Welcome to Design Patterns!

For syllabus, course specifics, assignments, etc., please see Canvas

What is this class about?

- While this class is called Design Patterns, there are many other items of critical importance that will be discussed
  - Object Oriented Analysis, Design, and Programming will be a centerpiece of class discussion on a daily basis. We'll just say 'OO' to describe in general anything Object Oriented in nature.
  - We will learn about Code Smells. It's a funny sounding term the first couple of times you hear it, but there's nothing funny about Code Smells – they are warning signs that something is amiss with your code.
  - We will also learn basic refactoring principles. Once we can identify code smells, we need to fix them (time and ability permitting). We can use OO principles and Design Patterns to aid in the refactoring process.

What is this class about?

- We will also learn fundamental design principles
  - These will aid us in developing code that is intent-revealing, ‘easy’ to test, ‘easy’ to maintain, ‘easy’ to modify, and much, much more
  - Here are a few examples of design principles we will learn to live by
    - Program to an interface, not an implementation
    - Favor composition over inheritance
    - Encapsulate what varies
    - SOLID: Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation, Dependency Inversion (each of these five principles will be discussed during lecture as we are exposed to patterns and other principles that use them)

What is this class about?

- This class is a ‘lite’ version of Software Engineering
  - We will use a great deal of terminology that is related to fundamental principles of good software engineering, with an eye on object oriented analysis, design, and programming
    - Coupling and cohesion are two terms we will learn early on as a basis for meaningful discussion regarding OO software development
    - We will talk about the basic steps of the Software Life Cycle
  - We think about ‘Programming in the Large’ (PITL)
    - Up until now, must programs you have written are at most 1000 lines of code
    - In this class we will begin thinking about writing code that will be part of large projects/code bases. There could be hundreds of other developers that interact with the code you write.
    - With this in mind, we want to design our classes and methods carefully to avoid unintended interactions and behaviors when others interact with our code

What is this class about?

- We will discuss the importance of building classes such that all instances are always in a valid state
  - Instance fields represent the state of a class object
  - We must ensure those fields always contain proper values – we cannot trust others that consume or work with our code to ‘do the right thing’
- To aid in this understanding we must start with the most fundamental pieces of OO, The Pillars of OO

The Pillars of OO

- While there is some discussion over the true number of pillars, we will assume there are four
  - Abstraction
  - Encapsulation
  - Inheritance
  - Polymorphism
- These concepts represent the foundation of all things object related
  - We must learn these first so we can talk intelligently about all that follows
  - Knowing the pillars and what each is about can be VERY helpful at job interviews (this has been reported by your predecessors who are now in industry)
  - They are so important that the first question on your first quiz will be to name and describe these pillars

The Pillars of OO

- Abstraction
  - This is probably the best pillar to discuss first, as the others arguably follow from this one
  - It is the notion that we have a need to represent some kind of entity in our software solution, but don't want to worry about HOW that entity works on the inside and how it represents data to do the things it needs to do
  - For our intents and purposes, the fundamental unit of abstraction is a class
    - A class is composed of attributes (which represent the state of a class instance) and behaviors (which represent the things a class or class instance can do)
    - From your background, you likely know attributes as instance fields or class level fields and behaviors as the methods of a class
    - Note that a class can also be used to represent the notion of an Abstract Data Type (ADT), but an ADT can be even more abstract than that (it can be represented via an interface). Stay tuned for more!

The Pillars of OO

- Encapsulation
  - This pillar is often considered part of Abstraction, but we'll separate the two
  - It says that a class should contain/encapsulate all things necessary to represent and do whatever that class is about (e.g. a Student class will contain all things necessary – attributes and behaviors – to properly represent the concept of a Student – and no more)
  - Part of the notion of encapsulation is that HOW the behaviors of a class are implemented is not something things outside the class need to know about to work effectively with the class
  - Another notion is that the data of a class should ALWAYS be hidden from the outside world to help maintain valid state for any objects of the class
  - SIDE NOTE: Saying always is a bit extreme on the surface, but this should be your point of view unless you can clearly justify why it is ok for other entities to directly access any data a class contains. One justification might be that the data is final and static and you want the outside world to know about it (e.g. the Math class in the Java API has PI and E as public static final fields)
  - In a nutshell encapsulation simply says "None of your darned business!"

The Pillars of OO

- Inheritance
  - This pillar is made possible due to the previous two (Abstraction and Encapsulation)
  - It allows us to build a new, more specialized version of an existing class
    - The super/parent/base class is a generalization, the sub/child/derived class is a specialization
  - It is characterized as an “is-a” relationship
    - Say it out loud and if it makes some sense you might have an inheritance relationship (e.g. Circle is a Shape or Human is a Mammal, but not Shape is a Dodecahedron)
  - It is a white-box approach to class design
    - In order to properly utilize inheritance you must know about the workings and perhaps the data of the super/parent/base class – this violates encapsulation principles!!
    - If you are going to override a method, you need to know the specifics of that method to ensure you do not break contract with the behavior of that method
    - Any fields declared anything other than private (protected, for example) will allow other classes to directly access those fields. Is this what you want, especially when programming in the large?
  - It allows for code re-use, which is nice, but the fact that it is white box says we need to be VERY careful when we use it. Many times we should instead use composition, which is a black box approach to design

The Pillars of OO

- Polymorphism
  - From the roots of the word it translates as “many forms”
  - Is possible due to inheritance
    - Any sub/child/derived class object “is-a” super/parent/base class object
  - Allows us to have a super/parent/base class reference to any kind of sub/child/derived class object
    - Shape s = new Circle(…);
    - s = new Square(…); is also ok
  - Allows for us to utilize polymorphic behavior
    - Recall behavior is object oriented speak for method
    - Polymorphic behavior says the version of a method that executes is determined at runtime based on the object(s) in memory (there is a dispatch process that is used to figure out the method – this dispatch process can actually break down in some circumstances – the Visitor pattern can help with this problem)
    - Thus polymorphic behavior is enabled through method override in Java
    - Side note: Java automatically supports polymorphic behavior, languages like C++ and C# require additional syntax to formally support it

So What is a Design Pattern?

- It is captured design expertise that solves a certain kind of problem in a software system
- It can be applied to a variety of different scenarios and enhance your software solution
- Design Patterns typically target what are know as Non-Functional Requirements. Some examples are
  - Readability
  - Testability
  - Maintainability
  - Re-usability
- There are many design patterns that are not OO in nature, we aren't here to talk about those
- There are also what are known as Anti Patterns – check Wikipedia for some 'fun' examples of them!

Some Words Regarding Work for the Course

- While assignments, etc. will be posted to Canvas, note that the majority of your work will involve application of patterns we learn to some software scenario.
- The first few assignments will focus on single patterns
- We will also do some refactoring assignments that incorporate patterns and OO principles
- Your final assignment/project will be quite large in scope and expect you to employ the OO and design principles we have discussed throughout the quarter as well as design patterns. The final assignment will allow for collaboration with a fellow student in the class if desired
- There will be two quizzes, a take home midterm (which will be open book and notes, but no ‘Googling’, etc.), and an in class final