Lab - 7: Object Oriented Design

Team Members:

Jay Sanjaybhai Patel (jy451478@dal.ca)

Kenil Kevadiya (kn486501@dal.ca)

Date:

March 07, 2024

Subject:

Software Development Concepts

Professor:

Michael McAllister

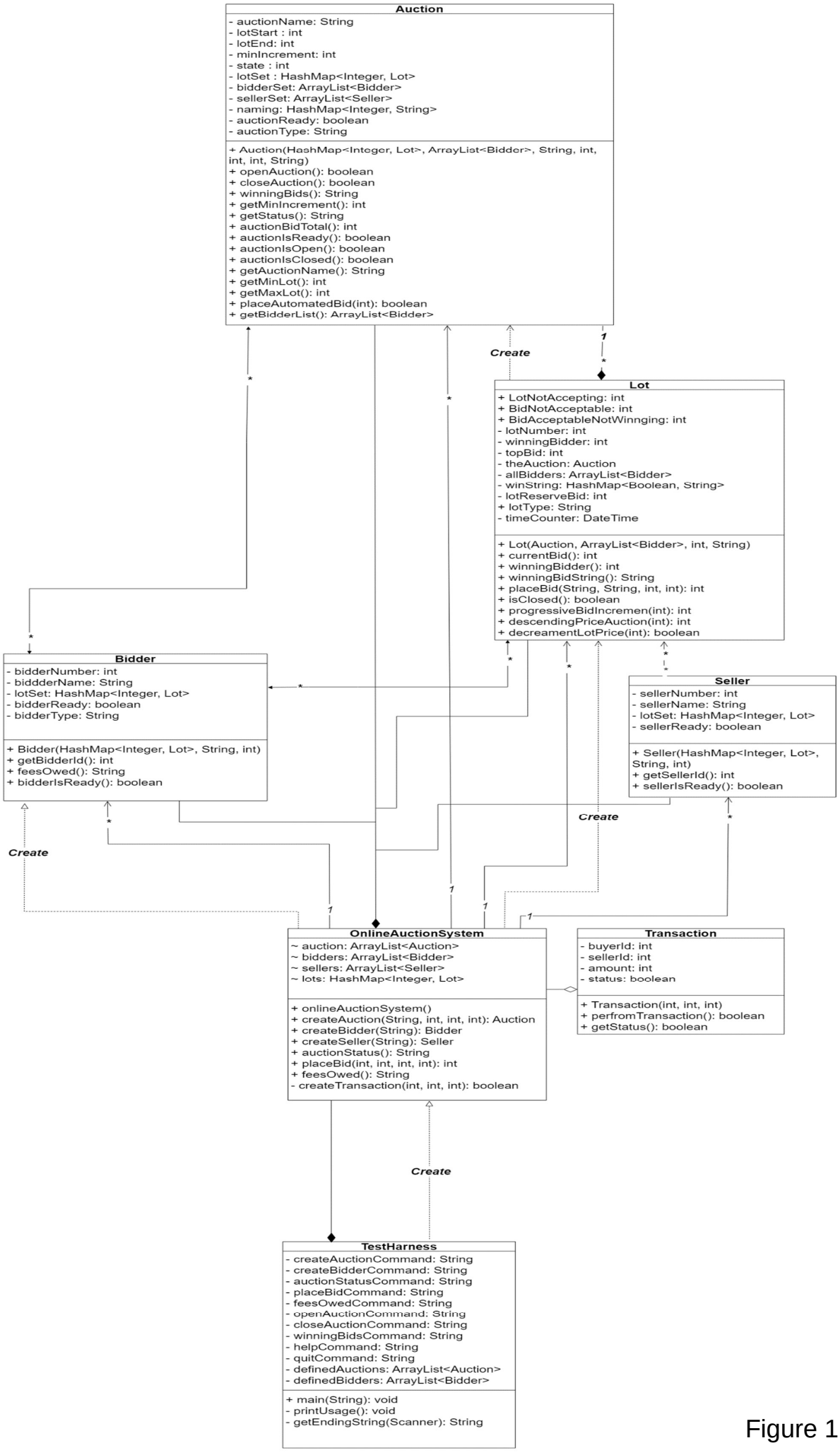

Figure 1: Initial Design

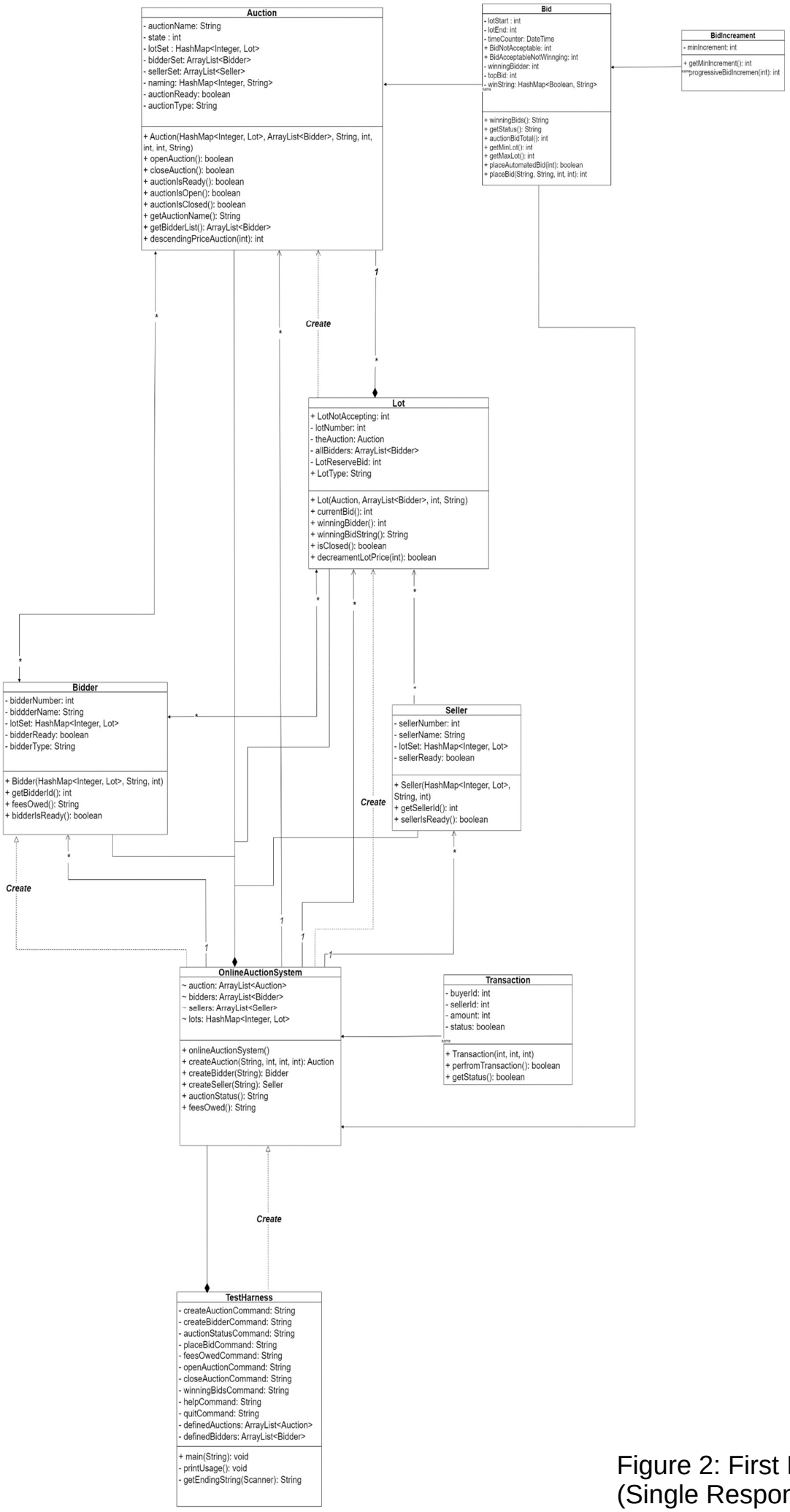

Figure 2: First Iteration
(Single Responsibility Principle)

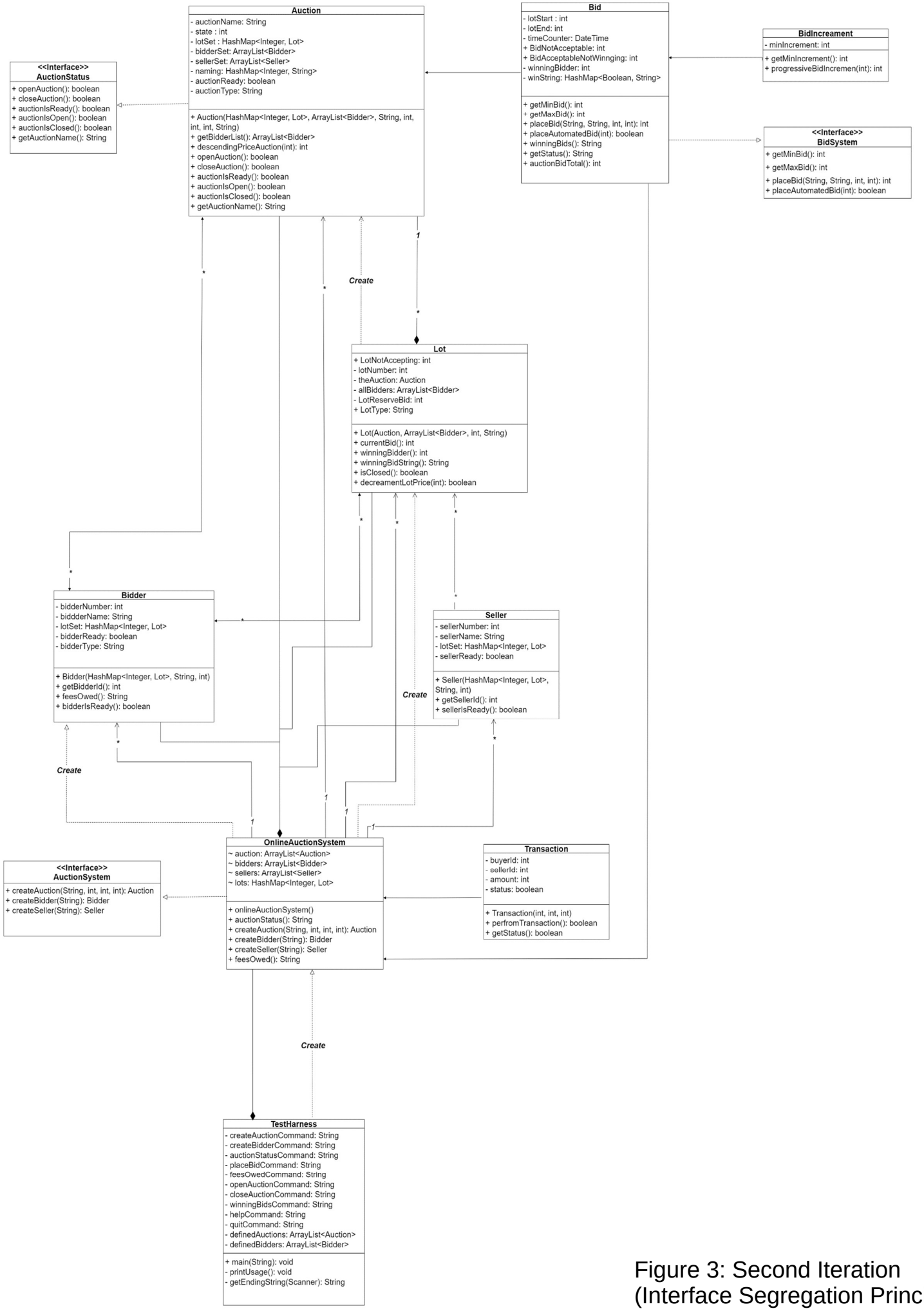

Figure 3: Second Iteration
(Interface Segregation Principle)

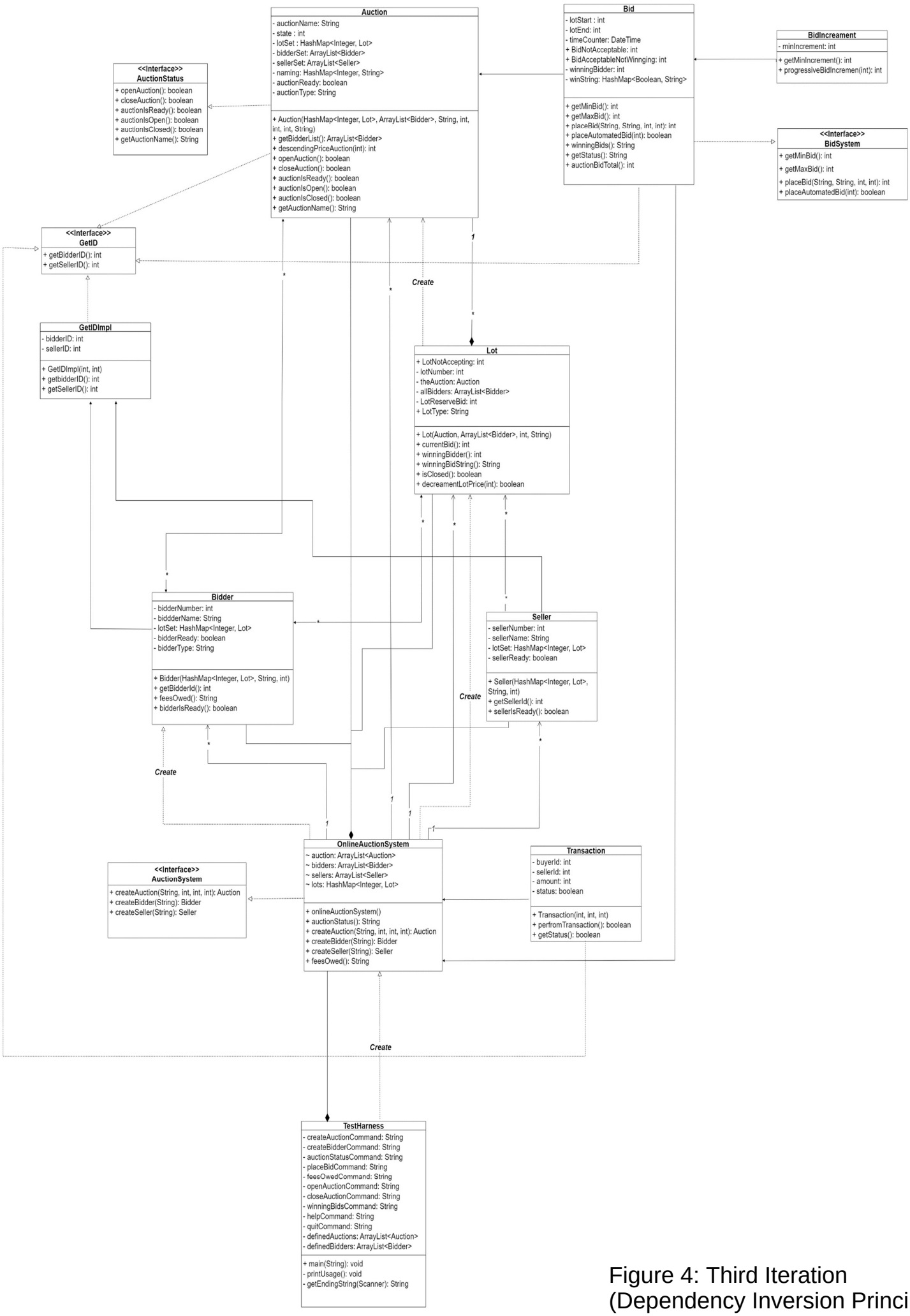

Figure 4: Third Iteration
(Dependency Inversion Principle)

Q – 1: Which SOLID property was the easiest to use for your refinement? What of the problem or your initial design made it easiest to use?

→ The Single Responsibility Principle (SRP) is the SOLID property that was easiest to use in this particular situation.

→ The classes Auction, Bidder, and Lot in the initial lab 6 design handled a variety of tasks like administering auctions, keeping track of bidders, processing bids, and reporting auction status and winning bids.

→ Dividing these classes into smaller, more focused components that are each responsible for a specific system operation is necessary to refine the design to be compatible with the SRP. For instance, improved organization and concern separation are made possible by separating classes to manage lots, bidders, auctions, and bidding procedures; this makes the system easier to use, maintain, and expand.

→ Furthermore, the introduction of updated auction styles and features in lab 7 further emphasizes the need for a clear separation of responsibilities within the system. Each component may concentrate on its unique functionality by sticking to the SRP, which makes it simpler to handle future modifications and improvements.

Q – 2: Which SOLID property was the hardest to use for your refinement? What of the problem or your initial design made it hardest to use?

→ The hardest SOLID property to apply in this case was the Dependency Inversion Principle (DIP). High-level modules must rely on abstractions rather than on low-level modules directly, as per DIP. Abstractions should not rely on details; instead, details should rely on abstractions.

→ There was a strong coupling between classes in the initial Lab 6 design since dependency inversion was not taken into consideration. Because of the direct dependencies between higher-level and lower-level modules, the system was less flexible and more difficult to manage.

→ I created a class called GetIDImpl and implemented an interface called GetID to solve this problem. This interface allowed classes requiring Bidder ID and Seller ID to implement it, thereby decoupling them from concrete implementations of IDs and adhering to the Dependency Inversion Principle. Through the reduction of dependencies and advancement of code reuse, this restructuring was required to increase the flexibility and maintainability of the system.