CS 5004: OBJECT ORIENTED DESIGN AND ANALYSIS

SPRING 2022

LECTURE 11

Divya Chaudhary

Northeastern University
Khoury College of Computer Sciences

440 Huntington Avenue ▪ 202 West Village H ▪ Boston, MA 02115 ▪ T 617.373.2462 ▪ khoury.northeastern.edu

AGENDA

- Course logistics
- Review – Design of programs
- Design of programs
  - Creational patterns
  - Structural patterns
  - Behavioral patterns
- Model-view control (MVC)

COURSE LOGISTICS

- Last Class Next Week
- HW9 due: Next week
- Final Exam – May 2nd - Full day to complete it.(Available for 1 day)
  - 4 programming problems
  - Submit in your individual repositories.

REVIEW – DESIGN OF PROGRAMS

CS 5004, SPRING 2022 – LECTURE 11

REVIEW: DESIGN PROBLEM FORMULATION

- How to decompose a system into parts, each with a lower complexity of the whole system, such that:
  - Interaction between parts is minimized
  - Combination of the parts together solves the problem

- No universal way to do this!
- Some rules of thumb:
  - Don't think of the system in terms of components that correspond to steps in processing: This adds complexity
  - Do provide a set of modules that are useful for writing many programs. Plan for reuse!

REVIEW: MODULAR DESIGN

- Module: some part of a program responsible for a specific area of functionality, and designed to interact with other modules

- Modular design regards modules as design units
  - Focus on:
    1. How are modules specified?
    2. What functionality they govern?
    3. How modules interact?

REVIEW: COHESION AND COUPLING

- Object-oriented design seeks to **maximize** cohesion, while **minimizing** coupling

- Cohesion
  - How well a module (class) encapsulate a single responsibility → **does everything in the class belong to the class?**

- Coupling
  - Interdependencies between classes → **does a class depend on some other class to be operational?**

DESIGNING SOFTWARE

- Design is *creative* problem solving
  - No surefire recipe for success
  - Learn from others' experience
  - Following known best practices is a a significant success factor

DESIGN PROBLEM FORMULATION

- How to decompose a system into parts, each with a lower complexity of the whole system, such that:
  - Interaction between parts is minimized
  - Combination of the parts together solves the problem

- No universal way to do this!
- Some rules of thumb:
  - Don't think of the system in terms of components that correspond to steps in processing: This adds complexity
  - Do provide a set of modules that are useful for writing many programs. Plan for reuse!

WHAT GOES WRONG WITH SOFTWARE?

- *But then something begins to happen. The software starts to rot. At first it isn't so bad. An ugly wart here, a clumsy hack there, but the beauty of the design still shows through. Yet, over time as the rotting continues, the ugly festering sores and boils accumulate until they dominate the design of the application. The program becomes a festering mass of code that the developers find increasingly hard to maintain.*

Robert C. Martin, Design Principles and Design Patterns

ROTTING SOFTWARE DESIGN

- Symptoms
  - **Rigidity**: hard to change
  - **Fragility**: breaks easily
  - **Immobility**: hard to reuse code
  - **Viscosity**: structure breaks down

- Causes
  - Changing requirements
  - Dependency management

DESIGN PATTERNS

CS 5004, SPRING 2022 – LECTURE 11

ELEMENTS OF A DESIGN PATTERN

- Name
  - Important because it becomes part of a design vocabulary
- Problem
  - When the pattern is applicable
- Solution
  - Design elements and their relationships
  - Abstract: must be specialized
- Consequences
  - Tradeoffs of applying the pattern
    - Each pattern has costs, as well as benefits
    - Issues include flexibility, extensibility, security…
    - There may be variations in the pattern with different consequences

TYPES OF DESIGN PATTERNS

- Creational design patterns - focus on ways to create or control the creation of objects
- Structural design patterns - focus on object composition, relations between objects (e.g. inheritance), and relations between objects and the system as a whole
- Behavioral patterns - focus on improving or streamlining communication between objects

CREATIONAL DESIGN PATTERNS

- Factory Method Pattern - write a method to return new instances of a class (or subclass, based on parameters sent to the method) without client object needing to call constructor directly
- Abstract Factory Pattern – a class that provides a family of factories to handle the responsibility of creating instances of subclasses
- Builder Pattern - a class that encapsulates a complex object creation process, e.g. including optional parameters
- Prototype Pattern - create objects by creating a prototypical instance and cloning new objects from that
- Singleton Pattern - create a single instance of a class. Constructor is private, called by a static method (*regarded by critics as an anti-pattern*)

STRUCTURAL DESIGN PATTERNS

- Adapter (wrapper) – a class to take objects of another class, and enable them to implement an interface that their original class does not implement
- Bridge - decouple an abstraction from its implementation so that the two can vary independently
- Composite - compose objects into tree structures to represent part-whole hierarchies. Enables clients to treat objects and compositions of objects uniformly.
- Decorator - attach additional behavior or responsibilities to an object dynamically. An alternative to subclassing for extending functionality

STRUCTURAL DESIGN PATTERNS - CONT

- Façade - provide a unified interface to a set of interfaces in a subsystem. Simplify use of a complex subsystem by wrapping it in a simpler, higher-level interface
- Flyweight - maintain state that is shared by multiple objects in its own container, rather than duplicating state across objects
- Proxy - create an object to hold the place of another object, able to carry out certain functionality on behalf of the original object, possibly in situations where the original object would not be available to do so.

BEHAVIORAL DESIGN PATTERNS

- Chain of Responsibility - build a sequence of handlers where each handler processes a request and decides whether to pass it on to the next handler
- Command - abstract multiple related processes (functions or methods) into a single process, distinguishing the original cases by parameters
- Iterator - separates the structure of a collection from the process of iterating over it by creating class that encapsulates a particular approach to traversing the collection. For example, a tree-structure collection may be associated with a depth-first iterator and a breadth-first iterator, and the appropriate one would be used to iterate through items in the tree structure.
- Mediator - a class that mediates communication between other classes. Simplifies potentially complex entanglements by routing all communications through the mediator.

BEHAVIORAL DESIGN PATTERNS

- Memento - create a representation of internal state (which can include private state) that can be used to restore the state of the original object
- Observer (publisher/subscriber) - enable multiple objects to respond to events or state changes on another object
- State - enable an object to change its behavior based on the state it's in
- Strategy - create classes representing separate algorithms for accomplishing a similar task (e.g. a route planning tool that works for bike, automobile, public transportation, or walk route planning)
- Template Method - design the skeleton of an algorithm in a superclass, but let subclasses determine the specifics of the implementation
- Visitor - implement functionality in separate classes that can "visit" objects of dissimilar classes and lend them consistent functionality.

CREATIONAL PATTERNS

CS 5004, SPRING 2022 – LECTURE 11

CREATIONAL PATTERNS

Constructors in Java are inflexible

1. Can't return a subtype of the class they belong to
2. Always return a fresh new object, never re-use one

Factories: Patterns for code that you call to get new objects other than constructors

- Factory method, Factory object, Prototype, Dependency injection

Sharing: Patterns for reusing objects (to save space *and* other reasons)

- Singleton, Interning, Flyweight

SINGLETON PATTERN

For some class C, guarantee that at run-time there is exactly one instance of C

- And that the instance is globally visible

First, *why* might you want this?

- What design goals are achieved?

Second, *how* might you achieve this?

- How to leverage language constructs to enforce the design

A pattern has a recognized *name*

- This is the *Singleton Pattern*

POSSIBLE REASONS FOR SINGLETON PATTERN

- One **`RandomNumber`** generator
- One **`KeyboardReader`**, **`PrinterController`**, etc…
- Have an object with fields/properties that are "like public, static fields" but you can have a constructor decide their values
  - Maybe strings in a particular language for messages

- Make it easier to ensure some key invariants
  - There is only one instance, so never mutate the wrong one

- Make it easier to control when that single instance is created
  - If expensive, delay until needed and then don't do it again

SINGLETON PATTERN – MULTIPLE APPROACHES

```
public class Foo {
  private static final Foo instance = new Foo();
  // private constructor prevents instantiation outside class
  private Foo() { … }
  public static Foo getInstance() {
    return instance;
  }
  … instance methods as usual …
}
```

Eager allocation of instance

```
public class Foo {
  private static Foo instance;
  // private constructor prevents instantiation outside class
  private Foo() { … }
  public static synchronized Foo getInstance() {
    if (instance == null) {
      instance = new Foo();
    }
    return instance;
  }
  … instance methods as usual …
}
```

Lazy allocation of instance

MOTIVATION FOR FACTORY PATTERN

Supertypes support multiple implementations

```
interface Matrix { ... }
class SparseMatrix implements Matrix { ... }
class DenseMatrix implements Matrix { ... }
```

Clients use the supertype (**`Matrix`**)

Still need to use a **`SparseMatrix`** or **`DenseMatrix`** constructor

- Must decide concrete implementation *somewhere*
- Don't want to change code to use a different constructor
- Factory methods put this decision behind an abstraction

USE OF FACTORY PATTERN

Factory

```
class MatrixFactory {
  public static Matrix createMatrix() {
    return new SparseMatrix();
  }
}
```

Clients call **`createMatrix`** instead of a particular constructor

Advantages:

- To switch the implementation, change only *one* place
- **`createMatrix`** can do arbitrary computations to decide what kind of matrix to make (unlike what's shown above)

DATEFORMAT FACTORY PATTERN

`DateFormat` class encapsulates knowledge about how to format dates and times as text

- Options: just date? just time? date+time? where in the world?
- Instead of passing all options to constructor, use factories
- The subtype created by factory call need not be specified

```
DateFormat df1 = DateFormat.getDateInstance();
DateFormat df2 = DateFormat.getTimeInstance();
DateFormat df3 = DateFormat.getDateInstance
                    (DateFormat.FULL, Locale.FRANCE);

Date today = new Date();

df1.format(today)  // "Jul 4, 1776"
df2.format(today)) // "10:15:00 AM"
df3.format(today));// "jeudi 4 juillet 1776"
```

ABSTRACT FACTORY VS. FACTORY METHOD

- Factory Method pattern:
  - A single method
  - Uses inheritance and relies on subclasses to handle desired object instantiation

- Abstract Factory pattern:
  - Encapsulates many factory methods
  - Single responsibility for creating a family of objects
  - Another class delegates responsibility of object instantiation to Factory class

FACTORY METHOD

```
class Race {
    public Race() {
        Bicycle bike1 = new Bicycle();
        Bicycle bike2 = new Bicycle();
        ...
    }
    ...
}

class TourDeFrance extends Race {
    public TourDeFrance() {
        Bicycle bike1 = new RoadBicycle();
        Bicycle bike2 = new RoadBicycle();
        ...
```

FACTORY METHOD

```
class Race {
    public Race() {
        Bicycle bike1 = new Bicycle();
        Bicycle bike2 = new Bicycle();
        ...
    }
    ...
}

class TourDeFrance extends Race {
    public TourDeFrance() {
        Bicycle bike1 = new RoadBicycle();
        Bicycle bike2 = new RoadBicycle();
        ...
```

- Problem: re-implementing constructor in every `Race` subclass to use a different subclass of `Bicycle`

FACTORY METHOD

- Problem: re-implementing constructor in every `Race` subclass to use a different subclass of `Bicycle`
- Solution: use a Factory method to avoid dependency on specific new kind of `Bicycle` in constructor

```
class Race {
    Bicycle createBicycle() { return new Bicycle(); }
    public Race() {
        Bicycle bike1 = createBicycle();
        Bicycle bike2 = createBicycle();
        ...
    }
    ...
}
```

FACTORY METHOD

- Solution: use a Factory method to avoid dependency on specific new kind of `Bicycle` in constructor
- Subclasses can override Factory method, and return any subtype of `Bicycle`

```
class TourDeFrance extends Race {
    Bicycle createBicycle() { return new RoadBicycle(); }
    public TourDeFrance() { super() }
    ...
}

class Cyclocross extends Race {
    Bicycle createBicycle() { return new MountainBicycle(); }
    public Cyclocross() { super() }
    ...
}
```

FACTORY METHOD

- Encapsulation: move the factory method into a separate class - a factory object
- Advantages:
  - We can now pass factories around as objects for flexibility
  - We can choose a factory at runtime
  - We can use different factories in different objects (e.g. races)
- Promotes composition over inheritance
- Supports separation of concerns

ABSTRACT FACTORY EXAMPLES

```
class BicycleFactory {
    Bicycle createBicycle() {
        return new Bicycle();
    }
}

class RoadBicycleFactory extends BicycleFactory {
    Bicycle createBicycle() {
        return new RoadBicycle();
    }
}

class MountainBicycleFactory extends BicycleFactory {
    Bicycle createBicycle() {
        return new MountainBicycle();
```

ABSTRACT FACTORY EXAMPLES

```
class Race {
    BicycleFactory bfactory;
    public Race(BicycleFactory f) {
        bfactory = f;
        Bicycle bike1 = bfactory.createBicycle();
        Bicycle bike2 = bfactory.createBicycle();
    }
}

class TourDeFrance extends Race {
    public TourDeFrance() {
        super(new RoadBicycleFactory());
    }
}
```

STRUCTURAL PATTERNS

CS 5004, SPRING 2022 – LECTURE 12

STRUCTURAL PATTERNS

A wrapper translates between incompatible interfaces

Wrappers are a thin veneer over an encapsulated class

- Modify the interface
- Extend behavior
- Restrict access

The encapsulated class does most of the work

Pattern	Functionality	Interface
Adapter	**same**	**different**
Decorator	**different**	**same**
Proxy	**same**	**same**

Some wrappers have qualities of more than one of adapter, decorator, and proxy

ADAPTER PATTERN

Change an interface without changing functionality

- Rename a method
- Convert units
- Implement a method in terms of another

Example: angles passed in radians vs. degrees

Example: use “old” method names for legacy code

ADAPTER PATTERN

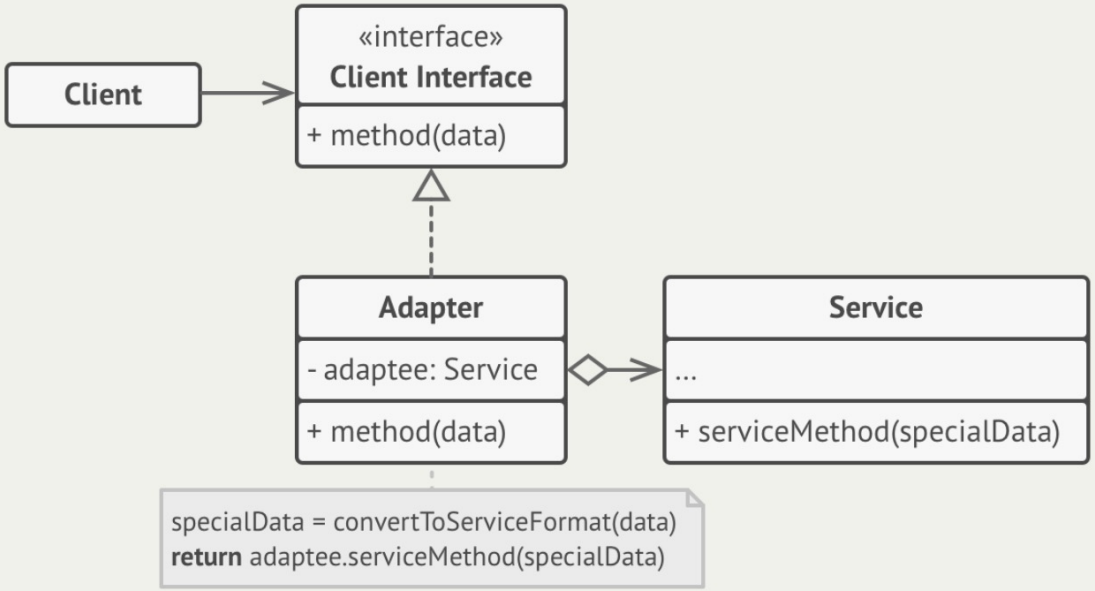
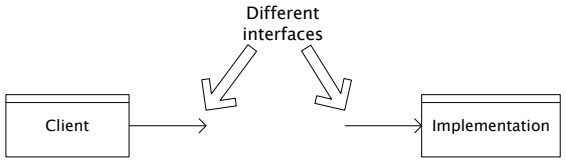


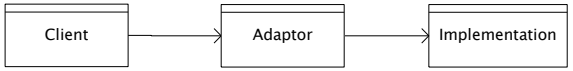
Image credit: https://refactoring.guru/design-patterns/adapter

TYPES OF ADAPTER PATTERN

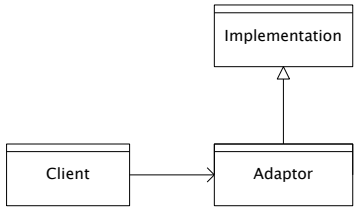
Goal of adapter:
connect incompatible interfaces



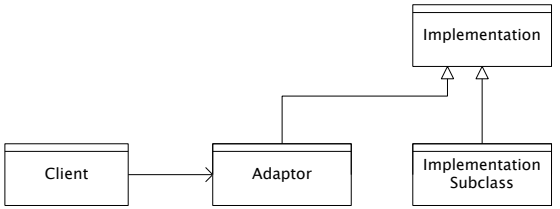
Adapter with delegation



Adapter with subclassing



Adapter with subclassing:
no extension is permitted



ADAPTER PATTERN - EXAMPLE

We have this **`Rectangle`** interface

```
interface Rectangle {
  // grow or shrink this by the given factor
  void scale(float factor);
  ...
  float getWidth();
  float area();
}
```

Goal: client code wants to use this library to "implement" **`Rectangle`** without rewriting code that uses **`Rectangle`**:

```
class NonScaleableRectangle { // not a Rectangle
  void setWidth(float width) { ... }
  void setHeight(float height) { ... }
   // no scale method
  ...
```

ADAPTER PATTERN: USE SUBCLASSING

```
class ScaleableRectangle1
                extends NonScaleableRectangle
                implements Rectangle {
  void scale(float factor) {
    setWidth(factor * getWidth());
    setHeight(factor * getHeight());
  }
}
```

ADAPTER PATTERN: USE DELEGATION

Delegation: forward requests to another object

```
class ScaleableRectangle2 implements Rectangle {
  NonScaleableRectangle r;
  ScaleableRectangle2(float w, float h) {
    this.r = new NonScaleableRectangle(w,h);
  }

  void scale(float factor) {
    r.setWidth(factor * r.getWidth());
    r.setHeight(factor * r.getHeight());
  }

  float getWidth() { return r.getWidth(); }
  float circumference() {
     return r.circumference();
  }
  ...
```

ADAPTER PATTERN: SUBCLASSING VS. DELEGATION

Subclassing

- automatically gives access to all methods of superclass
- built in to the language (syntax, efficiency)

Delegation

- permits removal of methods (compile-time checking)
- objects of arbitrary concrete classes can be wrapped
- multiple wrappers can be composed

BEHAVIORAL PATTERNS

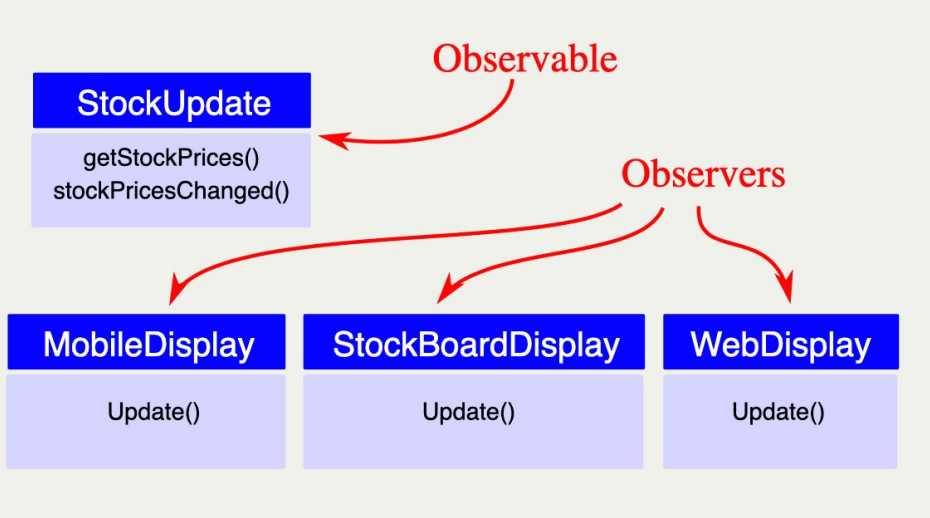
CS 5004, SPRING 2022 – LECTURE 11

OBSERVER PATTERN

- The Observer pattern – idea:
  - Object that might change ("observable" or "subject") keeps a list of interested "observers" and notifies them when something happens
  - Observers can react however they like
  - Used in many areas
    - Events and listeners for button clicks
    - Java RMI (remote method invocation): observable acts as server providing remote access for an observer to subscribe to it
- Support in the Java library: interface **`java.util.Observer`** and class **`java.util.Observable`**

OBSERVER PATTERN - EXAMPLE

- Consider monitoring and displaying updates on stock prices
- Observable maintains state that holds stock prices
- Observers await notifications about changes, and display the news



OBSERVER PATTERN WEAKENS THE COUPLING

- What should `StockUpdate` class know about viewers?
- Observer pattern: call an `update()` method with changed data

```
interface PriceObserver {
    void update(PriceInfo priceInfo);
}
```

```
class StockUpdate {
    private PriceInfo priceInfo;
    private List<PriceObserver> observers;

    ... // constructor comes here

    void addObserver(PriceObserver newObs) {
```

OBSERVER PATTERN

- `StockUpdate` is not responsible for viewer creation
- Application (e.g. `Main`) passes viewers to `StockUpdate` as observers
- `StockUpdate` keeps list of `PriceObservers` and notifies them of changes via callback
- Note: `update()` must pass specific information to (unknown) viewers (that otherwise have no idea about stock prices)

NOTE ABOUT PULL VS. PUSH

- Observer pattern implements push models
  - i.e.,if there is a change, `StockUpdate` notifies all of the relevant viewers

- Alternative - pull model: give all viewers access to `StockUpdate`
  - Let viewers extract whatever data they need
  - More flexible, but more tightly coupled

USING JAVA.UTIL.OBSERVABLE

```
public class Course extends Observable
{
    private int courseID;
    private String title;
    private List<Student> students = new ArrayList<>();

    // ... constructor

    public void addStudent(Student student) {
        students.add(student);
        setChanged();
        notifyObservers(student.getName());
    }

    // ... getters
```

USING JAVA.UTIL.OBSERVER

```
public class ElectronicRoster implements Observer
{
    // called when observers are notified
    public void update(Observable o, Object arg) {
        if (o instanceof Course) {
            System.out.println(
                "Course: " + ((Course)o).getTitle()
                + "\nRegistered student " + arg
                + "\nStudent count: " + ((Course)o).size() + "\n"
            );
        }
    }
}
```

REGISTERING AN OBSERVER

- Somewhere in main:

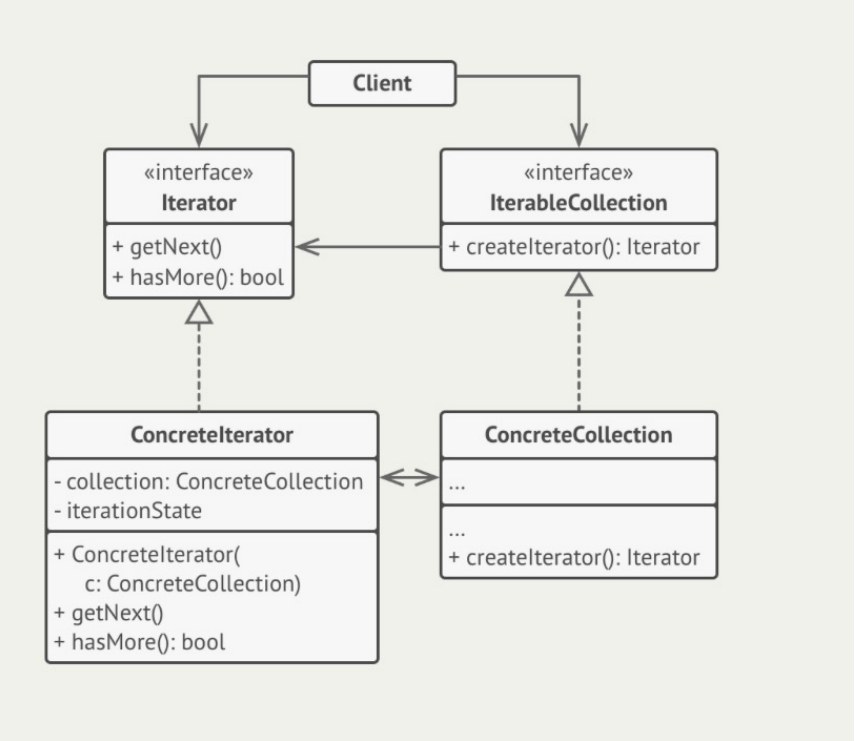
```
Course course = new Course("Underwater Basket Weaving", 151);

course.addStudent(new Student("Hermione Granger", 12345));
// nothing happens

course.addObserver(new ElectronicRoster());
course.addStudent(new Student("Ron Weasley", 67890));
// observer's update method called
```

ITERATOR PATTERN

- We frequently make use of collections or aggregate objects
- An iterator object handles sequential access
- Different iteration algorithms can be handled by different iterators for the same collection

ITERATOR PATTERN

- Iterator structure:
  - `Iterator` interface declares methods necessary to sequentially access collection elements
  - `Iteratable` interface specifies that a collection can create its own iterator
  - Specific traversal algorithm defined in a concrete iterator



ITERATOR PATTERN - EXAMPLE

- Consider a class with an underlying tree structure
- Using the iterator pattern, we can offer both breadth first or depth first traversal without exposing structure directly

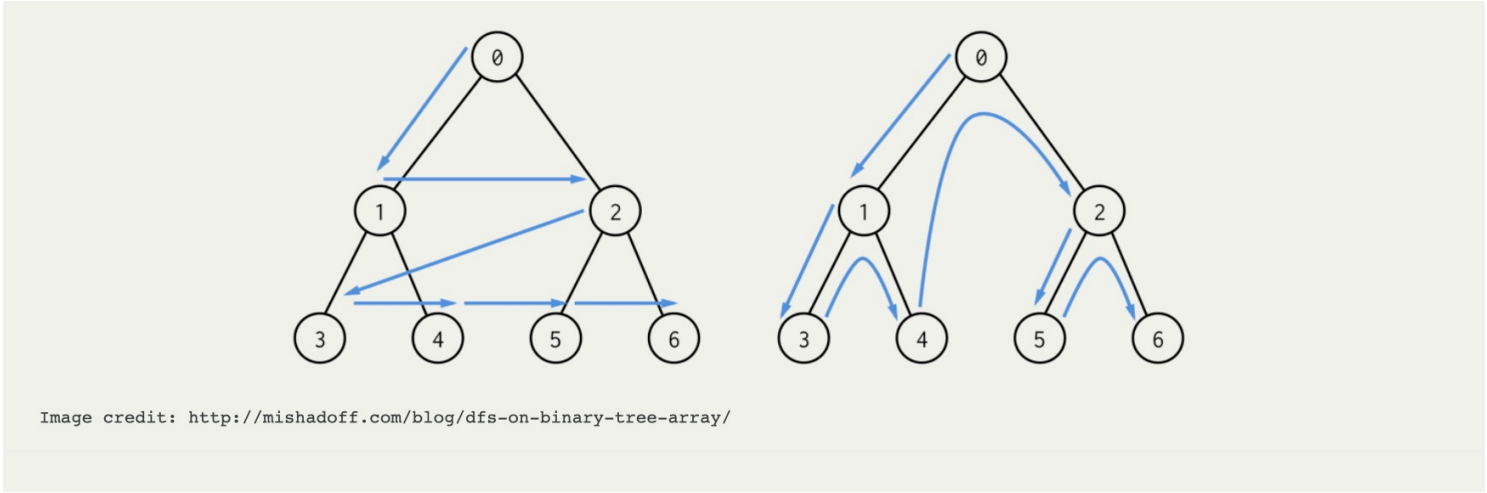


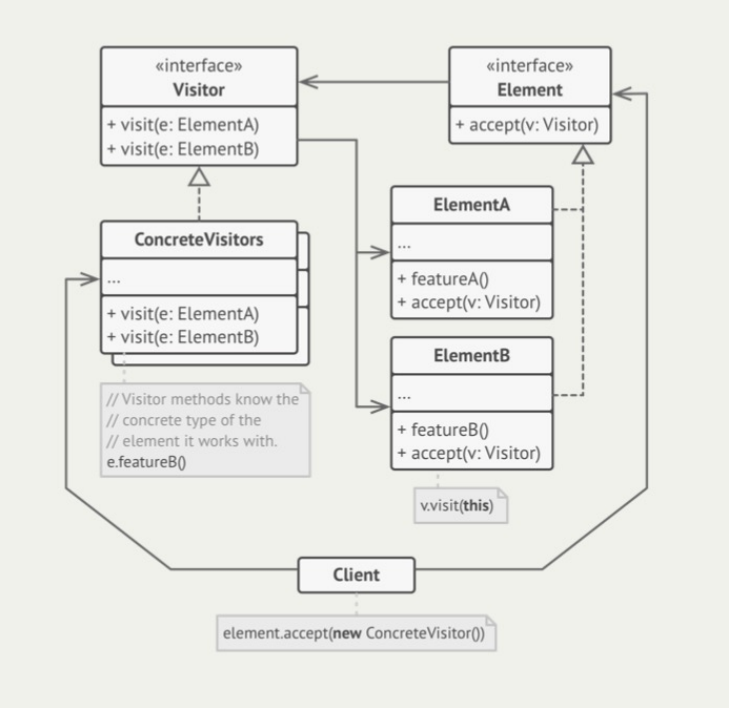
Image credit: http://mishadoff.com/blog/dfs-on-binary-tree-array/

VISITOR PATTERN

- Sometimes we need unrelated, varied objects to provide consistent and appropriate behavior for a specific purpose
- The behavior may not be appropriate or sensible to include in those classes' definitions or interfaces
- Visitor pattern enables us to "visit" objects with personalized functionality as needed

VISITOR PATTERN

- Visitor structure:
  - Element classes accept a visitor
  - A concrete visitor works directly with a specific concrete class (visit method overloaded)
  - Accept method can be called recursively, and/or inherited, passing the visitor structure



VISITOR PATTERN - EXAMPLE

- Consider an existing system made up of various objects and relationships
- You have been tasked with creating an XML documentation export service for the system
- You'd like to have each object in the system provide a toXML() method, but they don't
- Not ideal to be modifying existing production code to write an exporter
- Plus, it's likely you'll need another format (JSON?) soon anyway, and you'll need to do it again
- Export functionality is not really appropriate behavior for these objects. It's specific to your purposes.

MODEL-VIEW-CONTROLLER

CS 5004, SPRING 2022 – LECTURE 11

ACKNOWLEDGEMENT

Notes adapted from Alex Mariakakis with material from Krysta Yousoufian, Kellen Donohue, and James Fogarty. Thank you.

MODEL-VIEW-CONTROLLER (MVC) ARCHITECTURE

- Used for data-driven user applications
- Such apps generally have three main tasks:
  1. Loading/storing data – getting data in/out of some storage
  2. Creating a user interface (UI) – what the user sees
  3. Interpreting user actions – deciding whether to modify the UI or data
- These tasks are largely independent of each other → can be separated into three components: model, view, and controller
  - Note 1: we don't necessarily mean three classes: Model, View, Controller
  - Note 2: each component can be represented as its own package (multiple classes)

MODEL

- Model – typically talks to data source to retrieve and store data
- For example:
  - A database table
  - A file
  - Some external API

VIEW

- View - asks model for data, and presents it in a user-friendly format
- For example:
  - Draws an application window with controls for interacting with the data
  - Takes user input, and sends it to the controller

CONTROLLER

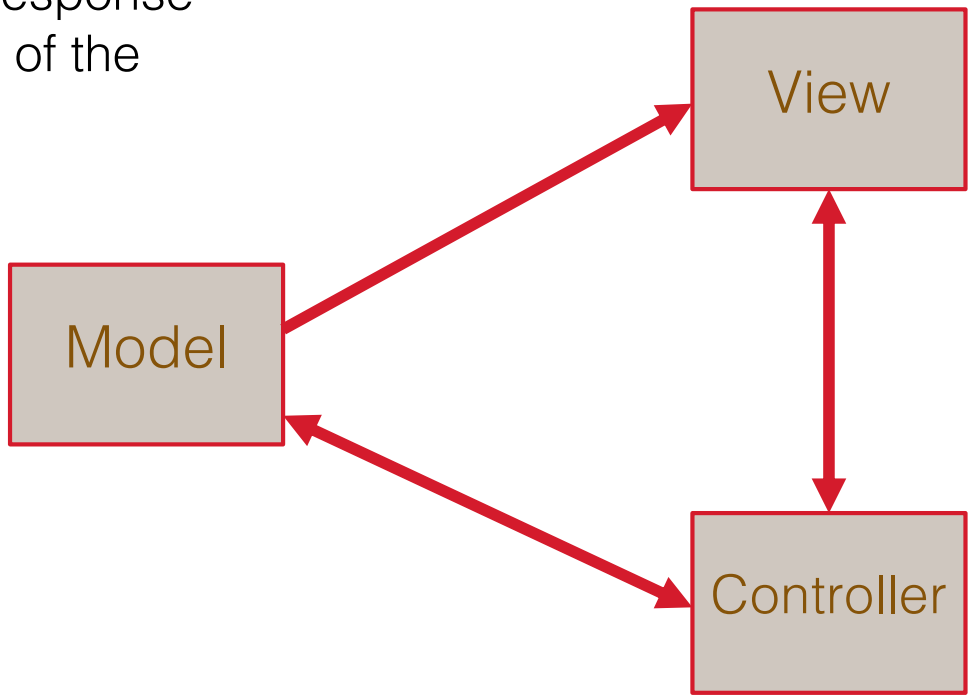
- Controller - listens for the user to change data or state in the UI, notifying the model, or view accordingly
- For example:
  - User clicks “next” button – tell view to load next screen
  - User submits form input – tell model to update the data

BENEFITS OF THE MVC ARCHITECTURE

- Code organization
  - Maintainable, easy to find what you need
- Ease of development
  - Build and test components independently
- Flexibility
  - Swap out views for different presentations of the same data (ex: calendar daily, weekly, or monthly view)
  - Swap out models to change data storage without affecting user

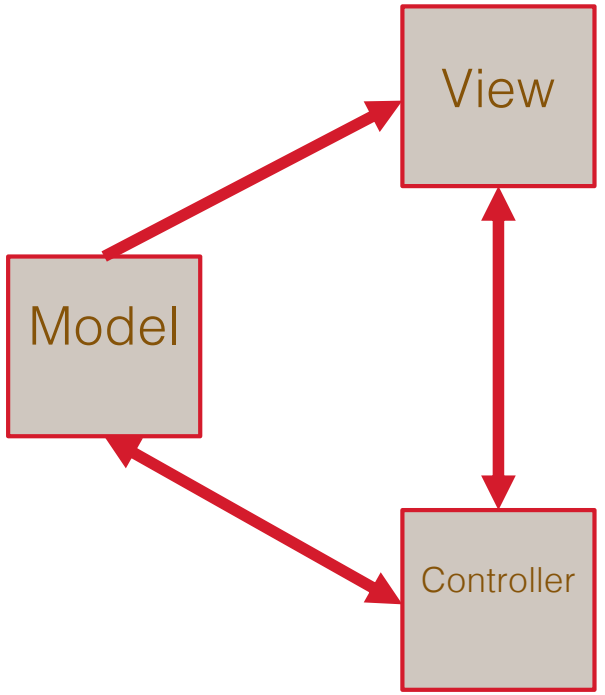
MVC FLOW IN THEORY

- How an application behaves in response to inputs (controller) is independent of the visuals of the UI (view)



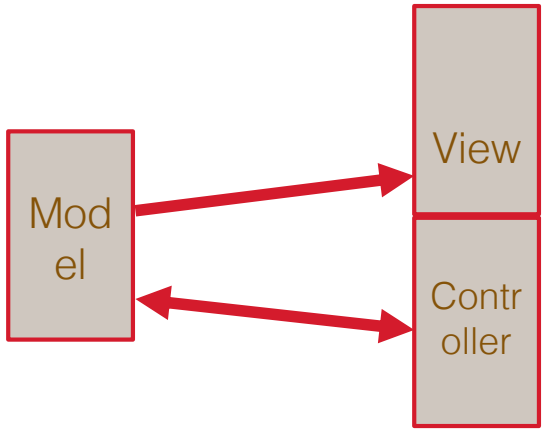
MVC FLOW IN THEORY

- In theory:
  - Pattern of behavior in response to inputs (controller) are independent of visual geometry (view)
  - Controller contacts view to interpret what input events should mean in the context of the view
- In practice:
  - View and controller are so intertwined that they almost always occur in matched pairs (ex: command line interface)
- Many architectures combine the two



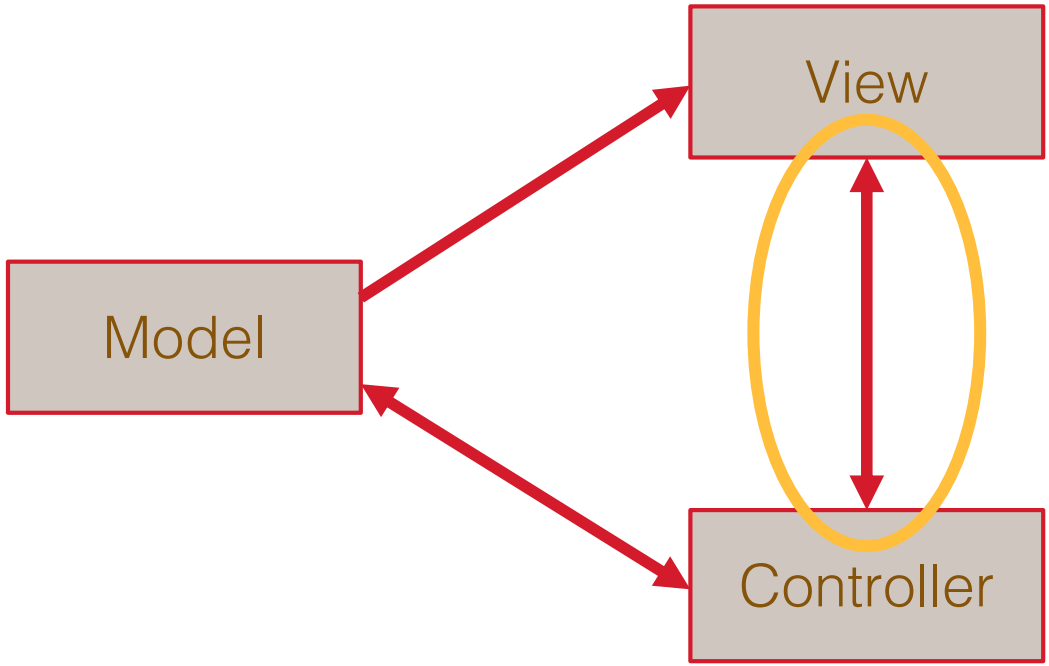
MVC FLOW IN THEORY

- In theory:
  - Pattern of behavior in response to inputs (controller) are independent of visual geometry (view)
  - Controller contacts view to interpret what input events should mean in the context of the view
- In practice:
  - View and controller are so intertwined that they almost always occur in matched pairs (ex: command line interface)
- Many architectures combine the two



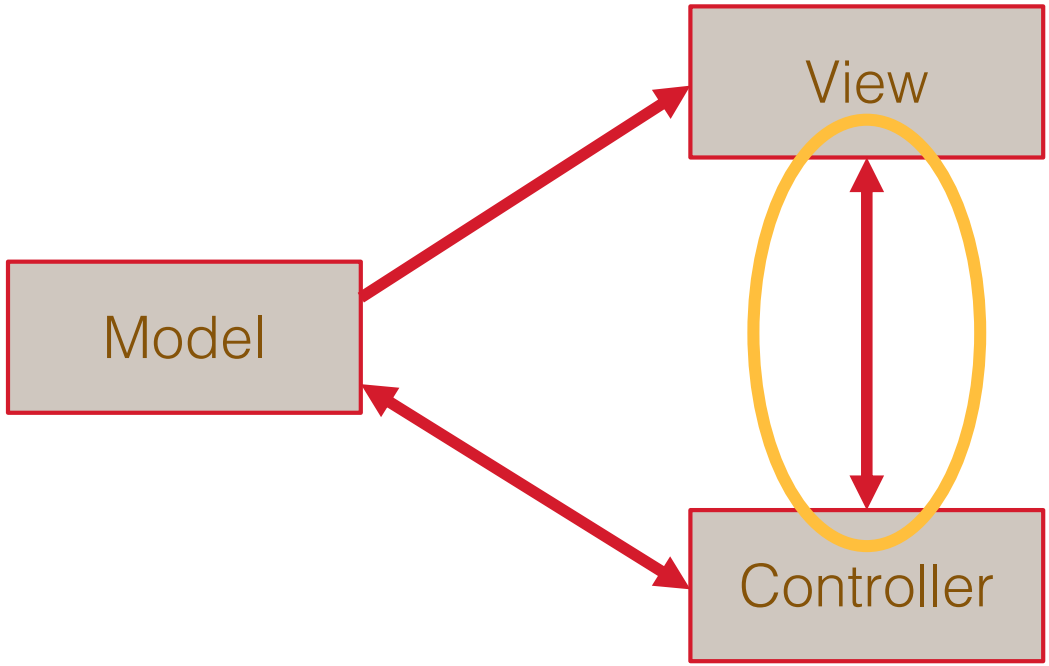
PUSH VS. PULL

- Push architecture:
- As soon as model changes, views are notified
- Guaranteed to have the latest data in the view
- Examples:
  - Some email inboxes
  - Live updating of sport scores



PUSH VS. PULL

- Pull architecture:
  - When a view needs to be updated, it asks the model for new data
    - Avoid unnecessary updates, not nearly as intensive on the view
  - Examples:
    - Some email inboxes



MVC EXAMPLE – TRAFFIC SIGNAL

MVC EXAMPLE – TRAFFIC SIGNAL

Component	Model	View	Controller
Detect cars waiting to enter intersection			X
Traffic lights to direct car traffic		X	
Decide to change the light's status	X		
Manual override for particular lights			X
Detect pedestrians waiting to cross			X
Pedestrian signals to direct pedestrians		X	
External timer which triggers changes at set interval			X

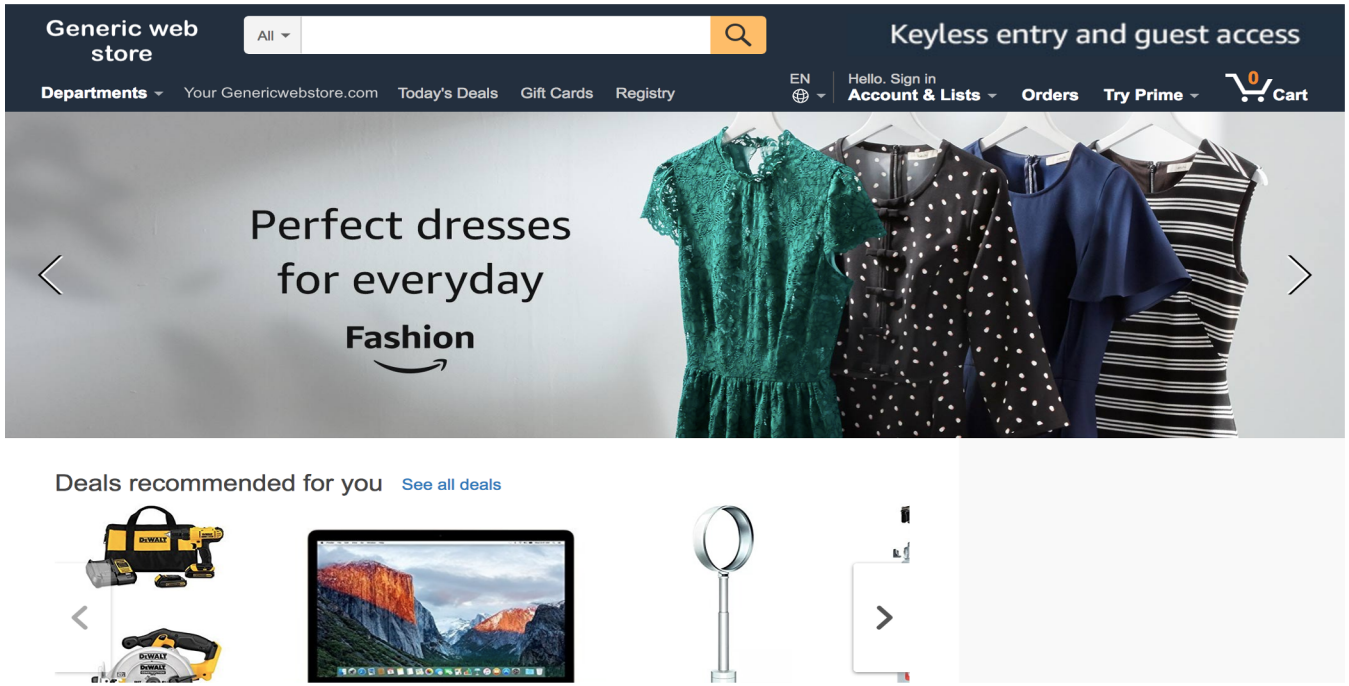
MVC EXAMPLE – TRAFFIC SIGNAL

- Model
  - Stores current state of traffic flow
    - Knows current direction of traffic
    - Capable of skipping a light cycle
  - Stores whether there are cars and/or pedestrians waiting
- View
  - Conveys information to cars and pedestrians in a specific direction
- Controller
  - Aware of model's current direction
  - Triggers methods to notify model that state should change

MVC EXAMPLE – TRAFFIC SIGNAL

- **Model**
  - `TrafficModel` – keeps track of which lights should be on and off
- **View**
  - `CarLight` – shows relevant state of `TrafficModel` to cars
  - `PedestrianLight` – shows relevant state of `TrafficModel` to pedestrians
- **Controller**
  - `PedestrianButton` – notifies `TrafficModel` that there is a pedestrian waiting
  - `CarDetector` – notifies `TrafficModel` that there is a car waiting
  - `LightSwitch` – enables or disables the light
  - `Timer` – regulates time in some way, possibly to skip cycles

MVC EXAMPLE – ONLINE STORE



MVC EXAMPLE – ONLINE STORE

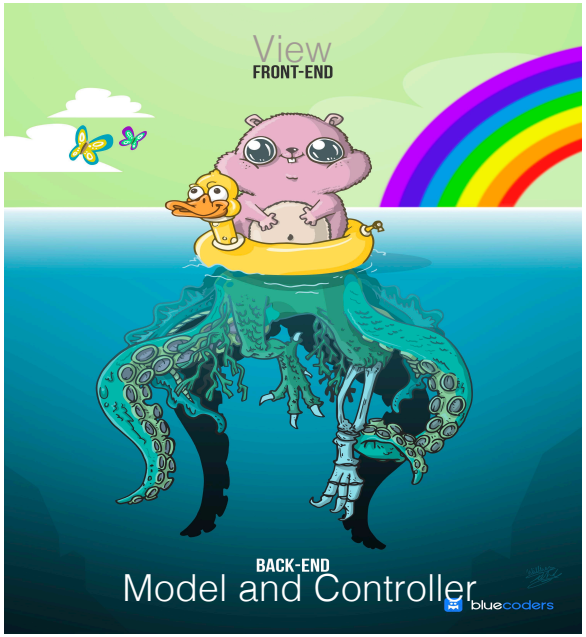
Component	Model	View	Controller
Update user's shopping cart			
Display price/details of a product			
Storage of product/inventory details			
Purchase items in shopping cart			
Record of customer transactions			
User sign-in			
Authenticate user sign-in attempt			
Check user credentials			

MVC EXAMPLE – ONLINE STORE

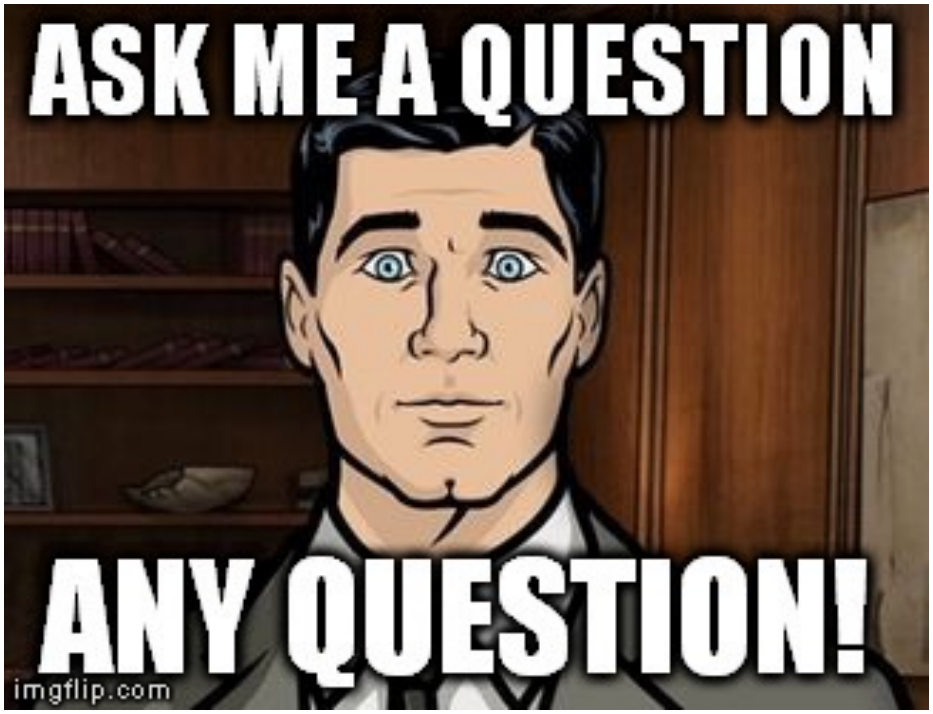
Component	Model	View	Controller
Update user's shopping cart			x
Display price/details of a product		x	
Storage of product/inventory details	x		
Purchase items in shopping cart			x
Record of customer transactions	x		
User sign-in		x	
Authenticate user sign-in attempt			x
Check user credentials	x		

MVC SUMMARY

- Please don't do this (you know better ☺)



YOUR QUESTIONS



[Meme credit: imgflip.com]