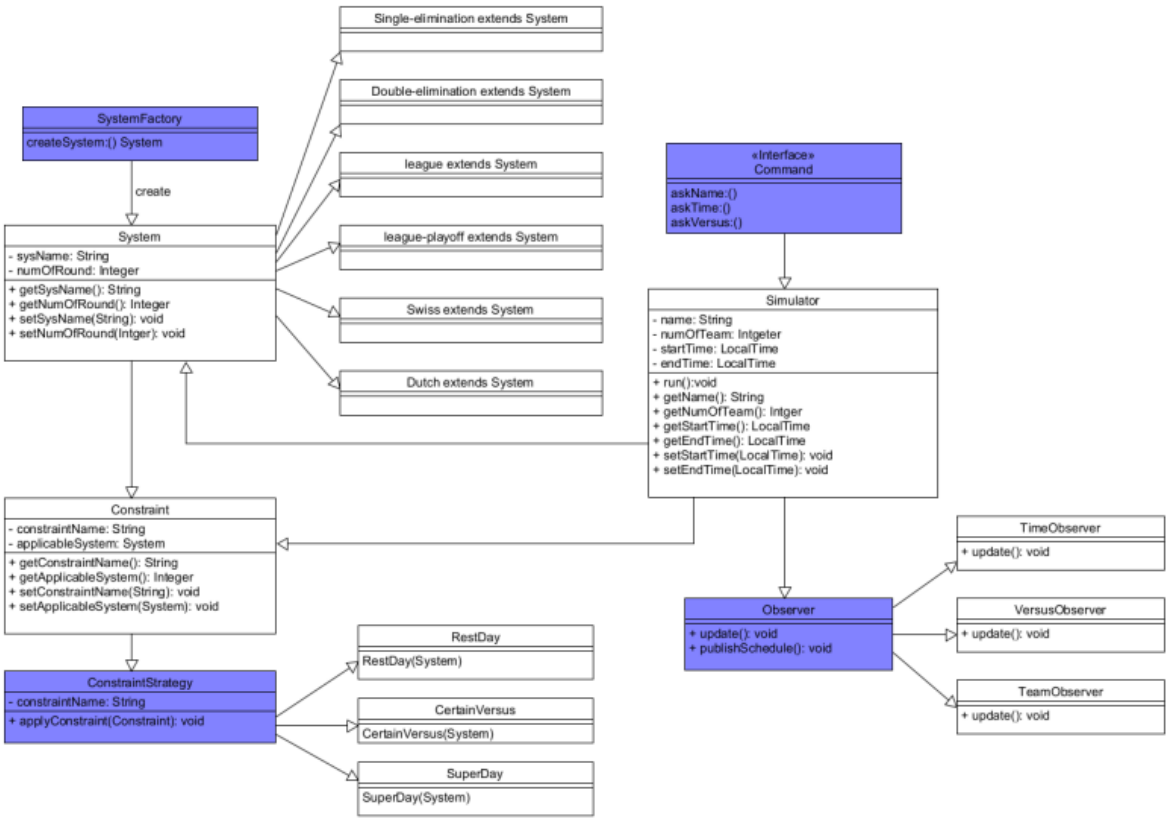
Race Schedule Simulator

Haolin Yang & Dawson Voth

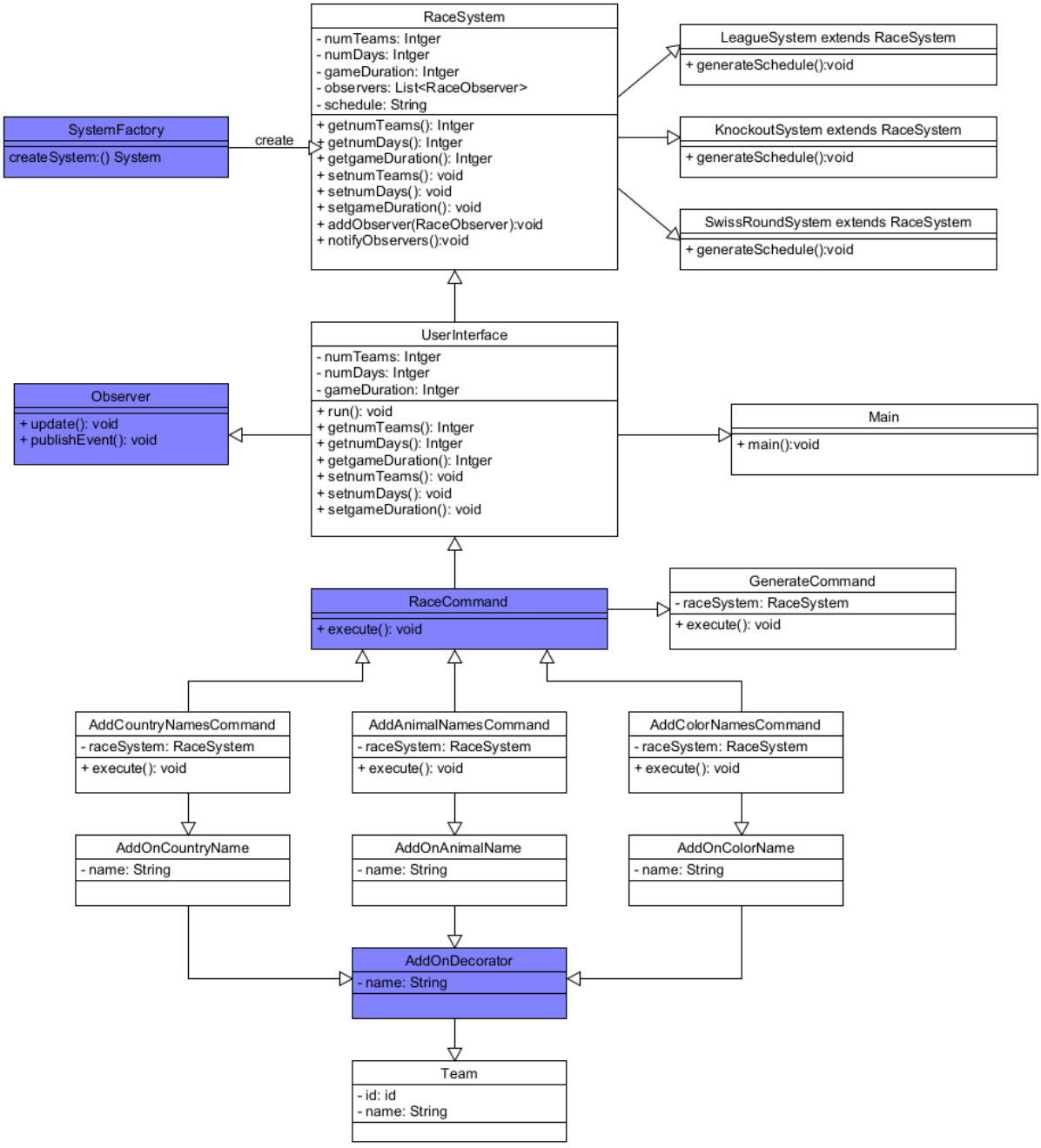
Final State:

The feature we implement is generating a race schedule based on the number of participating teams, the total number of days allowed for the game, and the duration of a single game that uses input. We ended up dropping the feature to generate schedules for existing common events like NBA, NFL because there is no existing data for them. We also modified the constraint from break day, super day with strategy pattern to random countries, animals and colors with decorator pattern because strategy pattern is not very suitable for constraints. Now the decorator pattern is used to randomly add countries, animals, and colors to team names.

Diagram in project 5:



Final Class Diagram:



We implemented constraints using the Decorator pattern instead of Strategy Pattern. We take the process of simulating the race schedule as a function in RaceSystem instead of a separate class. We create a class for the User Interface and use the Command pattern for it.

Third-Party code vs. Original code:

Command Pattern: Implemention of UserInterface is original code; Some frame code from Command Pattern.pdf (csulb.edu)
Factory Pattern: Algorithm implementation of competition system is original code;
Some frame code from Java Factory Pattern Explained - HowToDoInJava
Observer Pattern: Some frame code from Observer Pattern - Javatpoint
Decorator Pattern: Original code pattern. The lists for the country names, animals, and colors used to change the name of the Team were generated by ChatGPT.

Statement on the OOAD process:

1. We have found that judicious use of **static variables** can lead to more efficient code. Many times we only need to use static variables to omit the operation of updating variables at the end of each function. If static variables are not used, sometimes we need to store data in some external data structures (like global variables), which may cause some unforeseen errors.
2. Using **Factory Pattern** is more reasonable than creating subclasses one by one. This not only makes it easy to introduce new types of objects or change the way objects are created, but also encapsulates objects more tightly, simplifying client code.
3. When implementing the tracker part of the **Observer Pattern** during one of the homeworks it became difficult to make sure it was the correct output because multiple objects were using the same instance. This was because we implemented it with a Singleton Pattern so that it could be easily accessed anywhere in the program. It ended up working out well once check safes were added, but was tricky to get right.