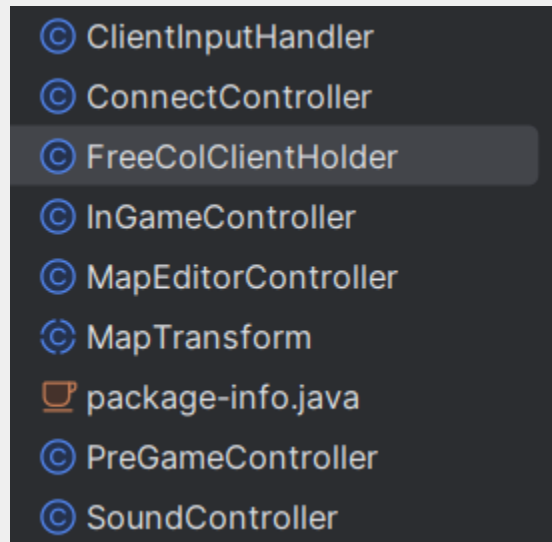


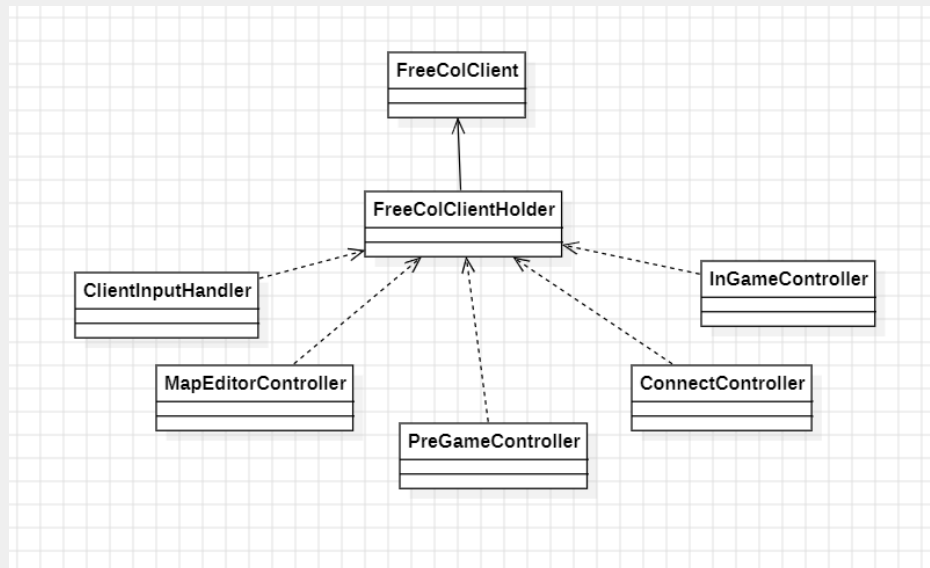
Design Patterns

Facade Pattern

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



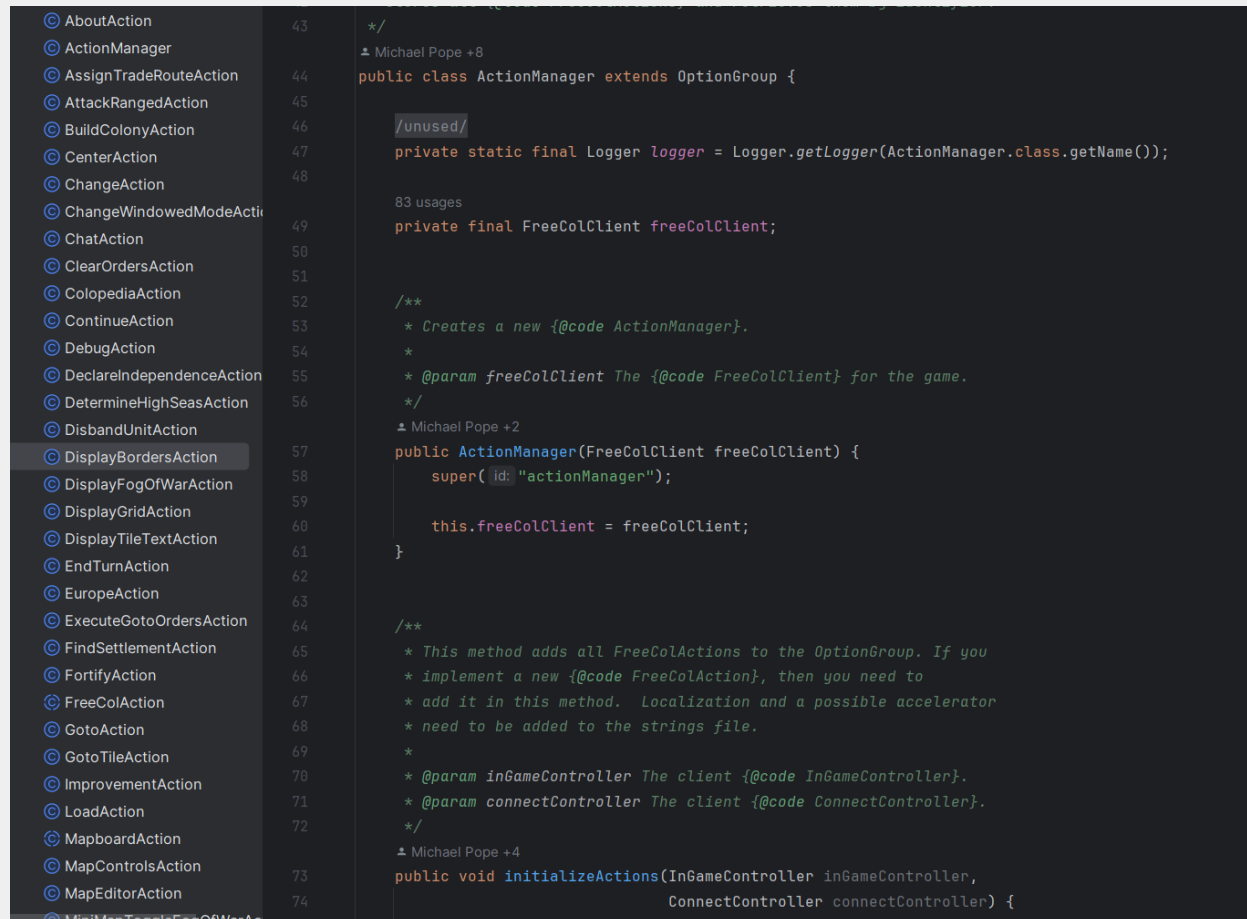
The design pattern that I consider here is the Facade design pattern, which is used so that the user can only access certain parts of a complex system, only the ones that are needed. The FreeColClientHolder constructor creates an object FreeColClient that directly has access to the essential game mechanics. 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.

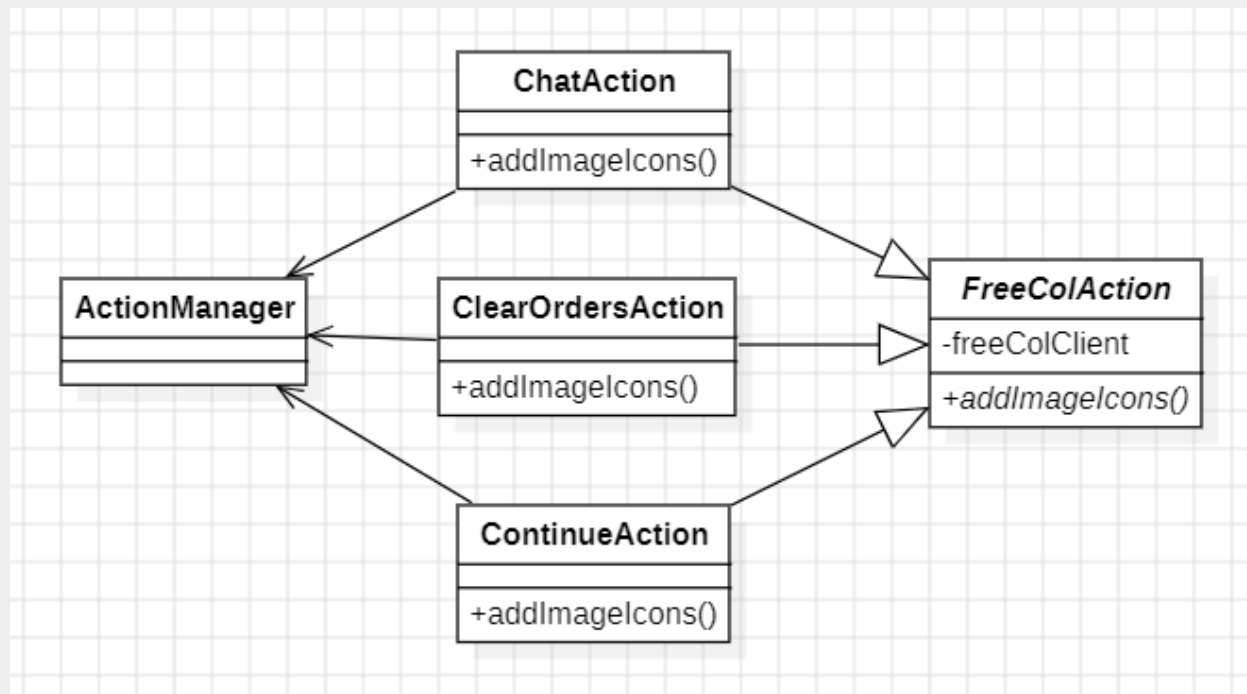
Command Pattern

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



```
43  */
44  public class ActionManager extends OptionGroup {
45
46      /unused/
47      private static final Logger logger = Logger.getLogger(ActionManager.class.getName());
48
49      83 usages
50      private final FreeColClient freeColClient;
51
52      /**
53       * Creates a new {@code ActionManager}.
54       *
55       * @param freeColClient The {@code FreeColClient} for the game.
56       */
57      public ActionManager(FreeColClient freeColClient) {
58          super(id: "actionManager");
59
60          this.freeColClient = freeColClient;
61      }
62
63      /**
64       * This method adds all FreeColActions to the OptionGroup. If you
65       * implement a new {@code FreeColAction}, then you need to
66       * add it in this method. Localization and a possible accelerator
67       * need to be added to the strings file.
68       *
69       * @param inGameController The client {@code InGameController}.
70       * @param connectController The client {@code ConnectController}.
71       */
72      public void initializeActions(InGameController inGameController,
73                                  ConnectController connectController) {
74
75      }
```

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.

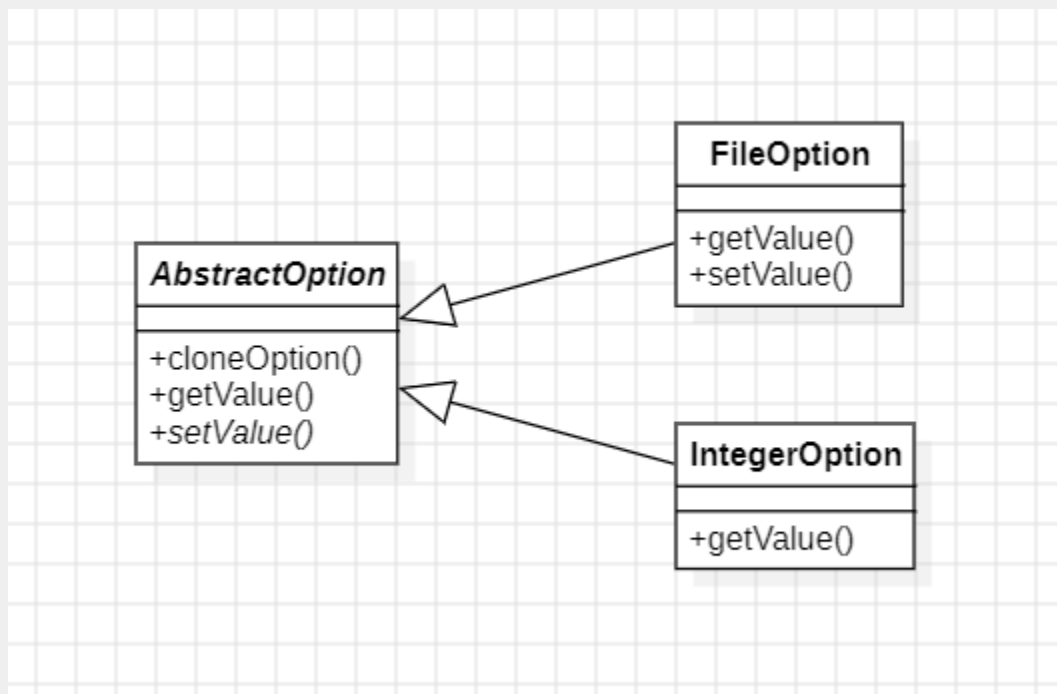
Template Method

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

```
option
├── AbstractOption
├── AbstractUnitOption
├── AudioMixerOption
├── BooleanOption
├── FileOption
├── GameOptions
├── IntegerOption
├── LanguageOption
├── ListOption
├── MapGeneratorOptions
├── ModListOption
├── ModOption
├── Option
├── OptionContainer
├── OptionGroup
├── package-info.java
├── PercentageOption
├── RangeOption
├── SelectOption
├── StringOption
└── TextOption

85 public AbstractOption(Specification specification) { this(id: null, specification); }
86
87 /**
88  * Sets the values from another option.
89  *
90  * @param source The other {@code AbstractOption}.
91  */
92
93
94 @Mike Pope
95 protected void setValues(AbstractOption<T> source) {
96     setId(source.getId());
97     setSpecification(source.getSpecification());
98     setValue(source.getValue());
99     setGroup(source.getGroup());
100     isDefined = source.isDefined;
101 }
102
103 /**
104  * Sets the value of this option from the given string
105  * representation. Both parameters must not be null at the same
106  * time. This method does nothing. Override it if the option has
107  * a suitable string representation.
108  *
109  * @param valueString The string representation of the value of
```

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()).