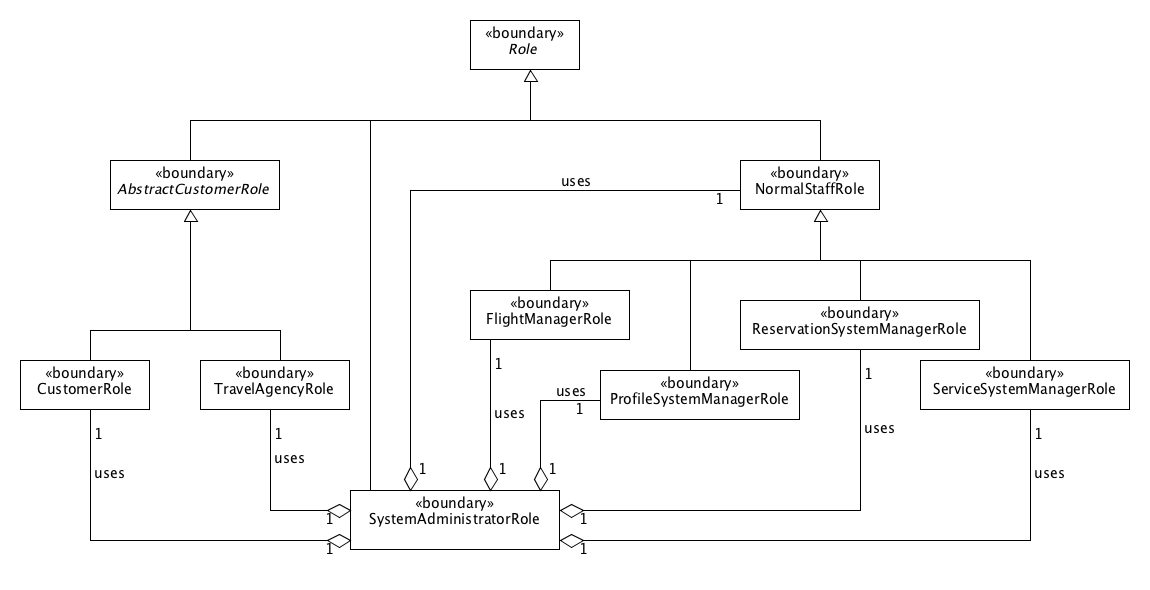
Design Patterns

In the Flight Management System we incorparated a few design patterns into our source code. During the design phase, we realized some problems with our design that could potentially cause issues of code being rewritten in many places. So after consulting some experienced software engineers and developers, they recommended design patterns that could improve our system design. The three design patterns that we incorporated in our implementation are:

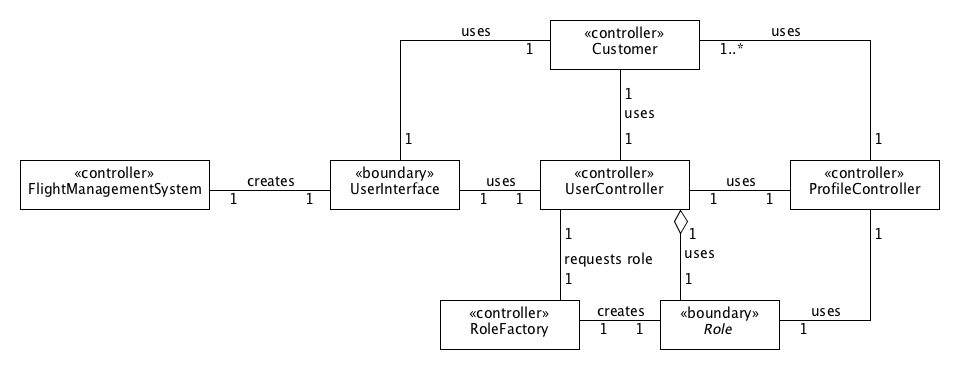
1. Delegation pattern
2. Factory pattern
3. Builder pattern

1. Delegation Pattern

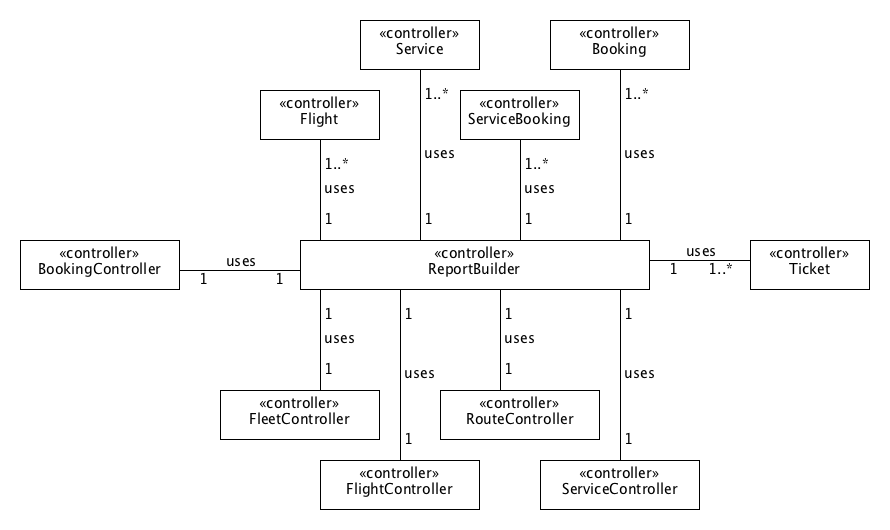
The delegation pattern is where an object uses another object (helper object) to perform a task. The reason we used this pattern is to solve our problem of having classes that require multiple inheritance to work. Therefore, the non-abstract classes in the “role” package are all helper objects. They represent the possible tasks that a particular user of the system is able perform in the system. The SystemAdministratorRole class is the class that uses this pattern. As shown in the class diagram below, all non-abstract classes have an aggregation relation with the SystemAdministratorRole class. 

With this design pattern in place, the System Administrator actor can act as any of the roles that the SystemAdministratorRole class has. Furthermore, this also solved the problem of multiple inheritance in our implementation language, Java.

2. Factory Pattern

The factory pattern is a creational pattern where a class method is called to produce an object. In our implementation, we used this design pattern to create a role based on the classes in the “role” package shown in the class diagram above. Based on the role of the user that was stored in the database, we passed a String representation of the user’s role to the RoleFactory’s class method “getRole” to get an object from the “role” package. Naturally, the code does not want to be concerned about the object it got back from the factory class. So we the “getRole” method always returned a Role reference to the object. So to use the returned object, the overridden methods in the Role class are called to take advantage of the polymorphism that the classes in the “role” package implement. The class diagram below shows how the UserController class requests a Role from the RoleFactory class and gets a Role reference to an object of the “role” package.

3. Builder Pattern

The builder pattern is where a class in instantiated by a client, its methods are called to “build” the desired object piece-by-piece by the client, and finally a “build” method is called to return an object that has been initialized to the building methods that the client called. In our implementation, we used a pseudo-builder pattern. The ReportBuilder class does not return an object but rather builds a report based on the given parameters and displays it. In the class diagram below, the ReportBuilder class uses several controller classes to build a report. The client will call a class method “displayReport” with arguments and based on the given arguments, the ReportBuilder class will generate the appropriate report.  The reason why we used this design pattern is to make sure our system has high maintainability and evolvability. With this in mind, this design allows the flexibility in adding more report types as the Flight Management System is evolved to the next version.