Name: Jacob Hallberg
Title: Flash
Project Summary: Flash is a web application that gives Climber's a convenient way to track their climbing progress, rate a route's difficulty/type, and generate Climber tailored climbing sessions. Flash allows Climbers to sign up, login/logout, add personal information, add climbing routes, add route difficulty, add route type, and generate climbing sessions.

Implemented Features:

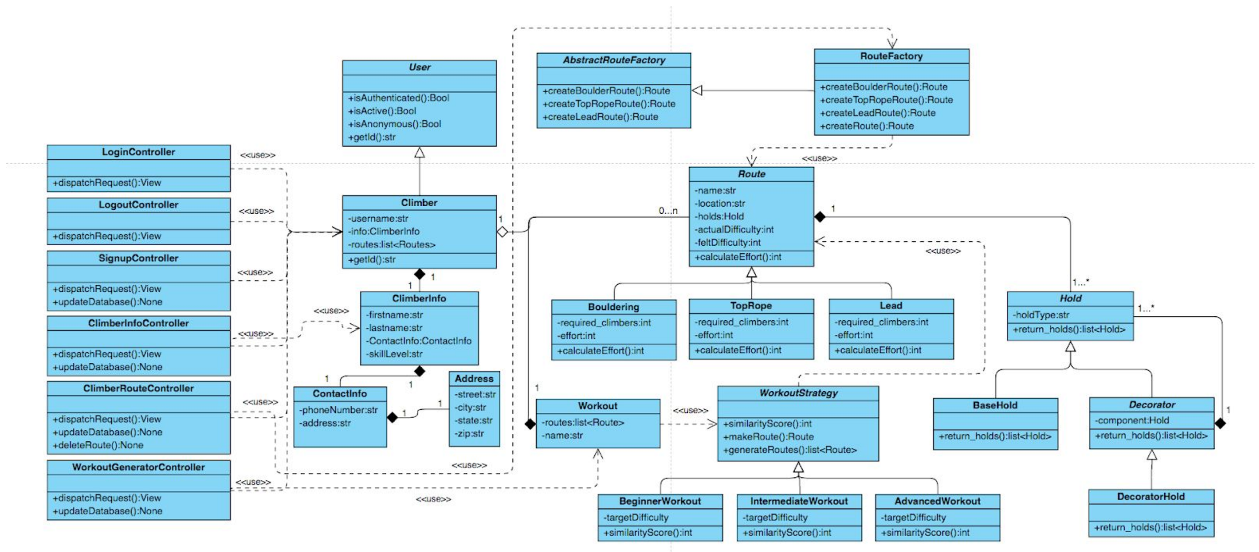
Requirement ID	Actor	Requirement
U01	Climber	Climber can add personal information.
U02	Climber	Climber can add/undo climbing routes.
U03	Climber	Climber can add route difficulty/type to a route.
U04	Climber	Climber can view route data for a specific gym.
U06	Climber	Climber can generate workout.

Not Implemented Features:

U05	Climber	Climber can add/undo their climbing deficiencies.

*It made sense to not add this requirement because it was cumbersome for the user experience. Every time they would want to workout on something else they would have to update their deficiencies. Now, they can just select what they'd like to work on when generating a workout which improves the overall experience.

Class Diagram:

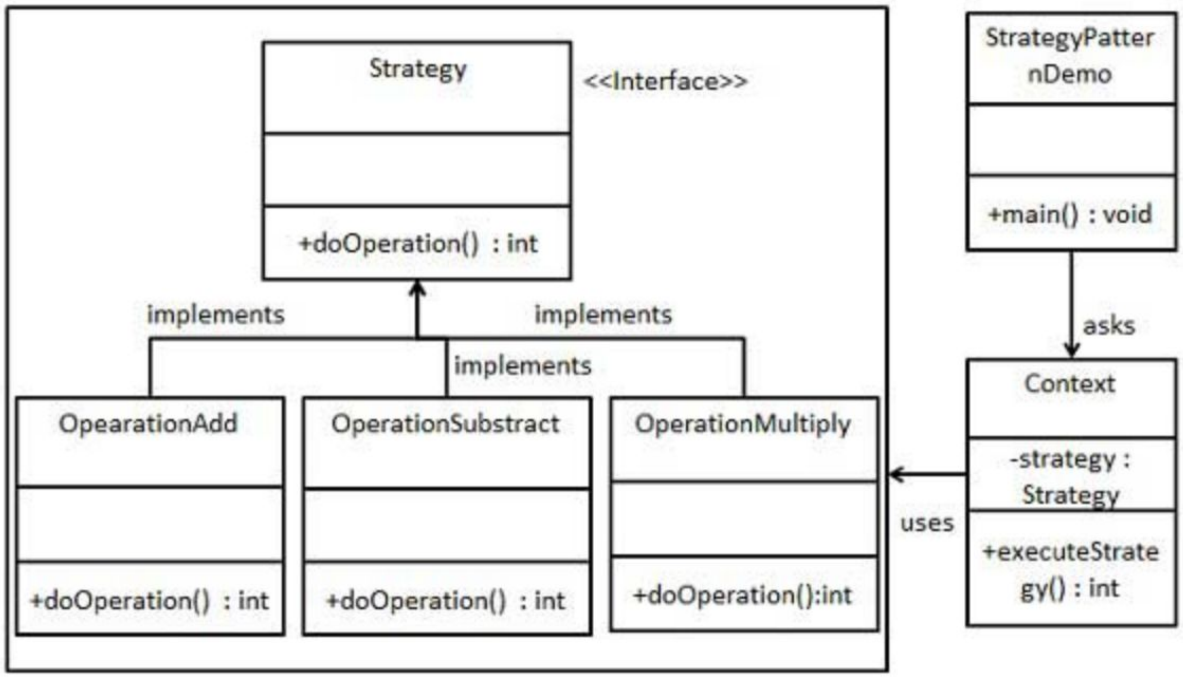


The first class diagram I created had no design patterns in mind, meaning the class diagram was inherently expected to change considering I made use of three design patterns. Compared to the previous diagram, you can see all of the three design patterns I implemented (Strategy, Factory, Decorator). I also added three new controllers for login, logout, and signup. Additionally, I removed the GymInfo class because it didn't enhance the user experience at all, but if I decided to continue to develop the app that would be something I would bring back. I also changed the method names for the controllers because Flask requires a method called dispatchRequest() when defining a view controller. I also removed the getter and setter methods for better readability. Furthermore, I added a User class which Climber inherits from. The user class defines functions that are necessary for Flask's login library. Overall, designing up front helped keep everything in order, even when iterating over the design and adding new design patterns. I don't regret the initial design stage at all.

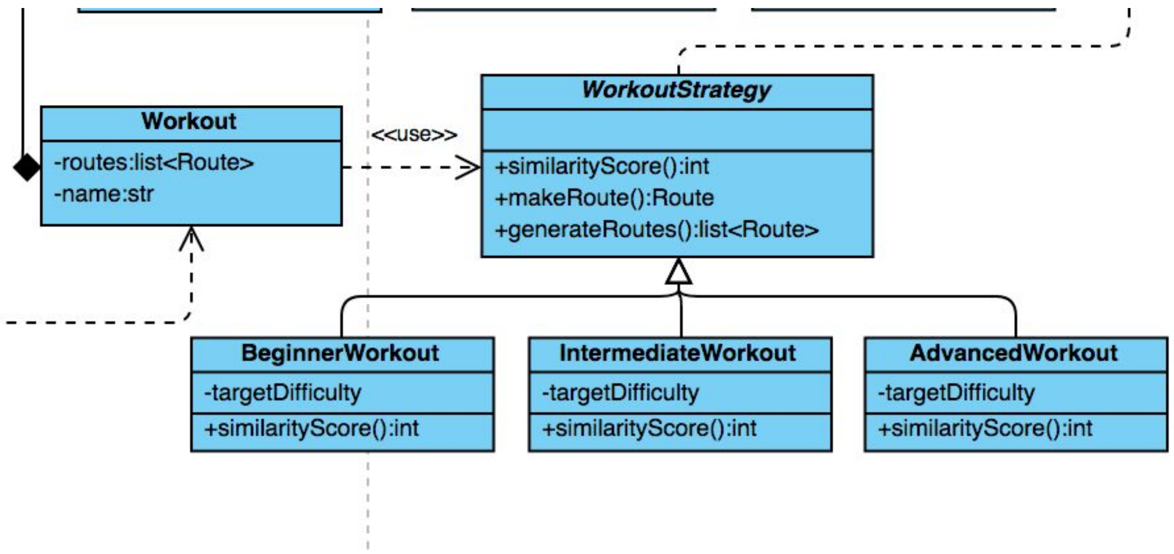
Design Patterns:

Sources: https://sourcemaking.com/design_patterns
https://www.tutorialspoint.com/design_pattern/

1. **Strategy:**

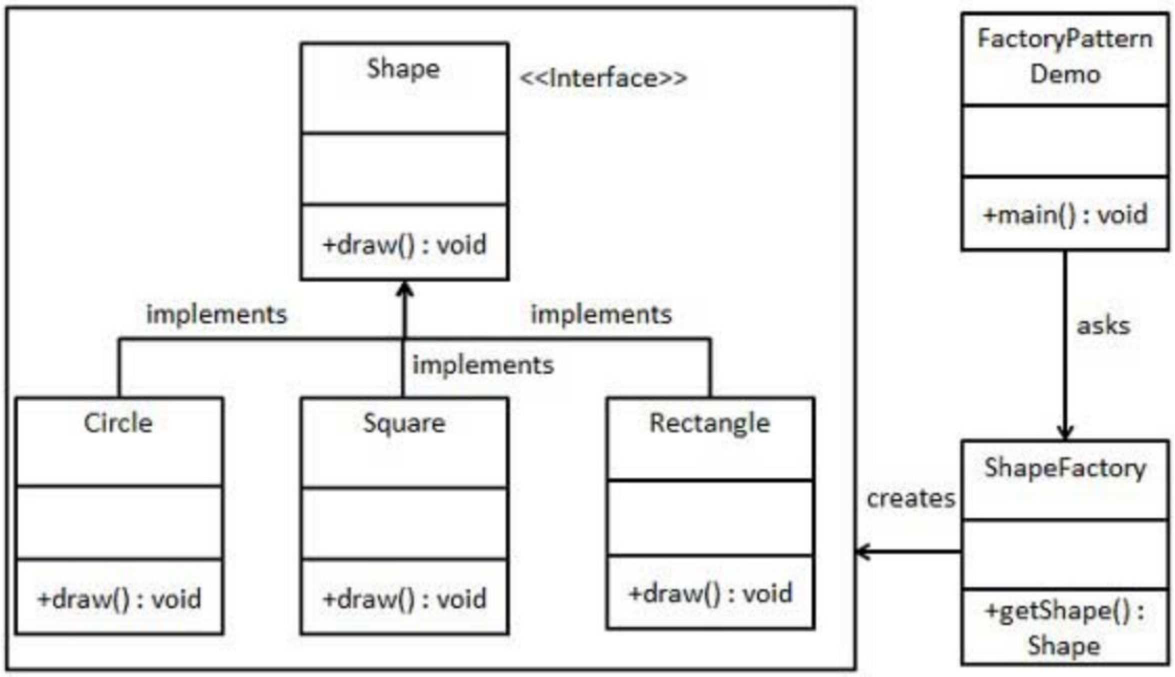


My implementation:

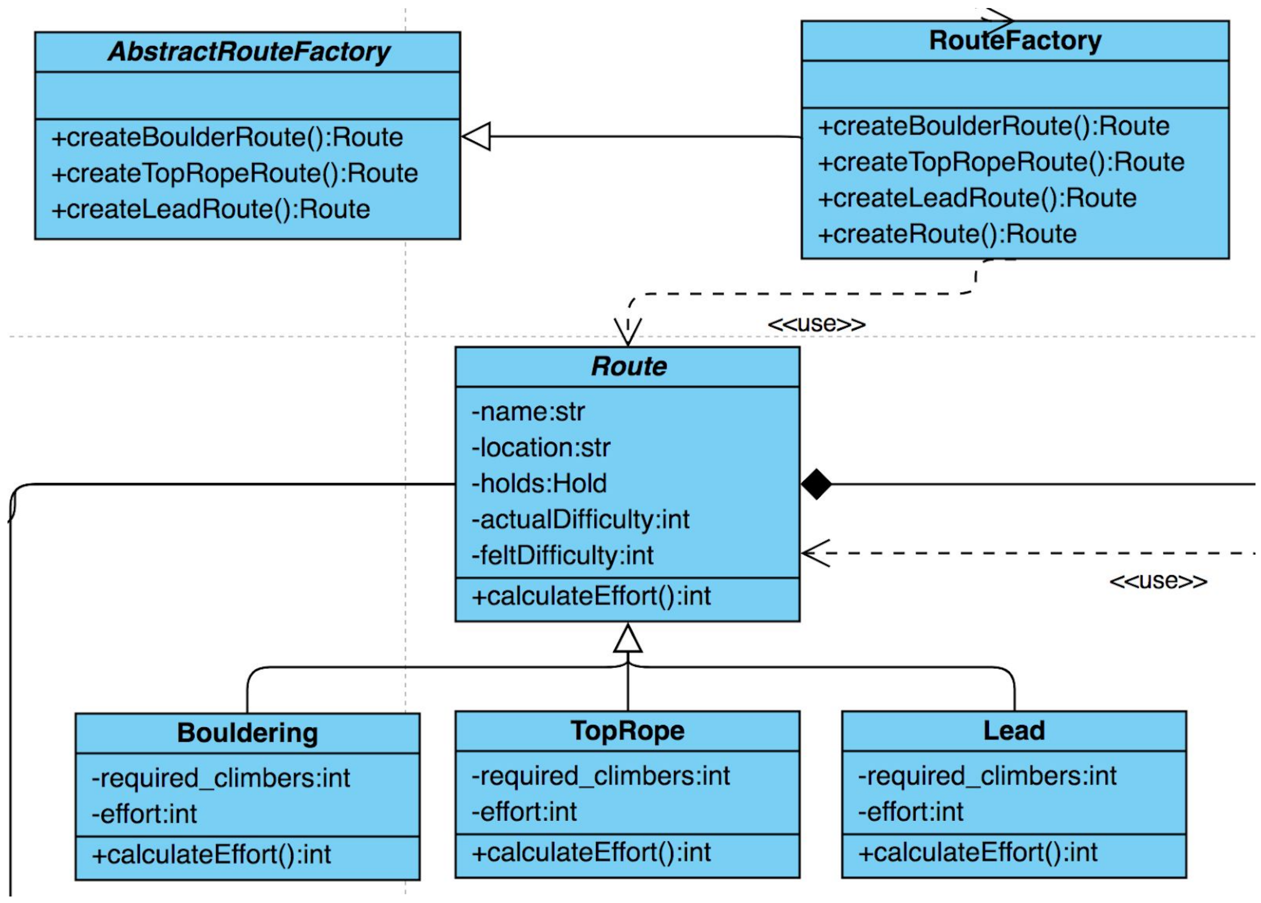


I chose to use the strategy design pattern because during my workout generation algorithm, I needed a way to create different types of workouts depending on the skill level that the user selected. Beginner workouts should be different from intermediate workouts and advanced workouts. To do this, I used strategy to decide when generating a workout, which similarity score algorithm to use within the generateRoutes() method. This enables different workouts to be generated depending on the skill level of the user. I implemented it by making creating an abstract class WorkoutStrategy with an abstract method similarityScore(). When a user requests a generated workout, the controller grabs the skill level from the Climber object and matches the skill level with the corresponding Workout class (Beginner Workout, Intermediate Workout, Advanced Workout). It then creates a Workout object with the Workout algorithm class passed in as a parameter which generates a workout.

2. **Factory:**

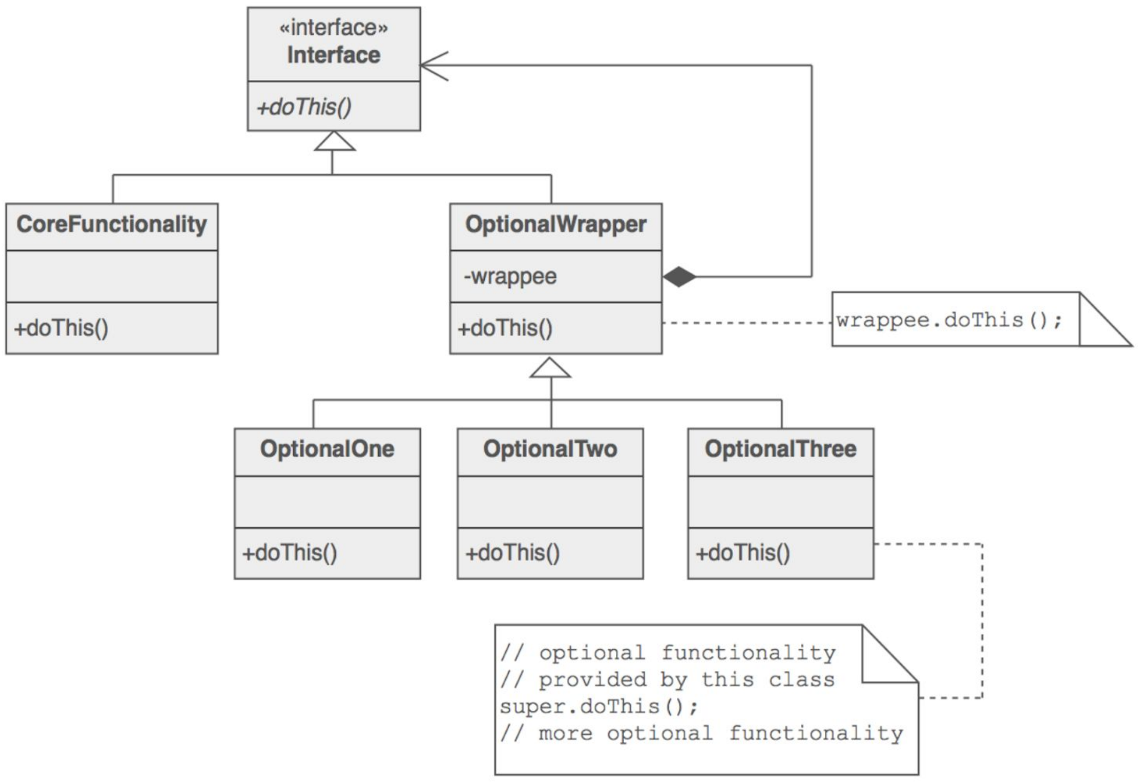


My Implementation:

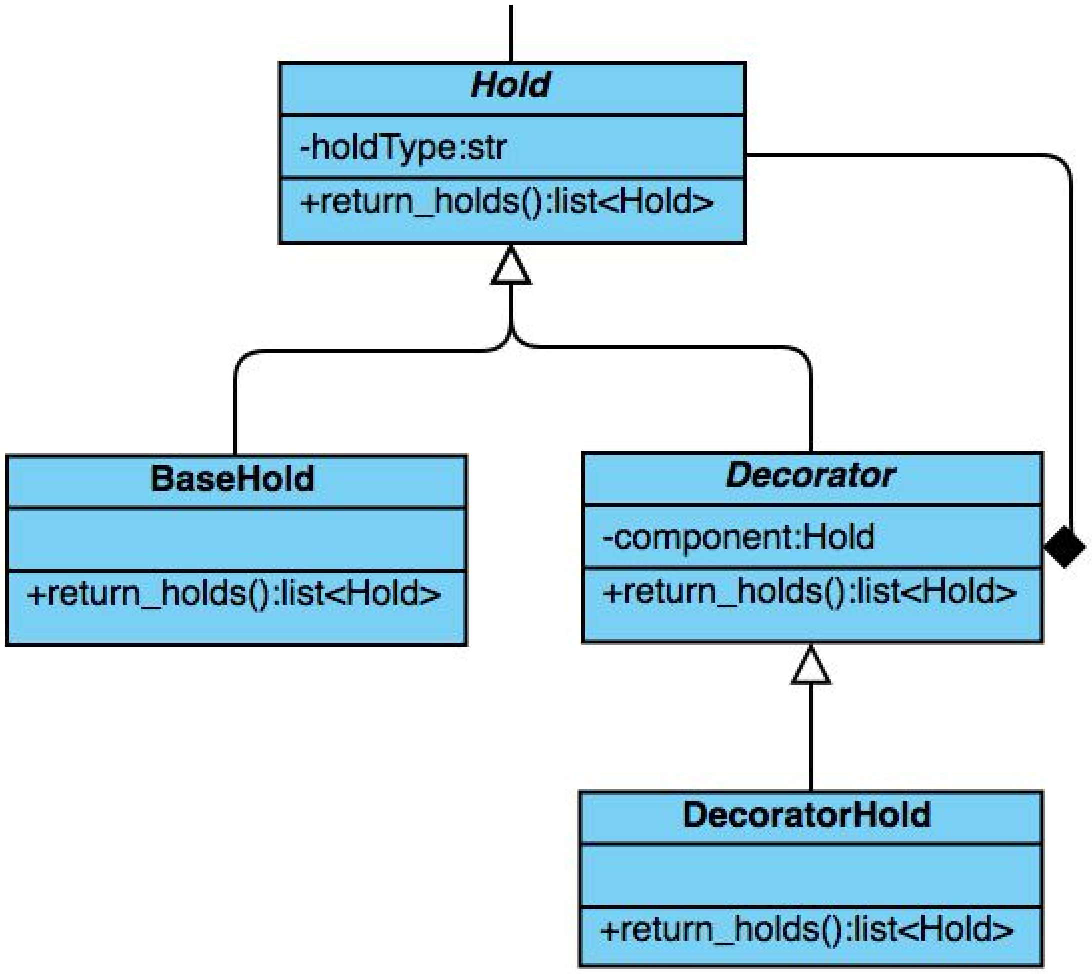


I chose to use the Factory design pattern because the web app constantly is creating route objects and I needed different types of route objects because each route has an unique effort function associated with it used in the workout generation algorithm. To get around this, I deployed the factory design pattern which enables the controller to make one call to the static method createRoute() in RouteFactory(). This abstracts away route creation and makes it very convenient for the client. I designed route factory by creating an AbstractRouteFactory class which is inherited by the RouteFactory class forcing it to implement the individual creation routines. I also implemented a createRoute() concrete routine that takes in a route type as a string and decides which route creation routine it should choose.

3. **Decorator:**


«interface»
Interface
+doThis()
CoreFunctionality
+doThis()
OptionalWrapper
-wrappee
+doThis()
wrappee.doThis();
OptionalOne
+doThis()
OptionalTwo
+doThis()
OptionalThree
+doThis()
// optional functionality
// provided by this class
super.doThis();
// more optional functionality

My Implementation:


Hold
-holdType:str
+return_holds():list<Hold>
BaseHold
+return_holds():list<Hold>
Decorator
-component:Hold
+return_holds():list<Hold>
DecoratorHold
+return_holds():list<Hold>

I chose to use the decorator design pattern because route objects may have many hold objects (Crimpy Hold, Jug Hold, etc.). If I was to build out this application even further, I would add additional functionality to these hold objects which makes the decorator design pattern even more valuable, because with one recursive call, the individual functions of each Hold object will also be called. Currently, the recursive call is building up a list of route types when return_holds() is called. This makes it very convenient to add as many hold objects as I would like to a Route and to easily get the hold types for every hold stored within a route object. To create the decorator design pattern, I created an abstract Hold class that has a variable holdType, and a concrete class BaseHold which inherits Hold. This allows a route to have an individual Hold. However, I also created another abstract class Decorator which inherits from Hold and the Decorator class has a component variable which stores a Hold object. This hold object can be either a BaseHold or DecoratorHold object allowing a single Hold object to contain as many Hold objects as needed.

Things I’ve learned:

I’ve learned how important object oriented design principles can be for code reusability, extendability, and readability. I’ve now realized the scope of OOP and can attest to the importance of these principals when coding and when discussing with others who understand OOP. Even whilst writing my code, I would show my friends my code and they would be able to easily understand it because the abstraction of implementation from the client makes code understandability far better. These pluses are the exact reason why designing with object oriented principles in mind can make the design pattern painless compared to jumping right in to coding. I’ve learned that design is often fundamental to success when dealing with a large codebase. Although small projects (startups) may get away with instant implementation without design, I’ve realized the benefits well thought out designs can have when dealing with hundreds of programmers. I’ve also learned how important iteration can be. My project was small and I still had to change things as I went and each iteration of change influenced functionality down the road. This is why planning and design is integral to success.