterns and documentation
- Many JavaScript libraries….see next slide
- Take care - wide range in scope, adoption and support for these
- Libraries play different roles in supporting development:
  - Framework for implementing patterns, e.g. Backbone.js for MVC, Knockout for MVVM
  - Support for implementing specific functionality within overall architecture, e.g. History.js

Many JavaScript libraries

- http://addyosmani.com/blog/tools-for-jquery-application-architecture-the-printable-chart/



Model-View-Controller (MVC)

- **Recap: MVC is a popular structural pattern where the idea is to divide an application into three parts so as to separate the internal representations of information from the presentation layer.**

- **MVC consists of components.**
  - **the model is the application object,**
  - **the view is the presentation of the underlying model object, and**
  - **the controller handles the way in which the user interface behaves, depending on the user interactions.**

Models

- Models are constructs that represent data in the applications which are agnostic of the user interface or routing logic.
- Changes to models are typically notified to the view layer by following the observer design pattern.
- Models may also contain code to validate, create, or delete data.
- The ability to automatically notify the views to react when the data is changed makes frameworks such as Backbone.js, Amber.js, and others very useful in building MV* applications.

Views

- Views are the visual representations of your model.
- Usually, the state of the model is processed, filtered, or *massaged* before it is presented to the view layer.
- In JavaScript, views are responsible for rendering and manipulating DOM elements.
- Views observe models and get notified when there is a change in the model.
- When the user interacts with the view, certain attributes of the model are changed via the view layer (usually via controllers).
- NB In some frameworks, the views are created using templating engines eg mustache or handlebars …
  - These templates themselves are not views.
  - They observe models and keep the view state updated based on these changes.

Controllers

- **Controllers act as a layer between models and views and are responsible for updating the model when the user changes the view attributes.**
- **Most JavaScript frameworks deviate from the classical definition of a controller.**
  - **For example, Backbone does not have a concept called controller; they have something called a router that is responsible to handle routing logic.**
  - **You can think of a combination of the view and router as a controller because a lot of the logic to synchronize models and views is done within the view itself.**

The Model-View-Presenter pattern

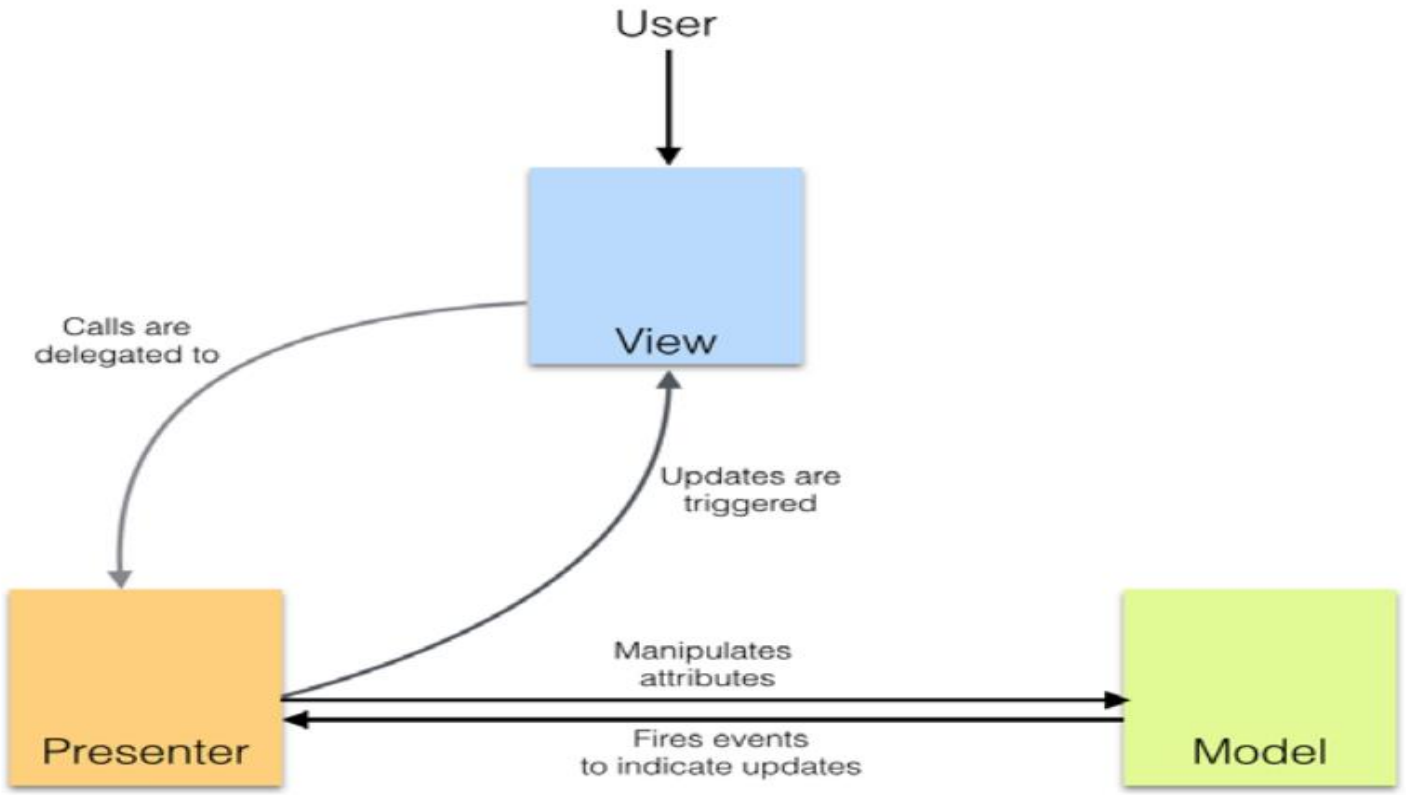
- Model-View-Presenter is a variation of the original MVC pattern that we discussed previously.
- Both MVC and MVP target the separation of concerns but they are different on many fundamental aspects.
- The presenter in MVP has the necessary logic for the view.
- Any invocation from the view gets delegated to the presenter.
- The presenter also observes the model and updates the views when the model updates.
- Many authors take the view that because the presenter binds the model with views, it also performs the role of a traditional controller.

The Model-View-Presenter pattern …

- **In implementations of MVP, the following are the primary differences that separate MVP from MVC:**
  - **The view has no reference to the model**
  - **The presenter has a reference to the model and is responsible for updating the view when the model changes**
- **MVP is generally implemented in two flavours:**
  - **Passive view: The view is as naive as possible and all the business logic is within the presenter.**
    - **For example, a plain Handlebars template can be seen as a passive view.**
  - **Supervising controller: Views mostly contain declarative logic. A presenter takes over when the simple declarative logic in the view is insufficient.**

Model-View-Presenter architecture

- **In implementations of MVP, the following are the primary differences that separate MVP from MVC:**



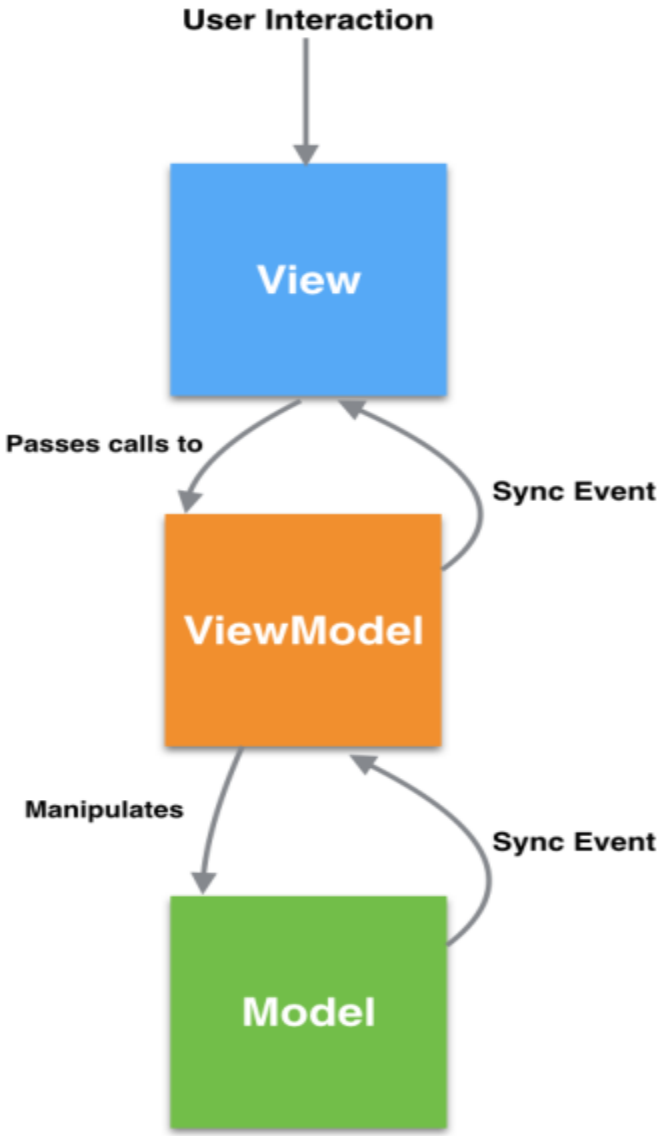
Model-View-ViewModel

- MVVM was originally coined by Microsoft for use with Windows Presentation Foundation (WPF) and Silverlight.
- MVVM is a variation of MVC and MVP and further tries to separate the user interface (view) from the business model and application behaviour.
- MVVM creates a new model layer in addition to the domain model that was discussed in MVC and MVP.
- This model layer adds properties as an interface for the view.
- Let's say that we have a checkbox on the UI. The state of the checkbox is captured in an IsChecked property. In MVP, the view will have this property and the presenter will set it.
- However, in MVVM, the presenter will have the IsChecked property and the view is responsible for syncing with it. Now that the presenter is not really doing the job of a classical presenter, it's renamed as ViewModel:

CSWD

Model-View-ViewModel

- Implementation details of these approaches are dependent on the problem that we are trying to solve and the framework that we use.
- If interested in this approach check out JavaScript implementation of this pattern supported by:
- KNOCKOUT.js



Summary

- Design will always involve some level of creativity and innovation but skillful application of design principles and patterns can help make the process a little more routine.
- While building large applications, we see certain problem patterns repeating over and over.
- These patterns have well-defined solutions that can be reused to build a robust solution.
- In this chapter, we discussed some of the important patterns and ideas around these patterns.
- Most modern JavaScript applications use these patterns.
- It is rare to see a large-scale system built without implementing modules, decorators, factories, or doesn't use MV* patterns.

Intent

- **The *intent* of a pattern is the problem solved or reason for using it.**
- **The intent of the State pattern is to allow an object to alter its behavior when its internal state changes.**
- **The intent of the Strategy pattern is to encapsulate different algorithms or behaviors and make them interchangeable from the client's perspective.**

design rules

Designing for maximum usability
– the goal of interaction design

- **Principles of usability**
  - **general understanding**
- **Standards and guidelines**
  - **direction for design**
- **Design patterns**
  - **capture and reuse design knowledge**

"[Usability refers to] the extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use." - ISO 9241-11

CSWD

Principles to support usability

- **Learnability**
  - **the ease with which new users can begin effective interaction and achieve maximal performance**
- **Flexibility**
  - **the multiplicity of ways the user and system exchange information**
- **Robustness**
  - **the level of support provided the user in determining successful achievement and assessment of goal-directed behaviour**

CSWD

Principles of learnability

- **Predictability – users don't like surprises (exception games and then only a few)**
  - **determining effect of future actions based on past interaction history**
  - **operation visibility**
    - **How quickly is someone likely to be able to do productive work with a system that is new to them**
- **Synthesizability – requires user to have a mental model (chap 1)**
  - **assessing the effect of past actions**
  - **immediate vs. eventual honesty – changing wysisyg doc vs. updating web pages**
- **Familiarity**
  - **how prior knowledge applies to new system**
  - **'guessability'; affordance**
- **Generalizability**
  - **extending specific interaction knowledge to new situations**
- **Consistency**
  - **likeness in input/output behaviour arising from similar situations or task objectives**

Principles of flexibility

- **Dialogue initiative – system and users in a conversation**
  - **freedom from system imposed constraints on input dialogue**
  - **system vs. user pre-emptiveness**
  - **understanding of main use-cases**
- **Multithreading**
  - **ability of system to support user interaction for more than one task at a time**
  - **concurrent vs. interleaving**
  - **multimodality – button click / alt + / menu item**
- **Task migratability**
  - **passing responsibility for task execution between user and system**
  - **ultimate user control**
- **Substitutivity**
  - **allowing equivalent values of input and output to be substituted for each other**
  - **representation multiplicity (graph/values)**
  - **equal opportunity (define line by drawing or specifying length/position)**
- **Customizability**
  - **modifiability of the user interface by user (adaptability) or system (adaptivity)**

CSWD

Principles of robustness

- **Observability**
  - ability of user to evaluate the internal state of the system from its perceivable representation
  - browsability; defaults; reachability; persistence; operation visibility
- **Recoverability**
  - ability of user to take corrective action once an error has been recognized
  - reachability; forward/backward recovery; commensurate effort
- **Responsiveness**
  - how the user perceives the rate of communication with the system
  - Stability
- **Task conformance**
  - degree to which system services support all of the user's tasks
  - task completeness; task adequacy

CSWD

Standards

- set by national or international bodies to ensure compliance by a large community of designers standards require sound underlying theory and slowly changing technology
  - many large organisations have their own standards
- hardware standards more common than software high authority and low level of detail
- ISO 9241 defines usability as effectiveness, efficiency and satisfaction with which users accomplish tasks
- There are also some ISO standards for usability reporting

Guidelines

- more suggestive and general
- many textbooks and reports full of guidelines
- abstract guidelines (principles) applicable during early life cycle activities
- detailed guidelines (style guides) applicable during later life cycle activities
- understanding justification for guidelines aids in resolving conflicts

Web Guidelines

- **The W3C Web Accessibility Guidelines are a really good, very practical guide to usability**
- **If you follow these guidelines your system will be good, not only for visually impaired users, but for all users**
- **Furthermore most of the guidelines apply to all systems, not just web systems**
- **http://www.w3.org/WAI/**

- **The user experience is now an important concept in many products with products needing to deliver more than just a '*good interface*'.**

User Interface Design Patterns

- **User Interface (UI) Design patterns are recurring solutions that solve common design problems.**
  - **Design patterns are standard reference points for the experienced user interface designer.**
- **Design patterns provide a common language between designers.**
- **They allow for debate over alternatives, where merely mentioning the name of a design pattern implicitly carries much more meaning than merely the name.**

UI Patterns

- **Consistency**
  - **‘Look and Feel’ should be same across all pages**
    - » **typography, colours, fonts, backgrounds, layout**
- **Accessibility**
  - **Compatibility with a variety of different devices**
- **Learnability**
  - **Intuitive user interfaces where users make use of what they already know**
  - **eg feedback mechanisms, progress indicators, use of wizards, use of back button, ‘undo’ facility, current status, input forms ...**
- **Navigability (see next few slides for ideas)**
  - **Keep primary navigation simple**
  - **Use breadcrumbs for each page (except home)**
  - **Add search capability at top of site**

Sample UI patterns - Navigation

- **Problem/Intent:** ***The user needs to locate specific features and content and needs navigation to accomplish this; alternatively used when space is limited***

- **Solution:**
  - **Tabs**
  - **Menus**
    - » **Vertical/Horizontal/Accordion)**
  - **Carousel**
  - **Favourites**
  - **Progressive Disclosure**
  - **Tag Clouds**
  - **Thumbnails**
  - **Pagination**

CSWD

Navigation ideas

- **Problem/Intent: The user needs to know their location in the website's hierarchical structure**
- **Solution:**
  - **Breadcrumb**

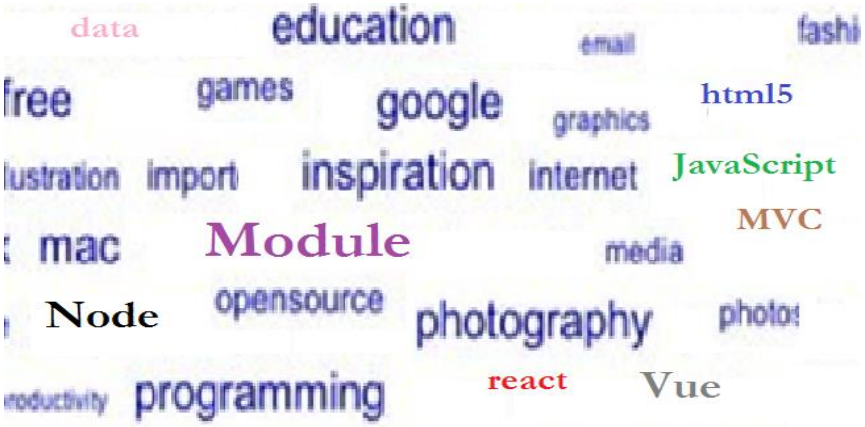
- **Usage:**
  - **Use when the structure of the site is partitioned in to sections which can be divided into more subsections and so on.**
  - **Use when the user is most likely to have landed on the page from an external source**
  - **Use when the page in question is placed fairly deep into the hierarchy of pages and when no other form of visual navigation can show the details of the same deep level.**
  - **Best Used together with some sort of main navigation.**

CSWD

Navigation ideas …

- **Problem/Intent: The user wants to browse content by popularity in a visually appealing way.**

- **Solution:**
  - **Consider using tag clouds**



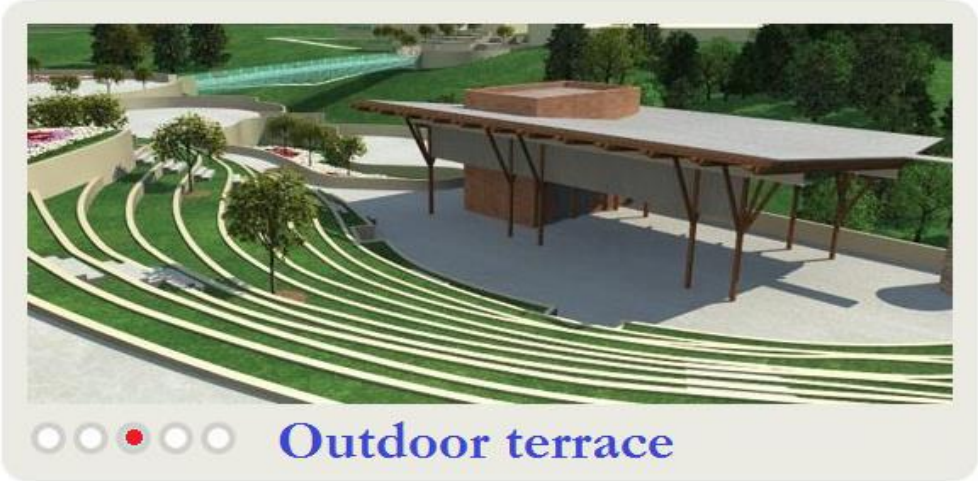
- **Usage:**
  - **Use when users of your website can add their own content and tags.**
  - **Use when your website has 10 or more different tags, each with different weights implying some sort of frequency or rating value**
  - **To increase visual appearance consider using different colours and fonts to show 'rating' values**

CSWD

Navigation ideas …

- **Problem/Intent: The user has a collection of media that needs to be displayed in a presentation as a sequence of images.**

- **Solution:**
  - **Consider using carousel or gallery with textual info**



- **Usage:**
  - **Use when the user needs to browse through a series of images in a sequential way**
  - **Use to enforce a sense of order as in ‘first’ and ‘last’.**

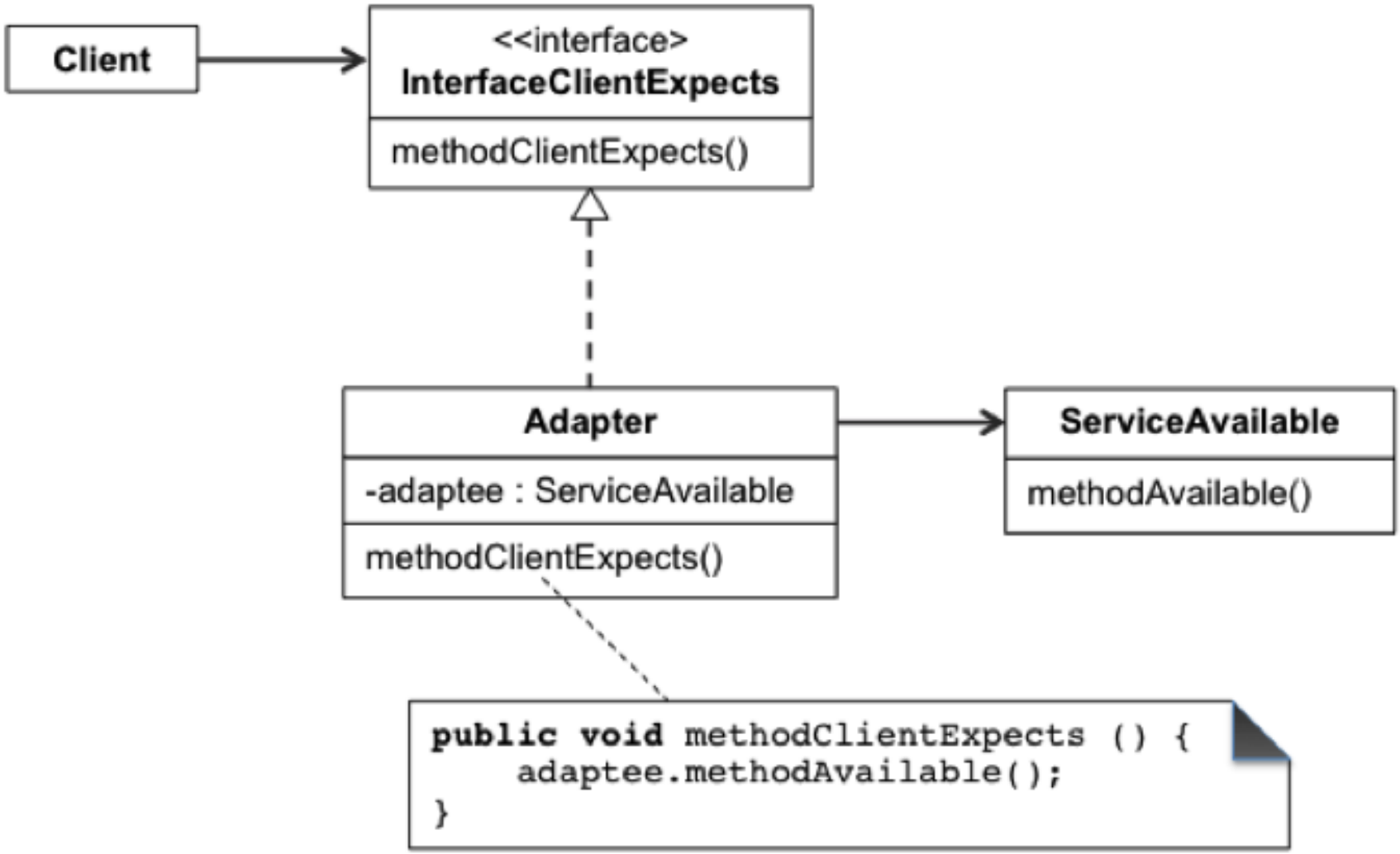
CSWD

Additional Patterns

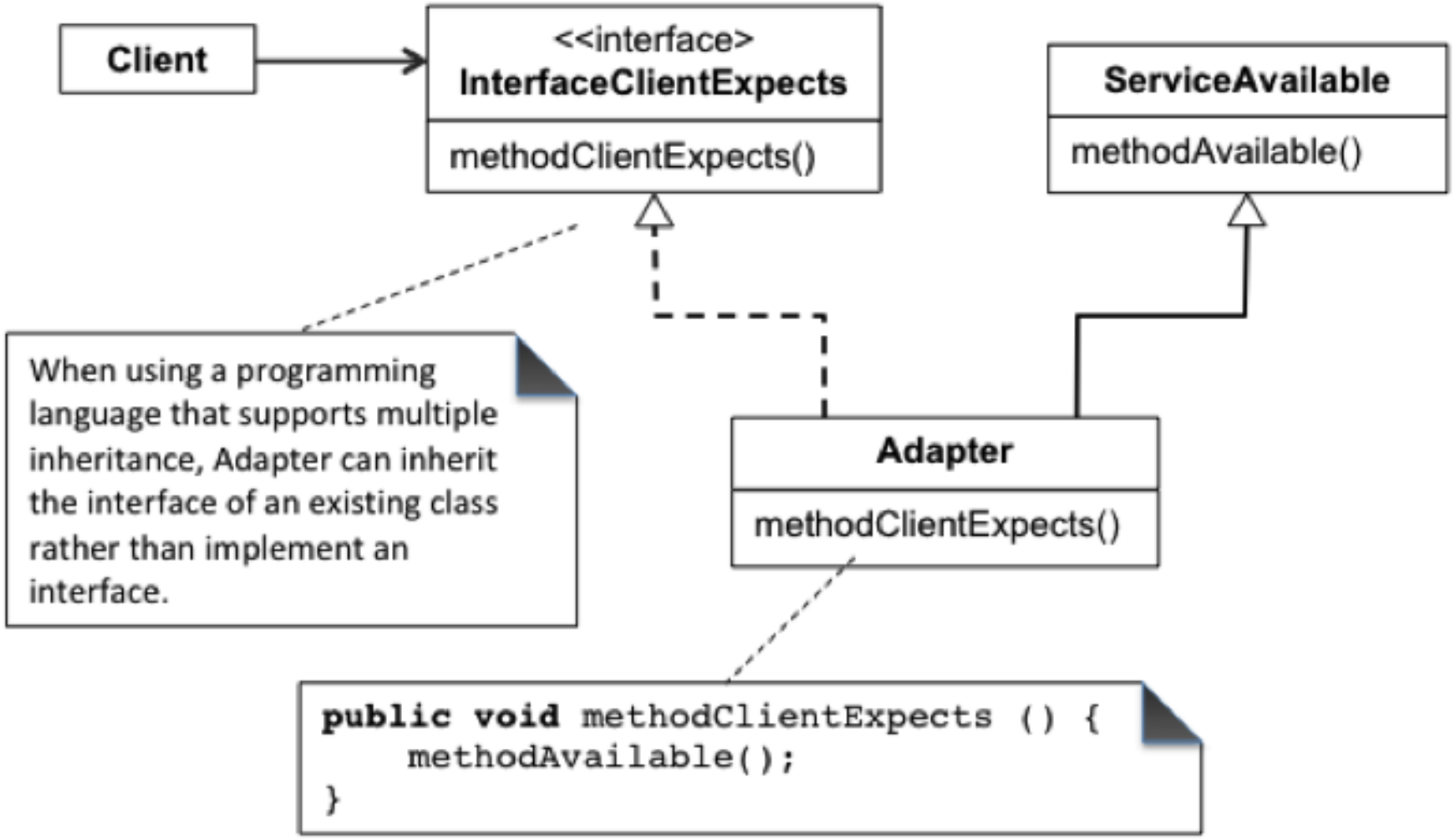
- **For those interested in patterns the next few slides identify some of the patterns that are generally found in software programming**

Adapter

- **Intent – The Adapter design pattern is useful in situations where an existing class provides a needed service but there is a mismatch between the interface offered and the interface clients expect. The Adapter pattern shows how to convert the interface of the existing class into the interface clients expect.**



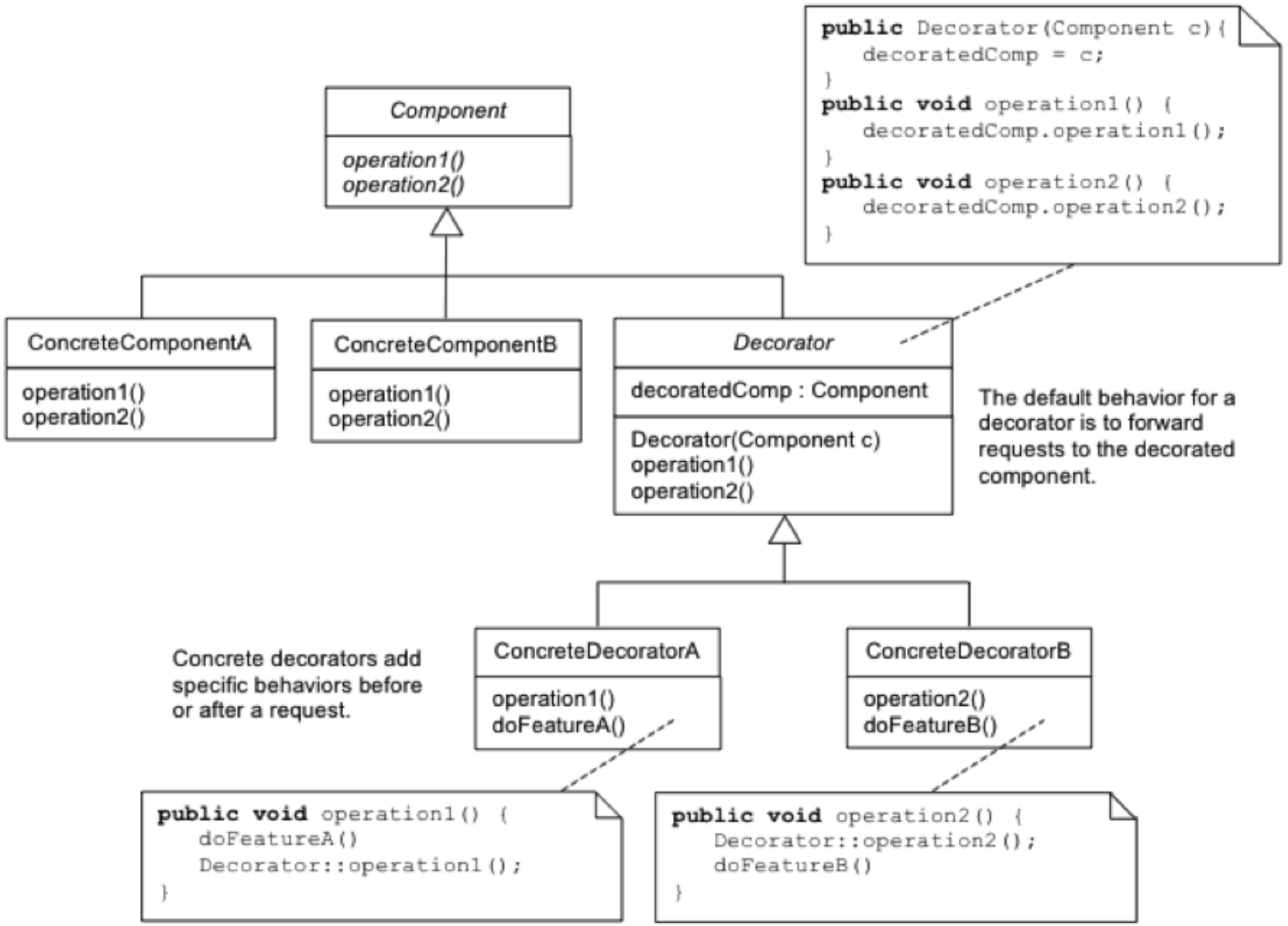
Solution using composition



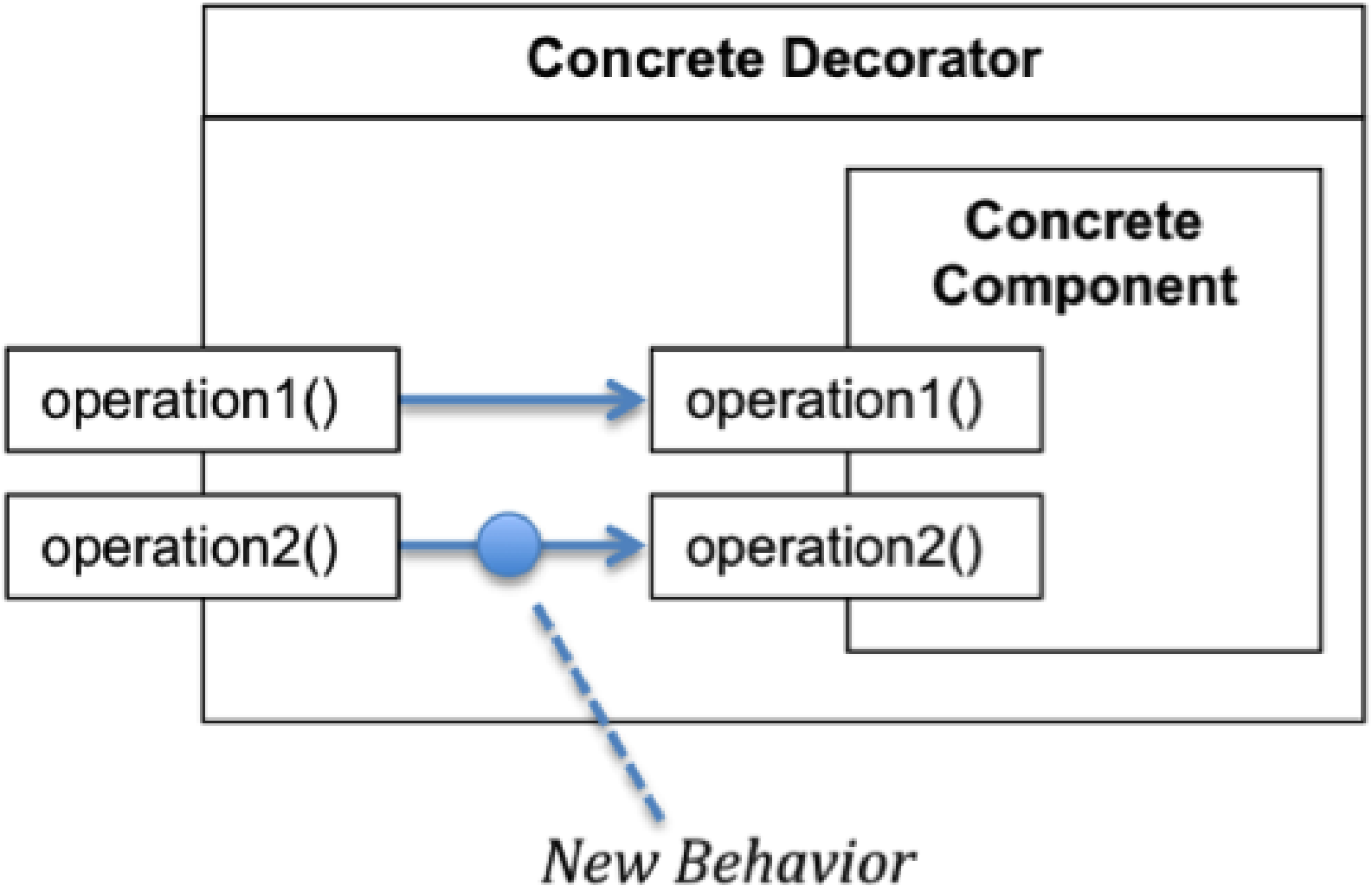
Solution using inheritance

Decorator

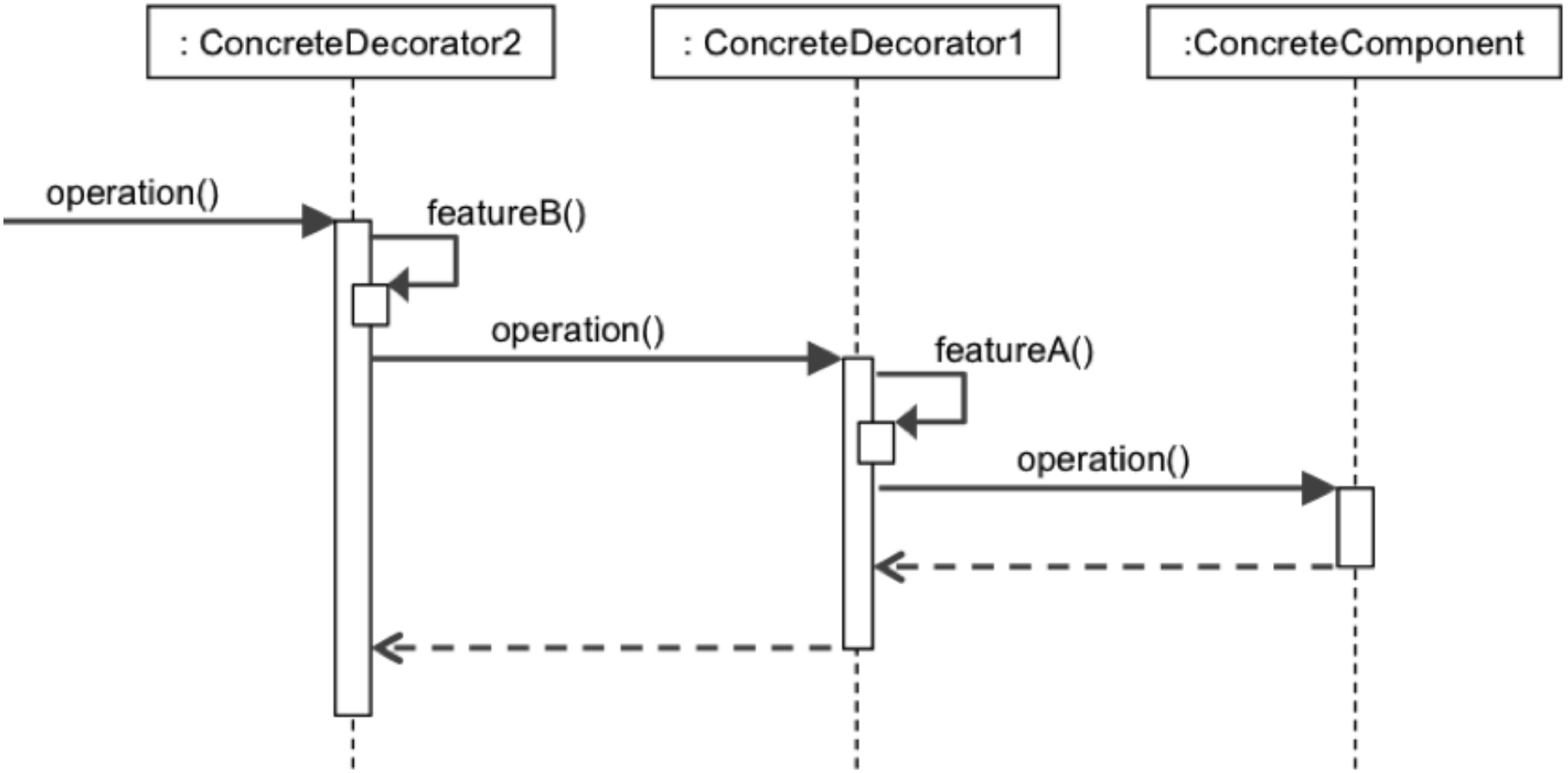
- **Intent – The decorator design pattern provides a way of attaching additional responsibilities to an object dynamically. It uses object composition rather than class inheritance for a lightweight flexible approach to adding responsibilities to objects at runtime.**



Solution – Static Structure



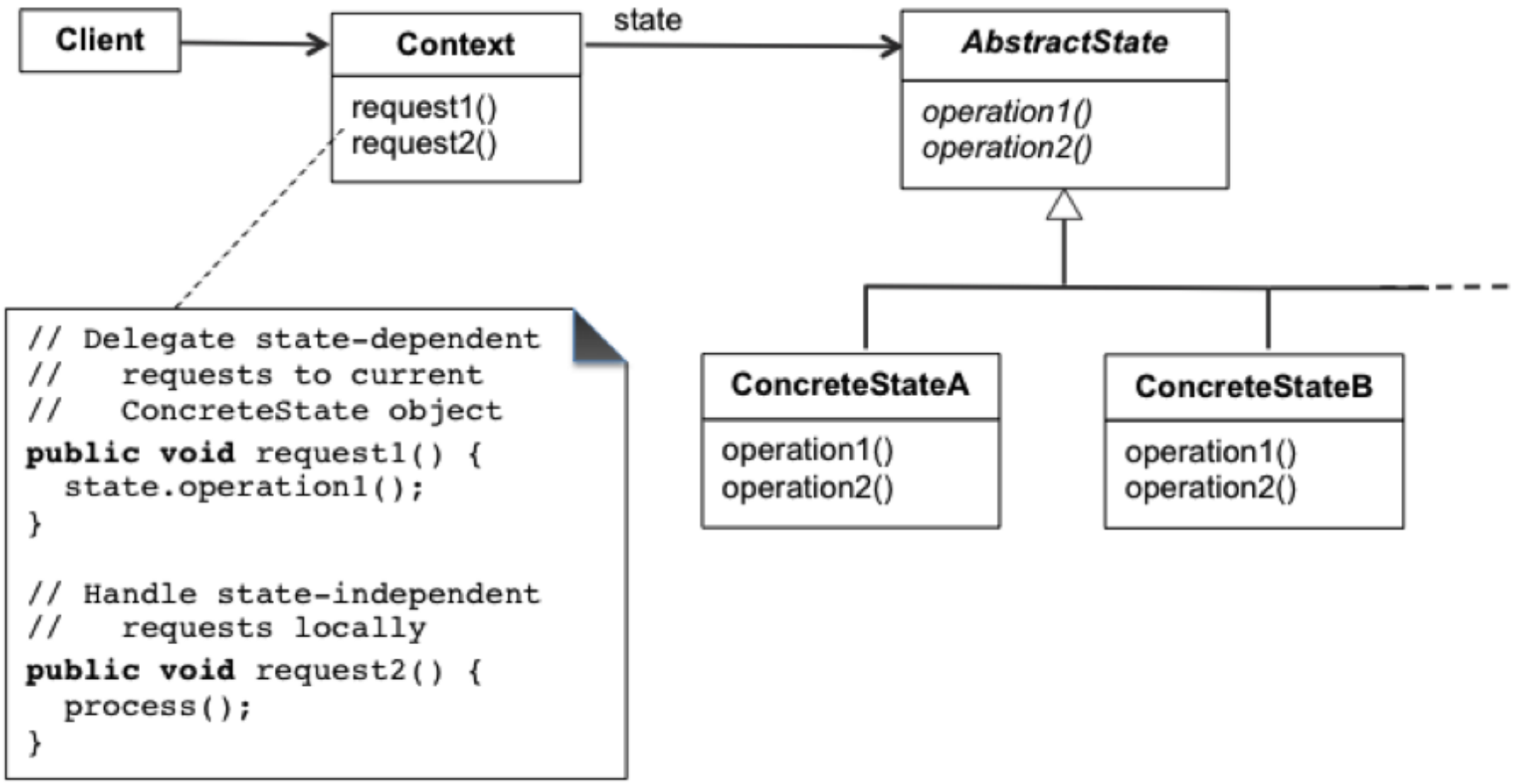
Conceptual Diagram



Solution – Dynamic Behavior

State

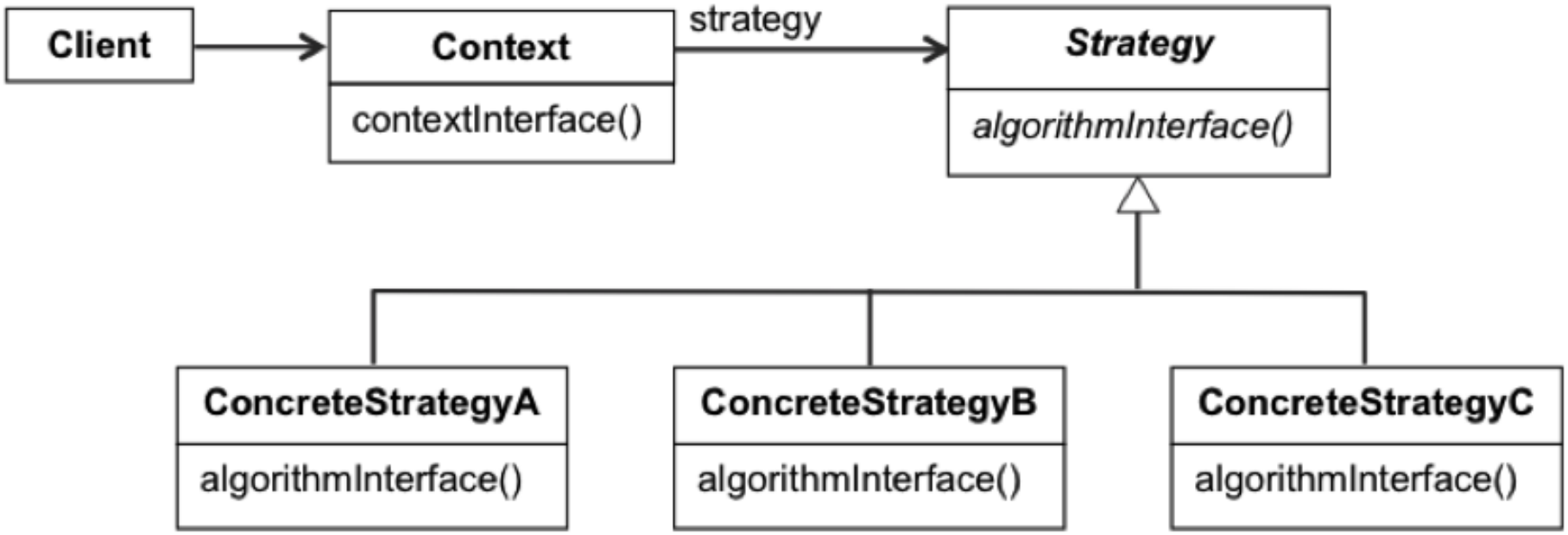
- **Intent – If an object goes through clearly identifiable states, and the object's behavior is especially dependent on its state, it is a good candidate for the State design pattern.**


Client
Context
request1()
request2()
state
AbstractState
operation1()
operation2()
ConcreteStateA
operation1()
operation2()
ConcreteStateB
operation1()
operation2()
// Delegate state-dependent
// requests to current
// ConcreteState object
public void request1() {
state.operation1();
}
// Handle state-independent
// requests locally
public void request2() {
process();
}

Solution

Strategy

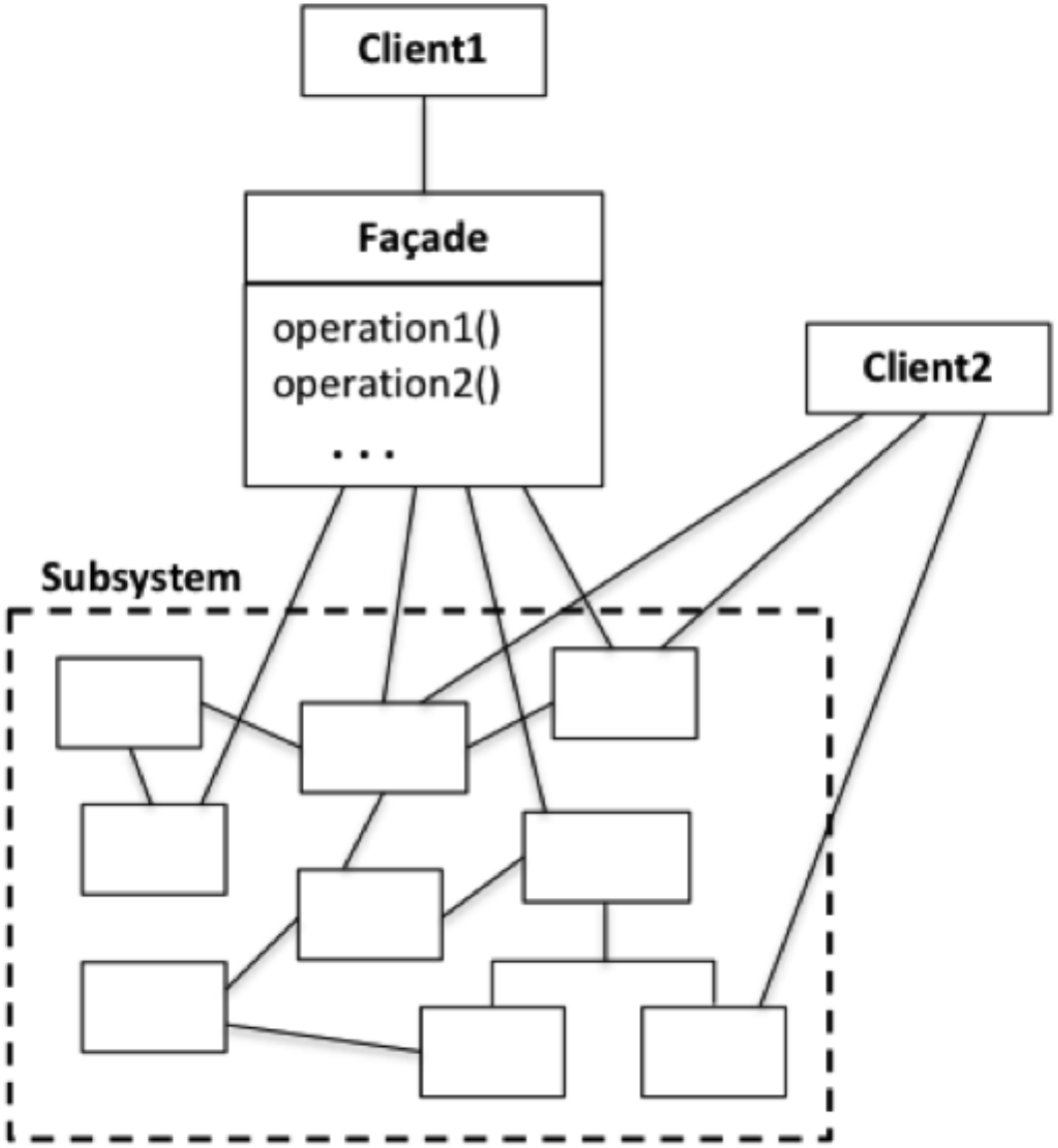
- **Intent – The Strategy design pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it. [Gang of Four].**


Client
Context
contextInterface()
strategy
Strategy
algorithmInterface()
ConcreteStrategyA
algorithmInterface()
ConcreteStrategyB
algorithmInterface()
ConcreteStrategyC
algorithmInterface()

Solution

Façade

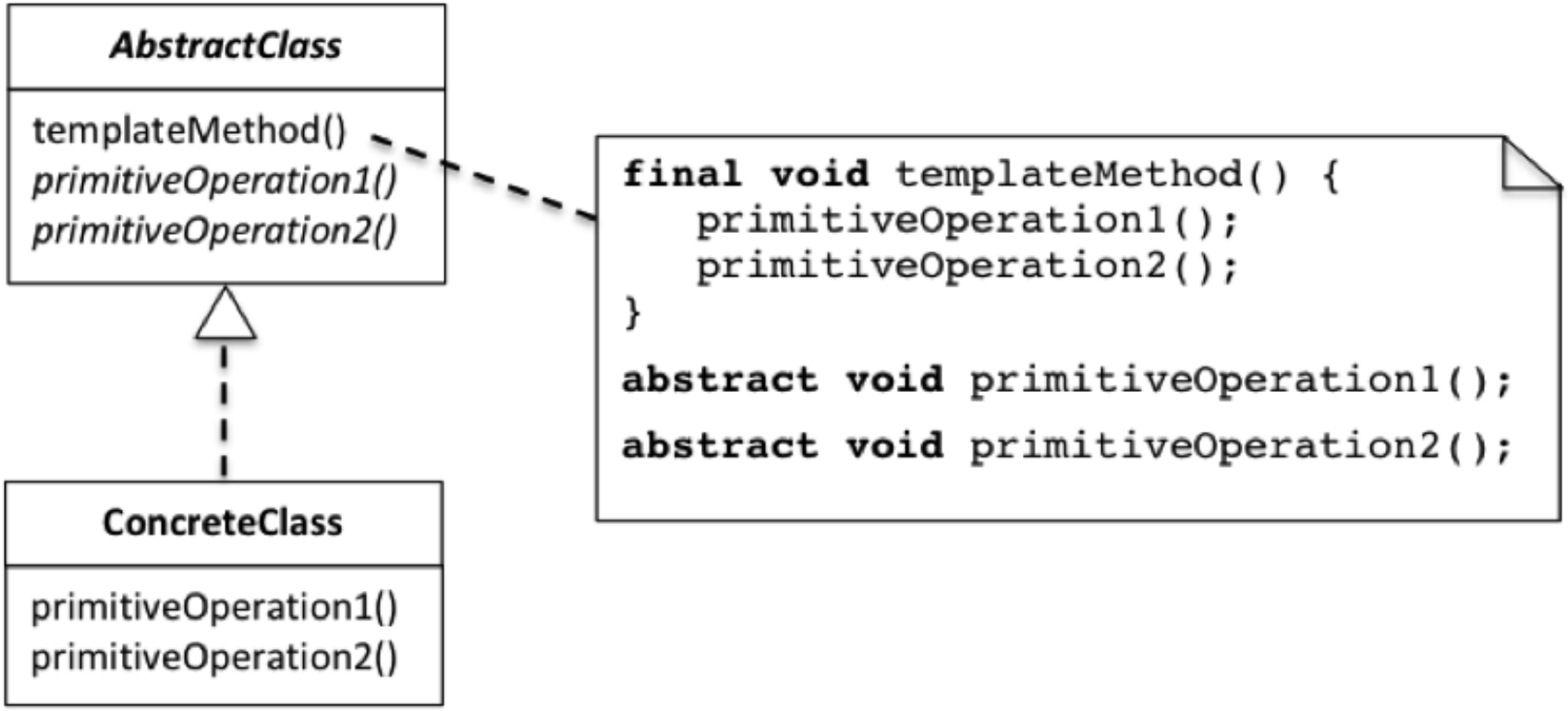
- **Intent – The intent of the Façade design pattern is to "provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use." [Gang of Four]**



Solution

Template Method

- **Intent – With the Template Method design pattern the structure of an algorithm is represented once with variations on the algorithm implemented by subclasses. The skeleton of the algorithm is declared in a template method in terms of overridable operations. Subclasses are allowed to extend or replace some or all of these operations.**


AbstractClass
templateMethod()
primitiveOperation1()
primitiveOperation2()
ConcreteClass
primitiveOperation1()
primitiveOperation2()
final void templateMethod() {
primitiveOperation1();
primitiveOperation2();
}
abstract void primitiveOperation1();
abstract void primitiveOperation2();

Solution