Team members:

Alex Verkerk: aver551
Oliver Chamberlain: ocha817

Task 1:

1.0 Class relationships

All command classes implement the command interface which is, in turn, depended on by CommandFactory. CommandFactory also depends on keyboard input as no permanent keyboard input is stored as the getUserCommand method uses the keyboard method. CommandFactory is used by FreeCell to run the game (ultimately allowing the user to interact via the keyboard) so there is a dependency in the direction of CommandFactory from FreeCell.

Freecell is associated with both ASCIIBoard and deck as it stores instances of both. ASCIIBoard in turn depends on Deck for board creation. Deck stores lists of cards so it has an aggregate relationship with the Card class in that is stores up to 52 cards but each card can only be associated with one Deck. CascadeToCascadeCommand, CascadeToCellCommand, SaveGameCommand and CellToCascadeCommand are all associated with ASCIIBoard as they all store references to instances of it in fields; they require access to the state of the board.

1.1 Discussions and conflicts

Initially, Alex believed there was an aggregation relationship between CommandFactory, however upon Oliver refuting this, and collectively both of us going back through a large number of slides, we concluded that CommandFactory did not store and Command objects and instead created and returned Command objects. This means that it is most likely a dependency between CommandFactory and objects of type Command.
Another dispute between the team members was around the reference for the Deck being stored within the board class. Upon trawling through copious amounts of slides we concluded that the Deck was passed into board but a local reference was not stored, making this a dependency rather than an association.

Task 2:

A singleton antipattern exists for the board and deck. Board and deck should only be instantiated once per game yet be accessible. The developer has opted to use public constructors which defies this idea as it does not guarantee only one instance will be created during the game. Upon discussion, we decided that the board and deck classes should have private constructors with public ‘get’ methods. These methods would create a new instance of the board and deck classes respectively and store them in their own fields. After this initial object creation the get methods will not instantiate the classes again but merely return the instances stored in their fields.

There exists an observer antipattern for keyboard input and the command factory class. The pattern stipulates that the observable (keyboard input) and the observer (command factory) should not directly reference each other as they do currently. This limits the ease of implementing other classes that, like command factory, may also need to 'listen' for keyboard inputs. This design pattern could be implemented by creating a subject class with attach, detach and notify methods which are associated with an observer interface. CommandFactory then extends this interface. This would allow CommandFactory, and anything else that implements the interface to listen, and act on, changes in keyboard input.

The template design pattern is already partially implemented in the Command interface. The current engage method itself acts as a hook which can call different implementations of itself in the implementing command classes. To fully implement the pattern, it could be split into a final engageChild method (which cannot be overridden) and a engage abstract method (which can be implemented differently by all of the command classes). This would allow different implementations of the engage method to be used whilst allowing for the expansion of functionality across command classes.

The Card class currently contains the immutable antipattern in that it is not final, and the fields are not final. This means the Card class could be extended and its state (the fields) edited which would open the game up to the cards being modified. Card should be made final and all fields also made final with a get method to return a copy of the card. This would allow other classes to access information on the card's state without editing it.

Report

Both of us had minimal UML experience prior to this lab. Neither of us were particularly confident with conventions for creating class diagrams. During this lab, and after a lot of struggling through papyrus, we found it very difficult to wrap our heads around UML, as papyrus felt very dated and not particularly user-friendly. We attempted to find more modern UML creation tools but found out that we were restricted to using papyrus for this lab. As a result, toward the end of the lab, we didn't feel like our grasp of UML had increased appreciably. Our understanding of the papyrus UML tool also didn't increase as we still struggled with extremely basic operations such as adding final modifiers to classes, methods and fields. Some guidance on actually using some of the more esoterically named parts of papyrus which were used for basic functionality would have been appreciated.
The changing nature of Freecell across the many lectures of part 1 also didn't aid in our understanding. The majority of the time we spent on this lab was trying to figure out which classes had the most up to date information in order to determine relationships. This meant that we spent less time trying to understand UML and more time trying to find up to date information.