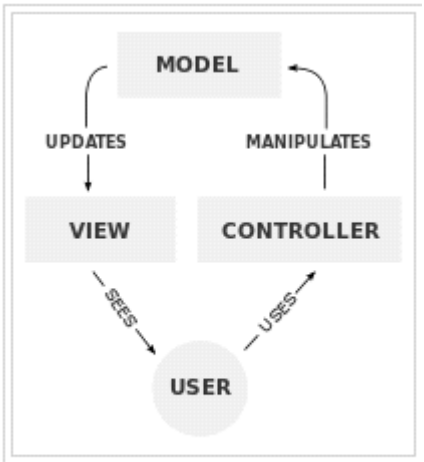
MVC Structure

March 27, 2015 2:21 PM

- MVC stands for model-view-controller
  - A software architectural pattern for implementing user interfaces
    - User interface is the space where humans and machines interact
- MVC divides an application into three connected parts:
  - 

    A typical collaboration of the MVC components

- A controller sends commands to the model to update the model's state
  - e.g. editing a document
  - It can also send commands to its associated view to change the view's presentation of the model
    - e.g. scrolled down a document, changing font size and etc..

- A model notifies its associated views and controllers when there has a been a change in its state
  - This allows the views to show the updated output
  - Allows controllers to change the available set of commands

- A view requests information from the model to generate a representation of it to the user

- MVC is divided between the client and the server
  - In order to reduce computation stress on the server
- Now MVC components execute partly on the client
  - e.g. AJAX

Security Practices for Web Applications

March 27, 2015 2:33 PM

General Security Recommendations:

- Back up often and keep them *physically* secure
- Keep the Web server *physically* secure
  - Prevents people from turning it off, or stealing it
- Apparently NTFS > FAT32 in terms of security
  - Research which file systems are best for security
- Secure Web server and computers on the same network
  - Use strong passwords
- Close unused ports and turn off unused services
  - Make you less vulnerable from a diversity of attacks

Run Applications with Least Privileges:

- Don't run applications as a superuser/administrator
- Run the app in the context of the user to enforce isolation

Guard Against Malicious User Input:

- Filter user input to check for HTML tags, SQL commands and etc.
- Before displaying untrusted information encode it to a string
  - Prevents scripts from running
- Never store unfiltered user input in a database
  - Seriously. Don't do it.
- Do not assume what you get from the HTTP request header is safe
  - Use safeguards query strings, cookies and etc.
  - Be aware user agent information from the browser could be spoofed
- Do not store important information in cookies or hidden fields

Create Safe Error Messages:

- Don't echo information that the database/server shows for debugging purposes
  - Users can use this information for malicious attacks (e.g. echoing a username)
- Use custom error messages for common error
  - Don't let the user know things they wouldn't need to

Keep Sensitive Information Safely:

- Use security certificates, and encrypt everything that is important

Use Cookies Securely:

- Don't store sensitive information in them
- Set expiration dates on cookies
- Encrypt some information in the cookie that isn't useful for the user

Guard against Denial-of-Service (DOS) Threats:

- Use error handling (e.g. try and catch) with a finally block in case of failure
- Configure IIS to use process throttling
  - Don't let one application from using a ton of server resources (e.g. CPU time)
- Set size restrictions to user input, database queries, file uploads and etc.

How To Prevent A DDOS (distributed-DOS) Attacks:

- Bandwidth Oversubscription
  - Have more than enough bandwidth to deal with a small attack
- Automated Mitigation
  - Track a pattern for traffic, if current traffic is greatly outside a pattern's range use DDOS mitigation tools to filter out the noise and allow 'legit' traffic through
- Upstream Blackholing:

    - Prevent certain types of traffic from getting through if it isn't needed
- System Hardening:
    - Having enough inodes on the linux system
    - Have the right number of Apache worker threads can help make it hard to take your service down
- Content Delivery Network:
    - Use several data centers across the world
    - Makes it hard to balance the attack to all your servers

HTTP Overview

March 27, 2015 2:57 PM

- HTTP (hypertext transfer protocol)
  - A protocol to exchange and transfer hypertext

- HTTP functions as a request-response protocol in the client-server computing model
- Client submits http request to a server via an IP address usually mapped to by the DNS (domain name system) as a human readable link
  - Server then responds by providing resources such as HTML files and other content
- HTTP/1.1 can reuse a connection multiple times, this reduces latency

- FTP = file transfer protocol

- TCP (transmission control protocol) complements the IP (internet protocol)
  - TCP provides reliable, ordered, and error-checked delivery of a stream data

- With HTTP protocol the browser can send a GET (and cookies) request to the server
  - Server responds with HTML text (and cookies sometimes)

- User connects to network interface card (NIC) which connects it to a local area network (LAN) which connects to the internet service provider (ISP) which connect to larger ISPs (largest ones are connected with fiber-optic lines, satellite links, undersea cables which run across the nation/region)

- IP address is a 32 bit number, split octets with a '.'
  - Static IP address doesn't change often

- Name server translates the server name to an IP address

- Www is the host name
  - Domain name: e.g. google
  - Top-level domain name: e.g. com

- Server uses numbered ports to connect to networks (e.g. internet, LAN, FTP, etc…)

- Protocol is the pre-defined way that someone who wants to use a service talks with the service
  - e.g. daytime protocol on port 13 tells you date and time and then closes the connection

SOLID Design Principles

March 27, 2015 4:32 PM

Single responsibility
Open-closed
Liskov substitution
Interface segregation
Dependancy inversion

Single Responsibility:

- Dictates that a class should have only one reason to change
- A class should have exactly one job
- A class should have a single responsibility
    - If too many extract some various actions and make them into their own classes

Open/Closed:

- Software entities should be open for extension but closed for modification
- Should be simple to change the behaviour for an entity (e.g. class)
- Closed for modification means we want to change behaviour of a class with minimal modification to the original source code
- When you're type checking often you probably broke the open/closed principle
- Want to:
    - Separate extensible behaviour behind an interface and flip the dependencies
        - Use an interface, and get rid of the type checks and use the interface's function instead
- Polymorphism: the ability to have different behaviour while sharing a common interface

Liskov Substitution:

- Objects in a program should be replaceable with instances of their subtypes without altering the correctness of the program
- Let q(x) be a property about objects x of type T.
    - Then q(y) should be provable for objects y of type S where S $\subset T$
- Derived classes must be substitutable for their base class
    - A subclass should be substitutable everywhere a parent class can be used
    - Preconditions for a subclass should not be greater than its parent
    - Return values for similar functions should be the same
- Any implementation of an abstraction or interface should be substitutable where the abstraction is accepted
- When checking the types of an object most likely you're breaking the open/closed principle and Liskov substitution

1. Signature must match
2. Preconditions can't be greater
3. Post conditions at least equal to
4. Exception types must match

Interface Segregation:

- Many client-specific interfaces are better than one general-purpose interface
- A client should never be forced to implement interfaces that it doesn't use
- Want to improve design and reduced coupling

Dependency Inversion:

- One should "Depend upon Abstractions, not depend on concretions"
- Not dependency *inversion*

- High level modules should not depend upon low level modules
    - Instead depend on abstractions, not concretions
- All about decoupling the code
    - Coupled code is hard to change
- High level code isn't concerned about the details
    - Low level code does care specifics
    - Both low and high level modules should depend on abstractions
- One class should never be forced to depend upon a specific implementation
    - Instead depends upon a contract/abstraction/interfaces
- High level code should depend on abstraction (not low level code), low level also depends on abstraction
    - This decouples the code while still providing modules to interface

Design Patterns

March 29, 2015 12:38 PM

- Procedural code gets information and then makes decisions
- Object-Oriented code tells objects to do things

The Decorator Pattern:

- Allows us to dynamically extend the behavior of a particular object at runtime
    - Without needing to resort to unnecessary inheritance
- Use an interface for the common function
- Then use a decorator which uses a constructor injection
- Then wrap the core service with any number of decorators
- e.g. echo (new TireRotation(new OilChange(new BasicInspection())))->getCost()
    - Where BasicInspection is the interface with the method getCost()
    - Remember to use protected variables
- Be very carefully with inheritance, don't break SOLID principles

The Observer Pattern:

- Want to create a one-to-many relationship so that when one object changes its dependencies/observers are immediately notified
    - Then they can respond to the event that just occurred
- Don't want to just dependency injection; don't want objects to have awareness of one another
    - This makes it more flexible and follows the single responsibility principle
- A change in one object needs to have a nice flexible decoupled way of notifying other objects of this change
    - They in turn can respond however they need to
- Gives use a way for objects to notify one another without being intrinsically linked

The Adapter Pattern:

- An adapter allows you to translate one interface for use with another
- Wraps one interface for use with another
- Think creating wrappers to add some functionality or transitively apply a different contract/interface

The Template Method Pattern:

- A behavioural design pattern that defines the program skeleton of an algorithm in a method called template method, which defers some steps to subclasses
- You can tell when you need a template method when you're copying and pasting multiple classes that are heavily related and only changing very minute details
- Use this design principle when worried about code duplication
    - Use an abstract class for commonalities
- Also be wary of duplicated algorithms
- Make a parent abstract class this lets all subclasses call the exactly same functions
    - For unique methods use an abstract function so that subclasses must implement their own version of it
- Abstract methods let subclasses fill in the missing information

The Strategy Pattern:

- Defines the family of algorithms, encapsulates each one and makes them interchangeable
    - By leveraging polymorphism we're able to develop loosely coupled applications
        - Lets us swap various components at runtime
- Do this by coding to an interface
    - Create a set of algorithms (multiple ways to execute a specific strategy)
    - Then force them to be interchangeable by making them adhere to an interface

    - Then in the context class specify that you need to use the interface
        - This makes any set of algorithms that use the interface usable for the app

The Chain of Responsibility Pattern:

- Allows us to chain any number of object calls while giving each of these objects to end the execution and handle the request
    - If it can't handle the request it can send your request up the chain to give the next object to have the chance to do the exact same thing
- Client can make a request without knowing specifically how the request is going to be handled
- I.e. set up a linked list of all the classes whose functions need to be called and iterate through and call them
    - But do this with a parent abstract class
    - The function that needs to be called should be an abstract function in the parent class to enforce the subclasses to have the function
    - Have a function in the parent class to set the successor
        - Set the successor at runtime manually

Tell, Don’t Ask:

- It is best to tell your objects what to do rather than asking them for their state
- The client shouldn't be doing any checking, it just creates an object and 'uses' it
- Move the manual algorithm (from the client side) into a class as a function which the client calls
    - Keep moving this method from class to class until it belongs in the appropriate class (the class who is responsible for it)
        - Called intermediary functions
        - Just make a 'wrapper' function
        - e.g. elevator->checkAlarms() is actually elevator->monitor->checkAlarms()