Project: 08 - OO Poker

Team: Nels Anderson
Ehsan Karimi
Cary Sullivan
Justin Visher

1. Features Implemented

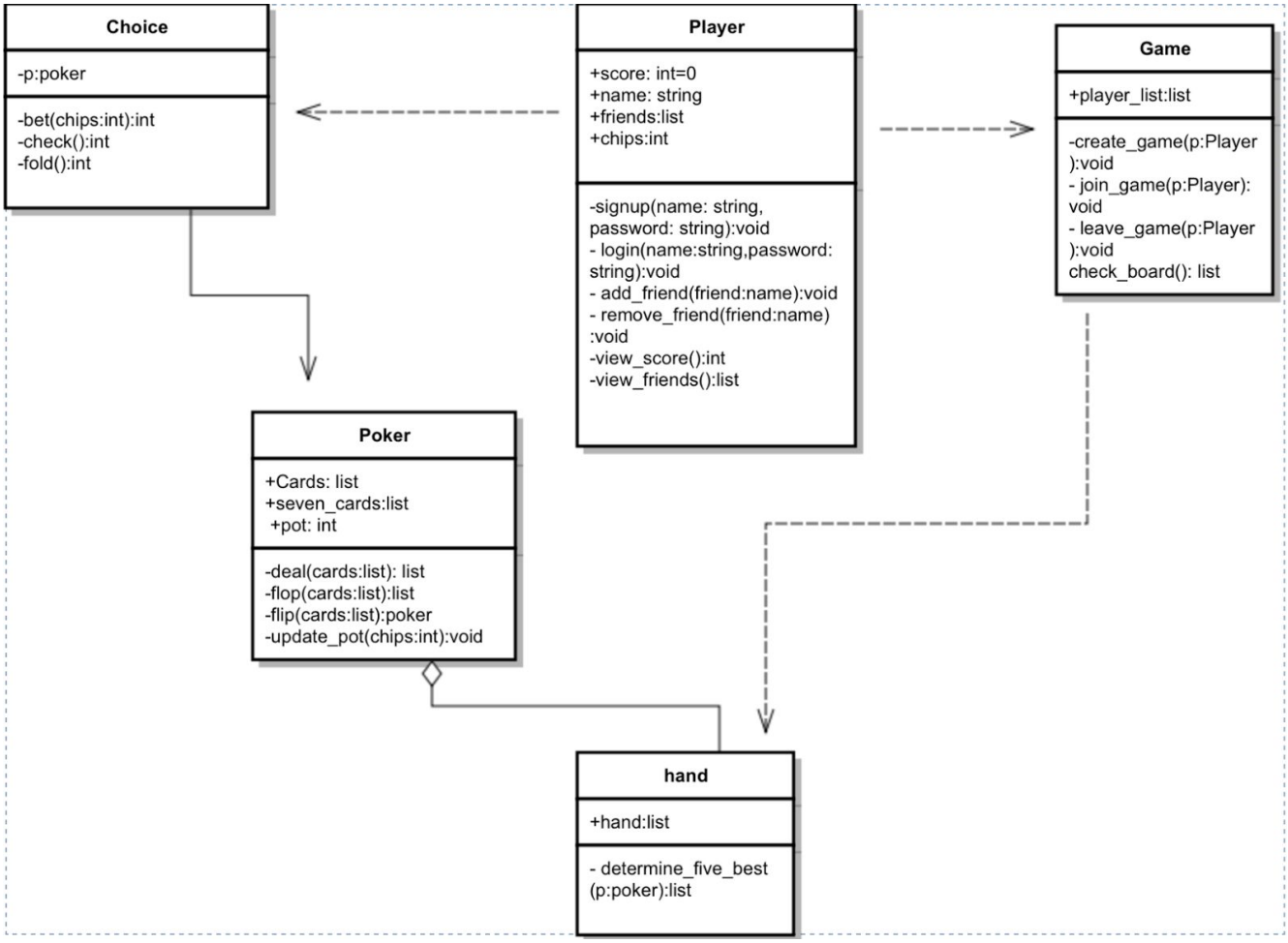
ID	Title
UR-01	Sign Up
UR-02	Sign In
UR-05	Join Game
UR-06	Leave Game
UR-09	Bet
UR-10	Check
UR-11	Fold

2. Features Not Implemented

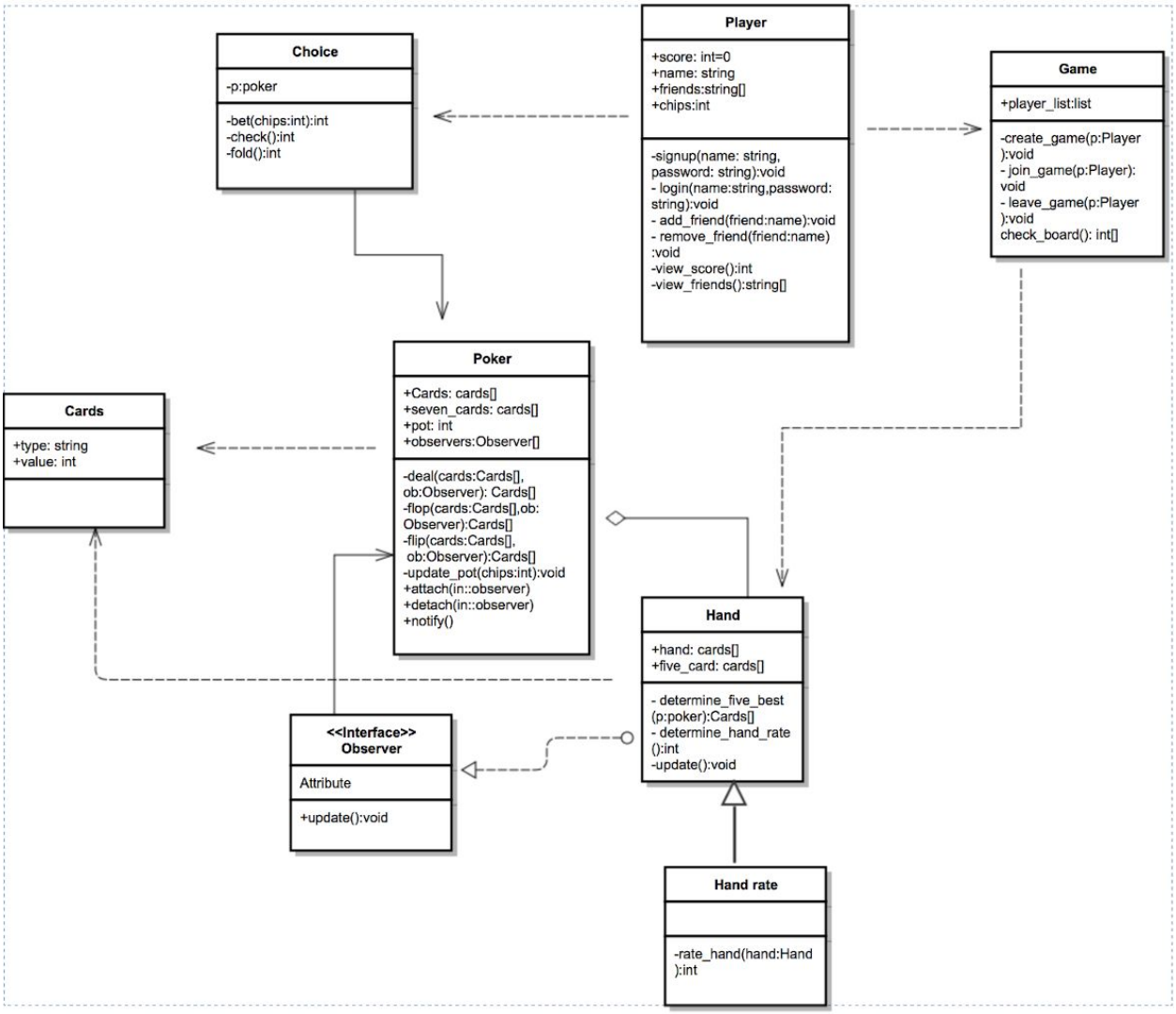
ID	Title
UR-03	Add friends
UR-04	Invite friends
UR-07	Keep track of score
UR-08	Receive chips an hour after running out
UR-12	See other players' names

3. Class Diagram Comparison

Part 2 Class Diagram



Final Class Diagram

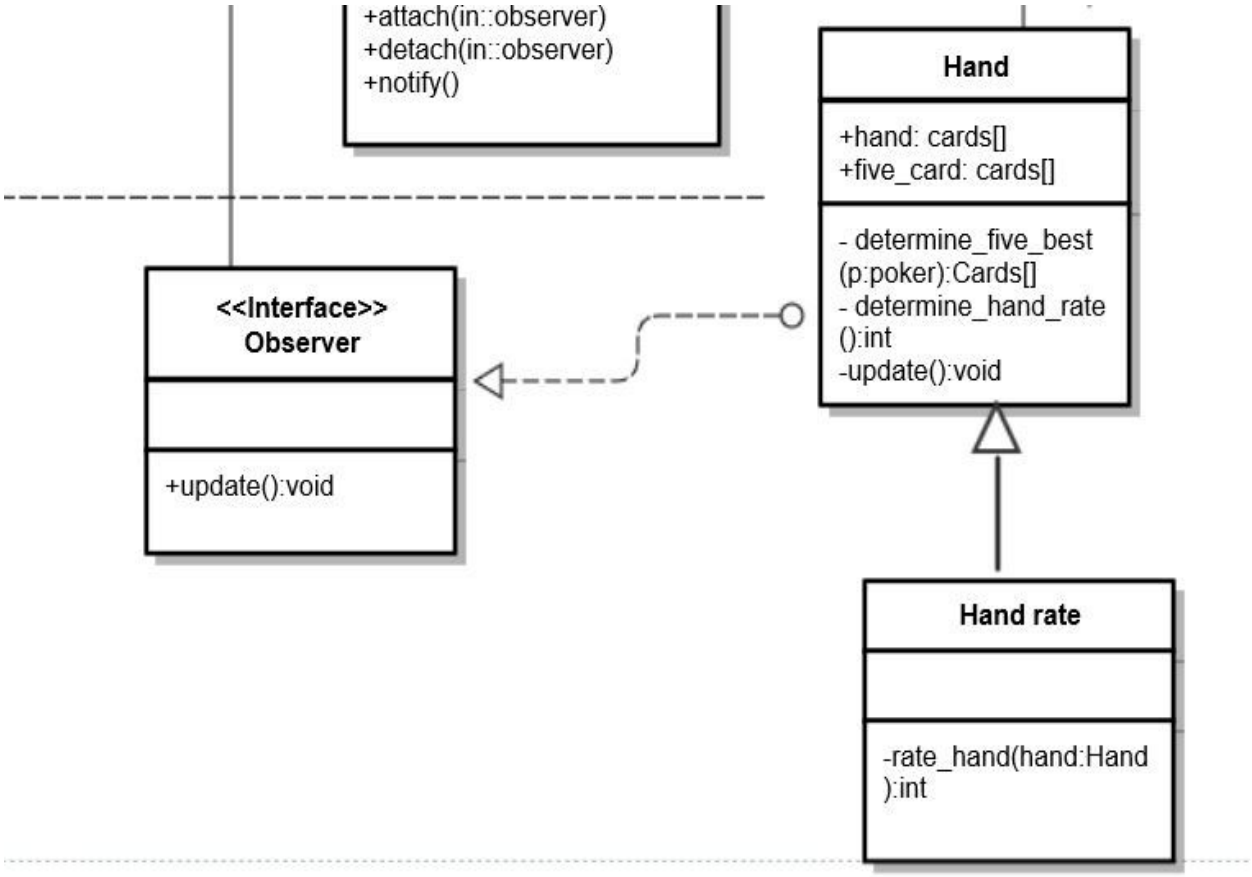


Class Diagram Changes

Between the initial and final versions of our class diagrams, the primary differences are the introduction of an observer, the design pattern we decided to apply to our system, and two new classes, Hand rate and Cards. This design pattern application is described in further depth in the next section. The new classes better modularized the functionalities of our system and increased robustness by accounting for aspects of poker we initially neglected. We did also refactor our existing classes some, including new scopes and functionalities.

4. Design Patterns Used

We used the Observer design pattern to rate each player's hand and determine the winner. After a player folds, they will also be able to see the Observer's calculated likelihood of who will win. We found this useful after refactoring because the best hand at the table constantly changes - something the observer is meant to, well, observe.



5. What We've Learned

After this class, we feel much more confident in our abilities to plan an object-oriented programming project, using design patterns that complement our feature set and avoiding anti-patterns. From the basics of UML and what is necessary to create useful diagrams and documentation to best help guide a project, to the huge number of design patterns that can be taken into consideration for any given system, the complexities of designing a system for large scale use raise issues that require very careful planning. One of the greatest takeaways from this class is that doing your due-diligence up front can save you from massive headaches as your project scales.

Furthermore, planning for everything necessary in your system on your first go is impossible, and we learned various helpful lessons in refactoring. Taking a repeated retrospective looks at your design allows developers to notice flaws and possible areas of improvement. Repeatedly refactoring both code and design aspects is imperative to creating a strong, lasting product.