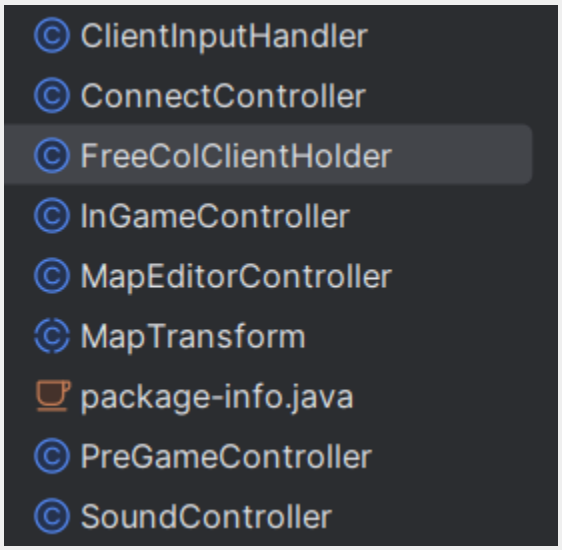
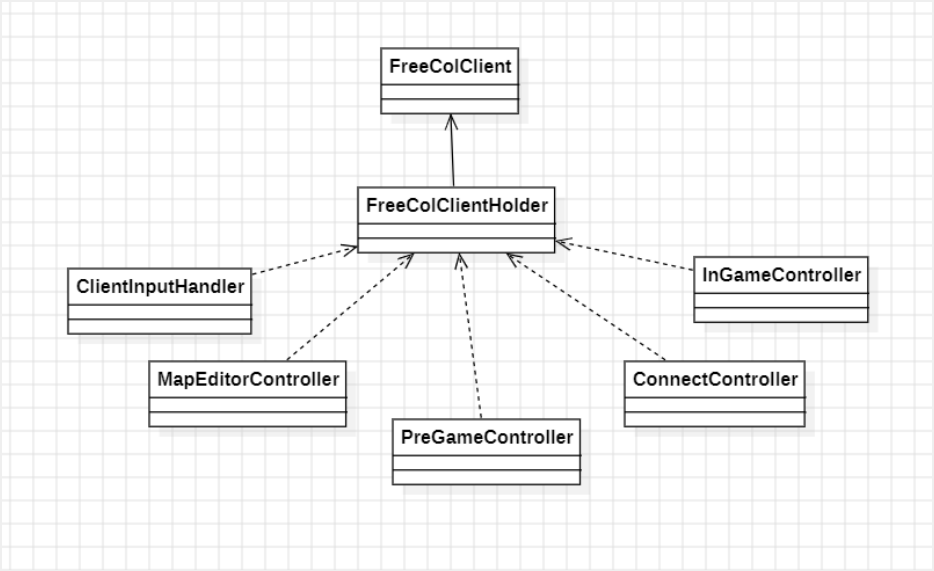
Design Patterns

Facade Pattern

Path: net/sf/freecol/client/control/FreeColClientHolder.java



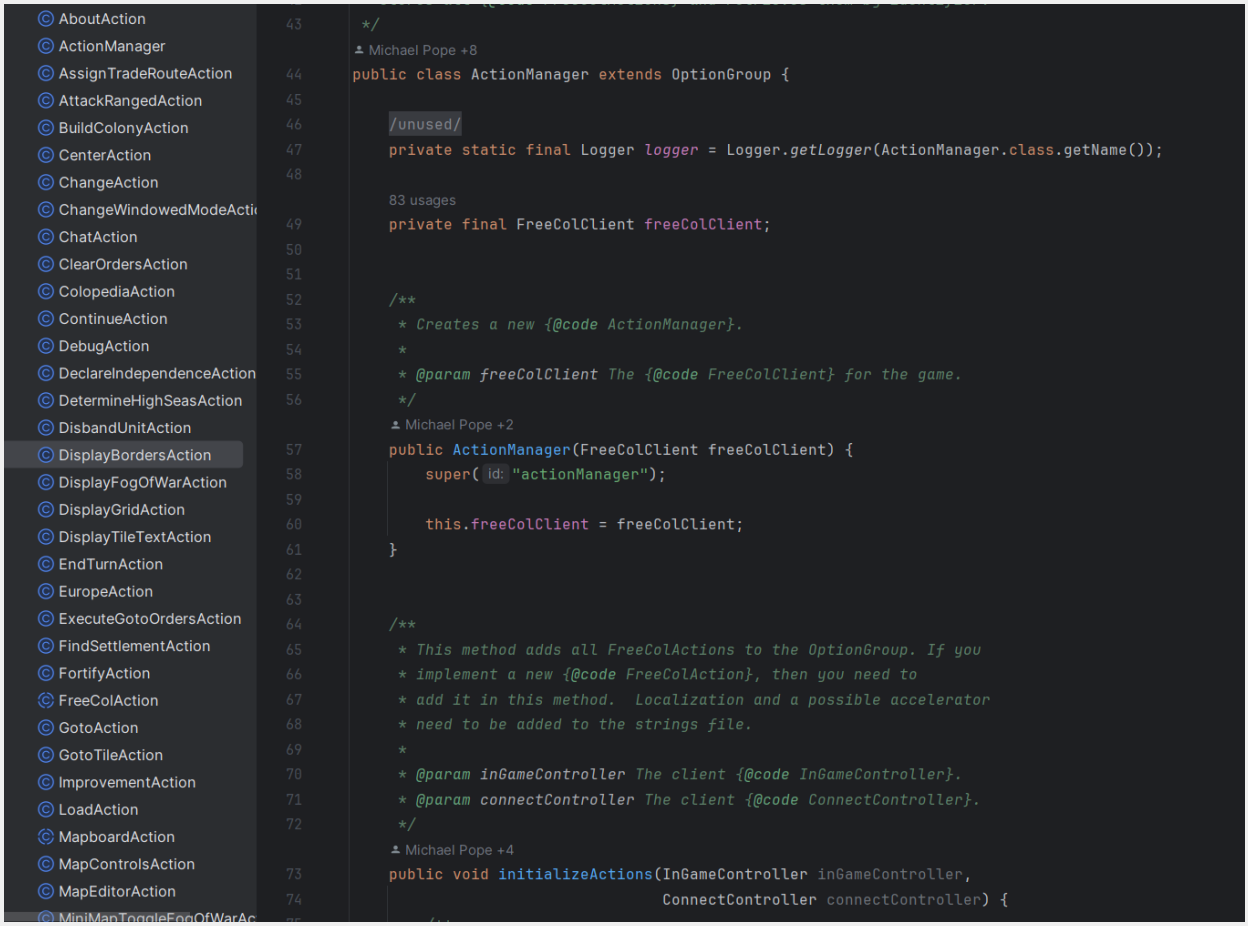
The design pattern that I consider here is the Facade design pattern, which is used so that the user can only access certain parts of a complex system, only the ones that are needed. The FreeColClientHolder constructor creates an object FreeColClient that directly has access to the essential game mechanics. All other classes in the package access the FreeColClientHolder, to not have direct access to the FreeColClient class. This makes so that all the other controllers have access to the methods that are necessary.



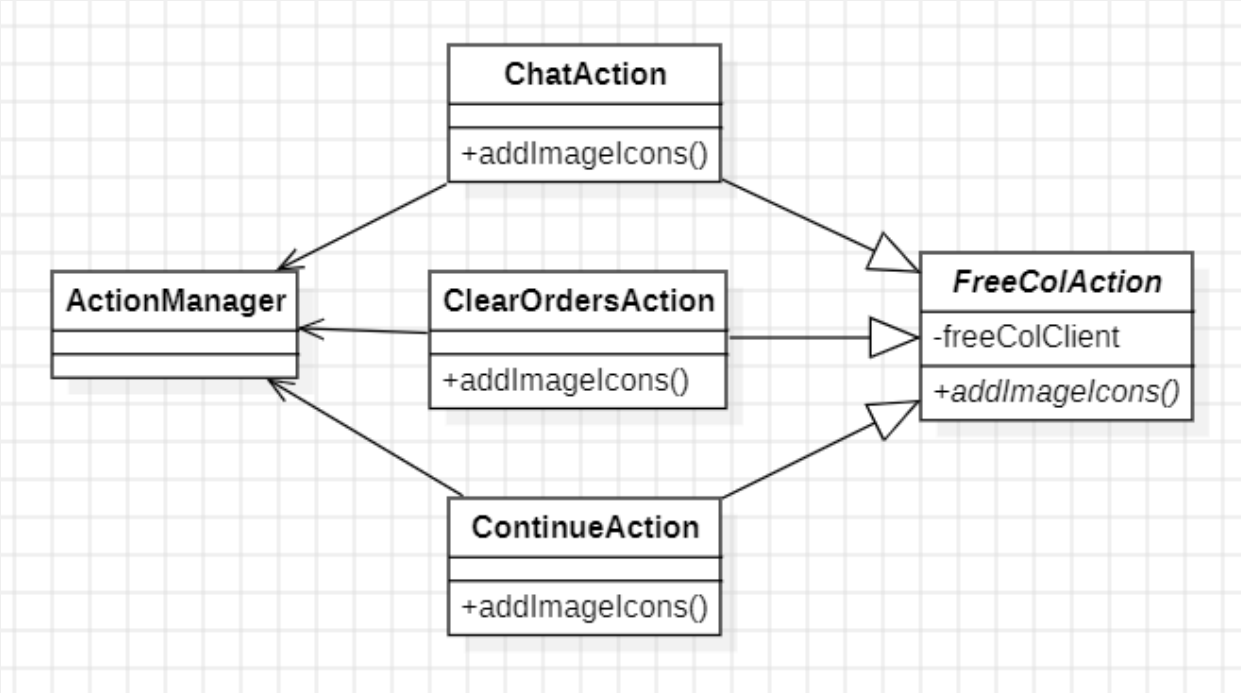
With the UML diagram I represent that the FreeColClient is the main system, the FreeColClientHolder class is the facade that is being used by the other five classes (the clients in this case) to access the main system.

Command Pattern

Path: net/sf/freecol/client/gui/action/ActionManager.java



The design pattern found here is the command pattern. I consider this a command pattern in code because we have the ActionManager, which is the class that calls the different actions possible. In our case the different type of actions are listed in the package (ChatAction, EndTurnAction, EuropeAction, etc.) are our various actions that ActionManager can use. These actions are then extending other abstract classes. These work as subclasses, as all of them extend FreeColAction. Comparing to the editor and command example taught in class, the editor in our example is the ActionManager and the commands are the actions, while the main command abstract class is FreeColAction.

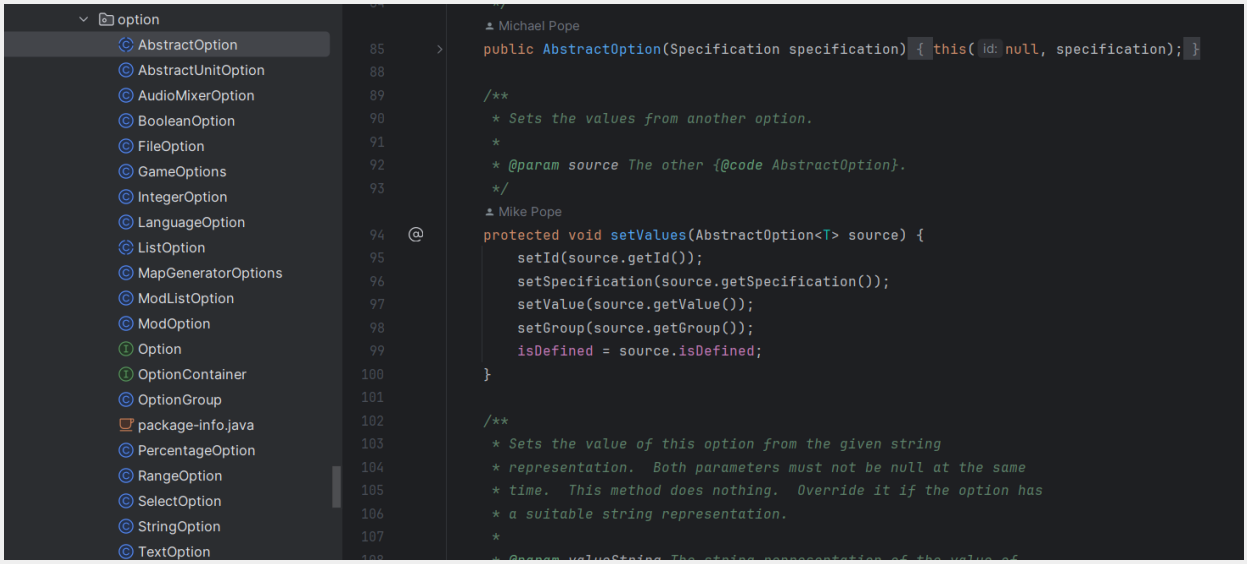


The operations and attributes used are minimal and not all extensions of abstract classes are shown.

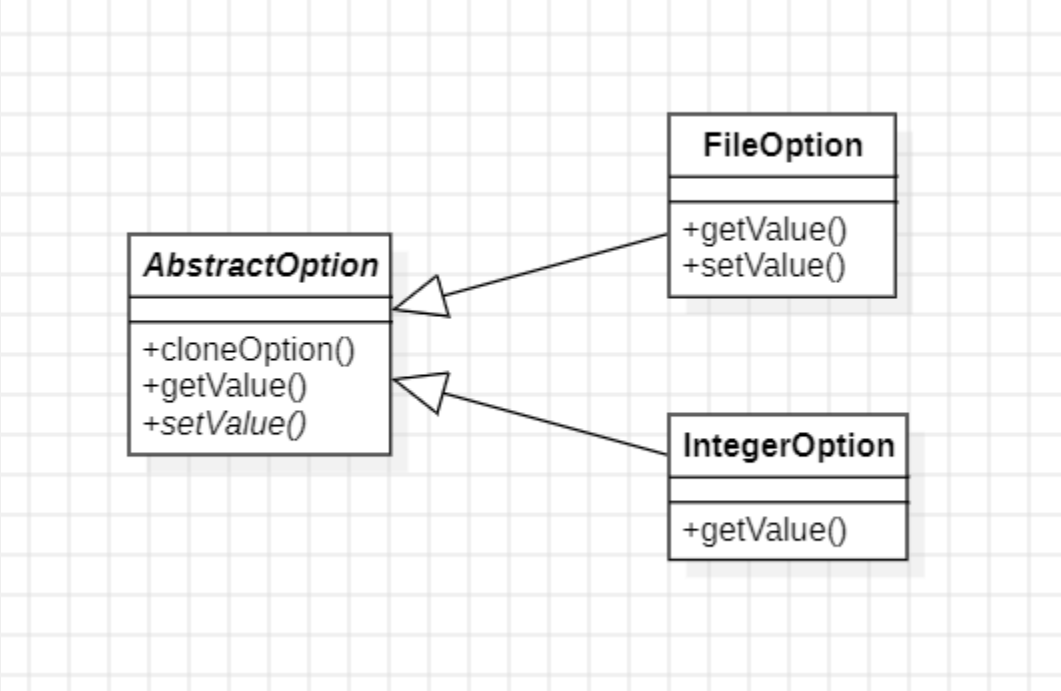
In the Class Diagram above we can see that the FreeColAction is the main abstract class while ChatAction, ClearOrdersAction and ContinueAction are the classes that extend, while the ActionManager is the class that calls the actions.

Template Method

Path: net/sf/freecol/common/option/AbstractOption.java



Here we can find the Template Method being used. The “skeleton” is the AbstractOption class while other classes such as FileOption, TextOption, IntegerOption, etc. are the various branches of the skeleton. This is because the AbstractOption class has methods that are base for the rest, such as cloneOption() and getValue(), and other methods to be overridden or implemented by the subclasses, such as, setValue() and toString().



The operations and attributes used are minimal.

In the class diagram we can see that the methods used in FileOption and IntegerOption are overriding the method in AbstractOption (getValue()) and is also implementing the abstract methods of AbstractOption (setValue()).