Exercise 4

UML Class Diagram Relationships ordered by refactoring complexity. Lowest (1) is easiest to refactor and Highest (6) is the hardest to refactor.

1. Dependency
2. Association
3. Aggregation
4. Composition
5. Specialization
6. Inheritance

```
Class A {
        B b;

        Void method() {
                C c;
        }

}
```

We believe dependency is easier to refactor than association because with a dependency you would only have to refactor that method, instead of an association where all the references to the attribute would need to be updated. For example, the Class A above has an association with class B. If you changed the class B you would have to update all the code using A.b. You would have to find each place where this happens. However with the dependency that class A has on class B you only need to change that single method.

Dependency and association are easier to refactor than aggregation and composition because aggregation and composition are more complex versions of associations. With aggregation the association means that something is part of a whole but the whole can exist without the parts. If you refactored an aggregation you would not only have to update the references to the attribute but also update the structure of the relationship so that the “part” of the whole doesn’t get removed with the “whole.”

Composition is harder to refactor than aggregation because there is another relationship that must be met in the refactoring. You must make sure that if the “whole” is destroyed then each “part” is also destroyed.

Specialization and Inheritance are harder to refactor than composition and aggregation because if you changed the “whole” of the composition you wouldn’t necessarily have to change each “part’s” attributes and functions. But if you changed an interface then you would have to refactor every interface or class that implement that interface. This is not always possible because if you released a package to a customer that contained an interface, they may implement that interface in their own classes. If you wanted to change the interface you cannot go into their code and implement it properly. You would have to make parts of the interface obsolete and add new functions.

Inheritance is more difficult to refactor then specialization because now you can change more. A superclass may have function signatures but also implemented methods and also attributes. Changing a superclass will again cause you to change all the classes implementing it. This again might not be possible. However now you have to change more than just the functions, you have to update any code that used functions implemented in the superclass and update attributes and all of their references.