Roman Hanushchak 127709

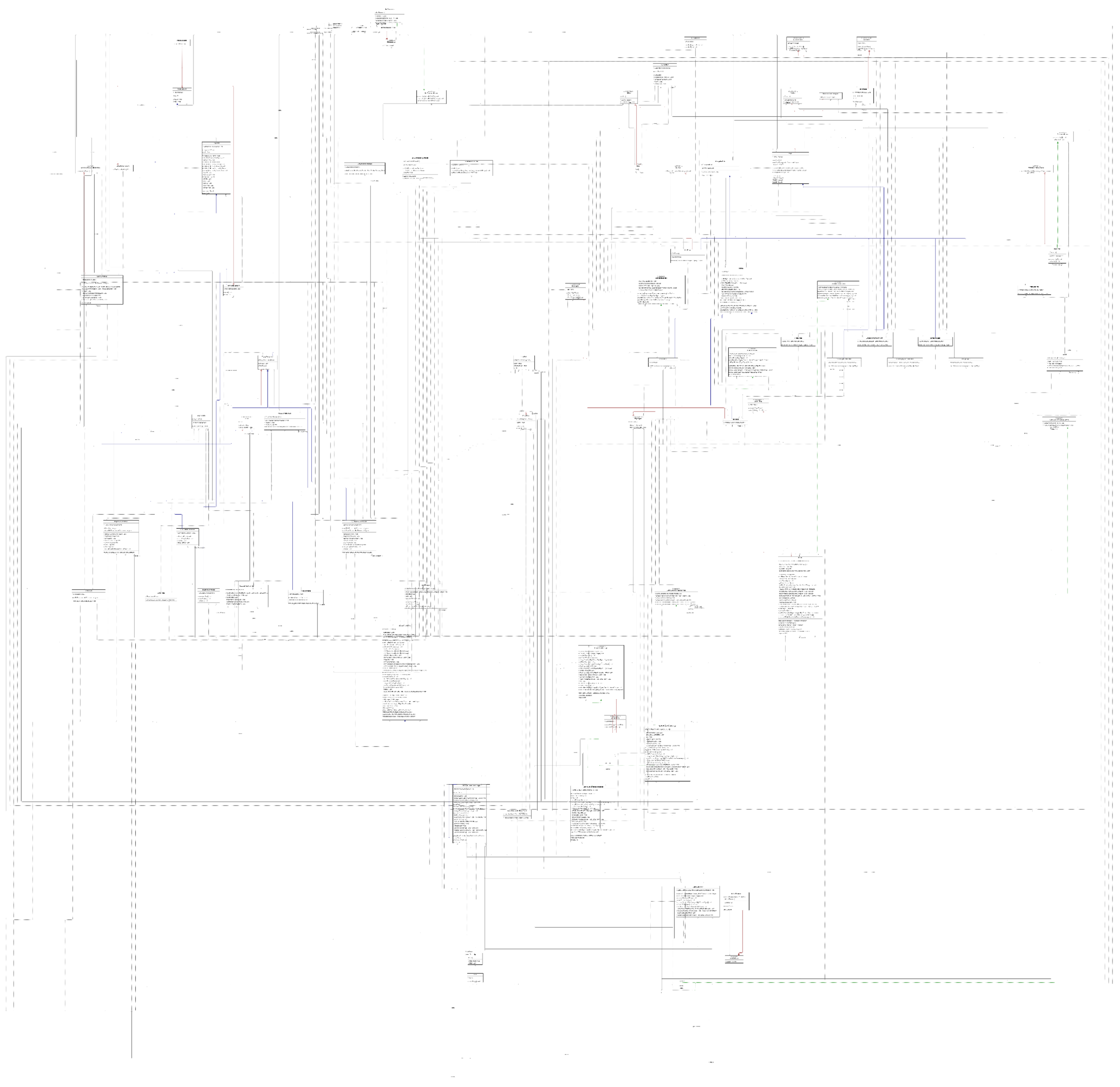
Game Reveal the spy

The goal of the project was to create a computer version of board game “Secret spy”.

Election mechanics - the core concept of the game is chancellor and in father game president election. The players have to choose their candidate based on the existing information and try to reveal other players' roles. The president chooses a chancellor and then players can vote for or against it. If the vote fails then the next president and chancellor candidate will be chosen.

On the right side of the screen the right part is present. Players can use the rights using these buttons. Only the president and chancellor have any rights. On the left side all players can be seen. All of them are bots. The main goal for liberals is to get 5 liberal cards to the board and not allow the shadow leader to become chancellor after 3 spy cards were added. Spies have to collect 6 spy cards or collect 3 or more and elect a shadow leader as chancellor.

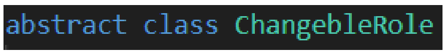
Class diagram



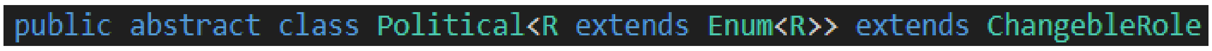
The diagram is sent separately with better quality of diagram, name diagram.drawio or diagram.png in project_info

The javadoc documentation can be found in javadoc/javadoc each file separately

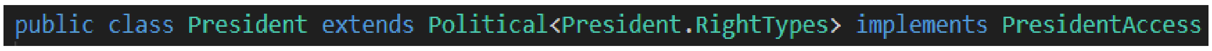
Double extent can be found in model/ChangebleRole/Political, where Political extends ChangebleRole and President extends Political and Chancellor extends Political. President implements Interface President Access.



```
abstract class ChangebleRole
```

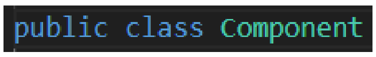


```
public abstract class Political<R extends Enum<R>> extends ChangebleRole
```

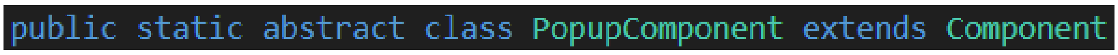


```
public class President extends Political<President.RightTypes> implements PresidentAccess
```

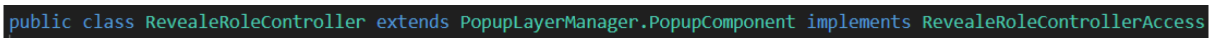
Other double extends can be found in test_ui/Components/`RevealeRoleController`, where `RevealeRoleController` extends `PopupLayerManager.PopupComponent` (in test_ui/PopupLayerManager), and `PopupComponent` extends test_ui/Components/Component/`Component`, and `RevealeRoleController` implements interface `RevealeRoleControllerAccess.`



```
public class Component
```



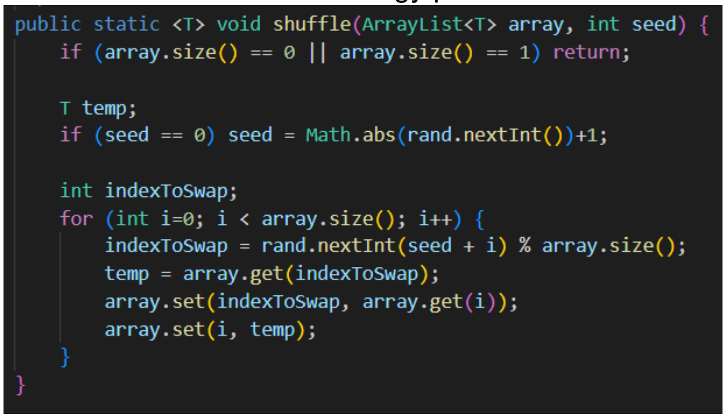
```
public static abstract class PopupComponent extends Component
```



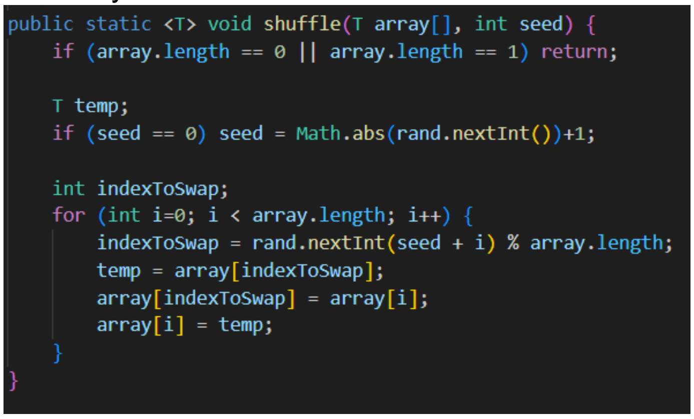
```
public class RevealeRoleController extends PopupLayerManager.PopupComponent implements RevealeRoleControllerAccess
```

Further criterias.

- Patterns implemented:
    1. Strategy pattern used in model/Cards/ArrayShuffle

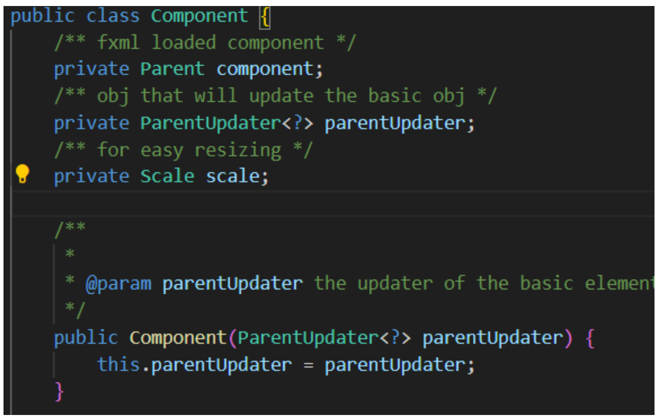


```
public static <T> void shuffle(ArrayList<T> array, int seed) {
    if (array.size() == 0 || array.size() == 1) return;

    T temp;
    if (seed == 0) seed = Math.abs(rand.nextInt())+1;

    int indexToSwap;
    for (int i=0; i < array.size(); i++) {
        indexToSwap = rand.nextInt(seed + i) % array.size();
        temp = array.get(indexToSwap);
        array.set(indexToSwap, array.get(i));
        array.set(i, temp);
    }
}
```

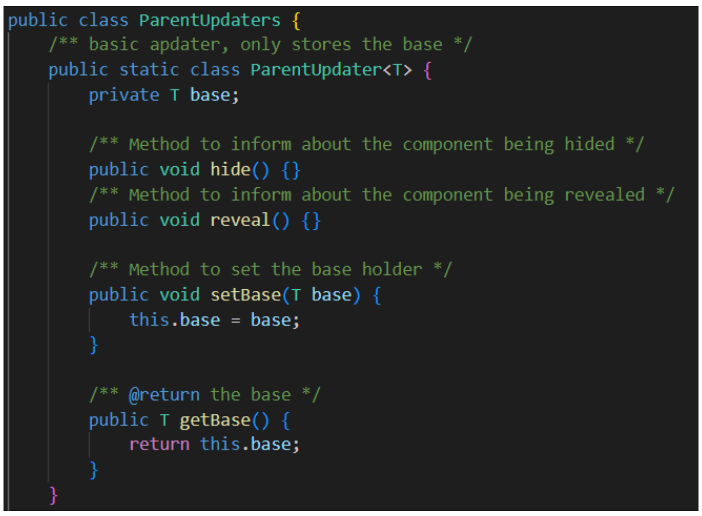


```
public static <T> void shuffle(T array[], int seed) {
    if (array.length == 0 || array.length == 1) return;

    T temp;
    if (seed == 0) seed = Math.abs(rand.nextInt())+1;

    int indexToSwap;
    for (int i=0; i < array.length; i++) {
        indexToSwap = rand.nextInt(seed + i) % array.length;
        temp = array[indexToSwap];
        array[indexToSwap] = array[i];
        array[i] = temp;
    }
}
```

Other using of strategy pattern can be found in test_ui/Components/Component/Component where the contractor takes the obj `ParentUpdater` and based on the type different parent updates will happen, so the creator of component can choose a type of parentUpdater what is needed.

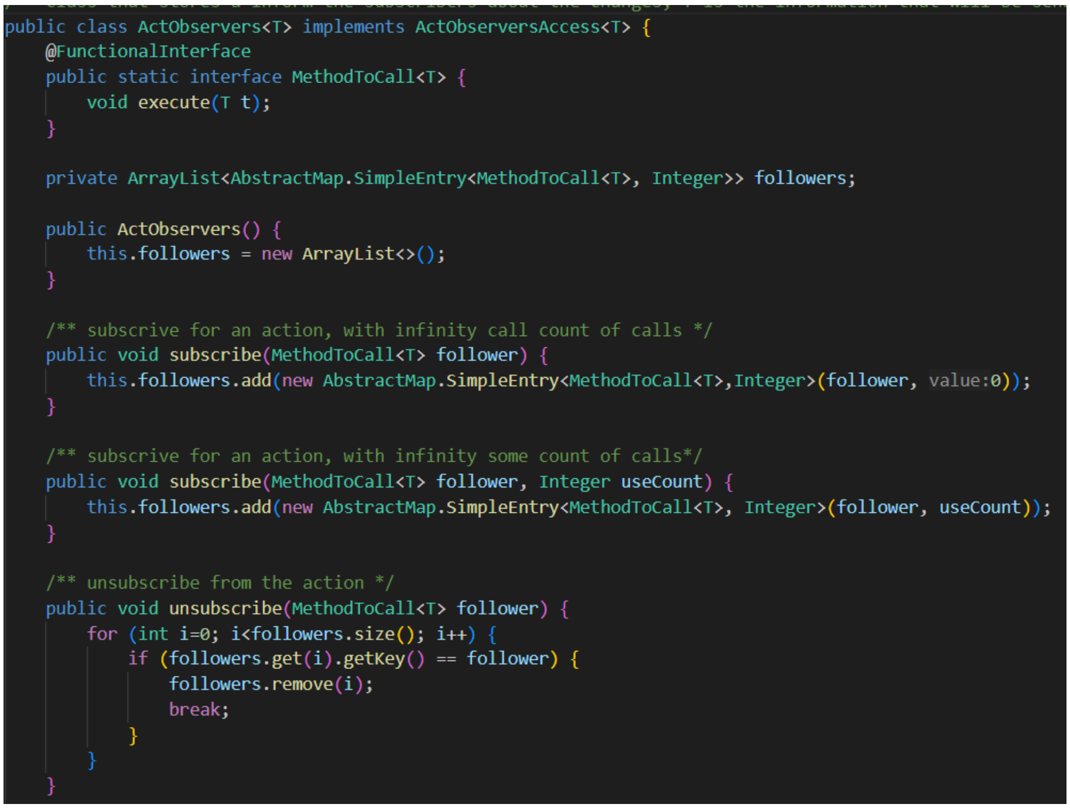


```
public class Component {
    /** fxml loaded component */
    private Parent component;
    /** obj that will update the basic obj */
    private ParentUpdater<?> parentUpdater;
    /** for easy resizing */
    private Scale scale;

    /**
     *
     * @param parentUpdater the updater of the basic element
     */
    public Component(ParentUpdater<?> parentUpdater) {
        this.parentUpdater = parentUpdater;
    }
```



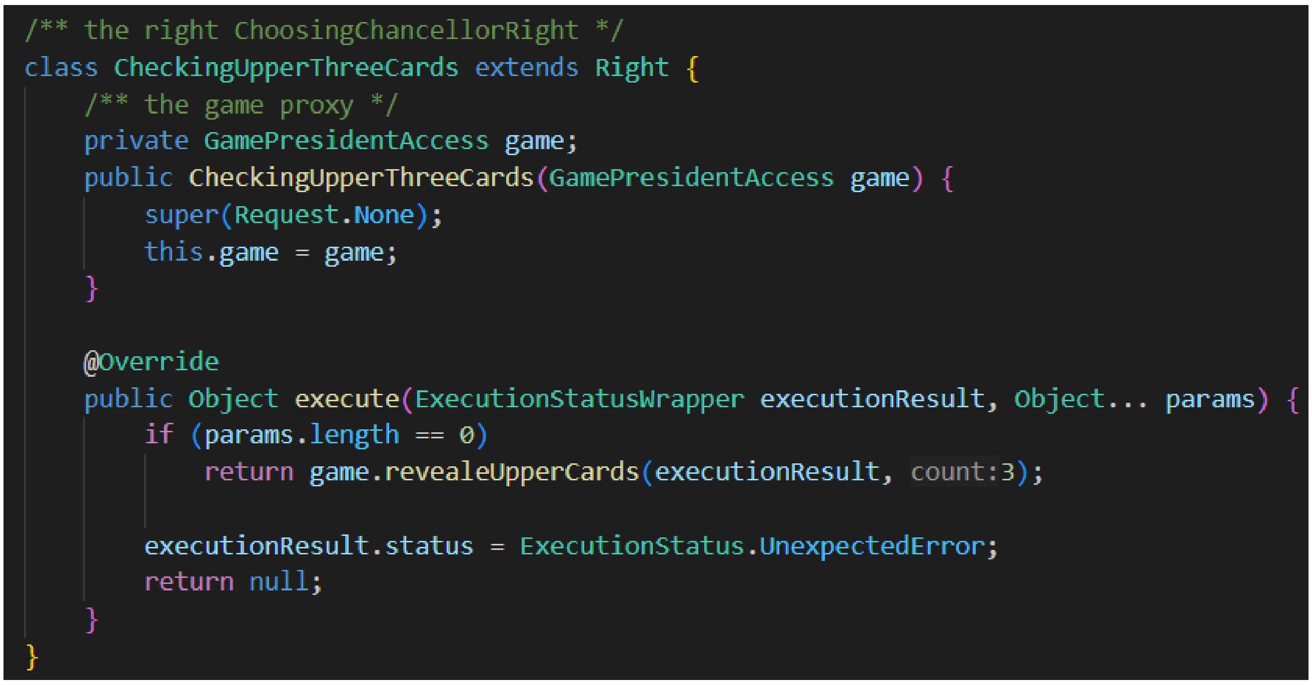
```
public class ParentUpdaters {
    /** basic apdater, only stores the base */
    public static class ParentUpdater<T> {
        private T base;

        /** Method to inform about the component being hided */
        public void hide() {}
        /** Method to inform about the component being revealed */
        public void reveal() {}

        /** Method to set the base holder */
        public void setBase(T base) {
            this.base = base;
        }

        /** @return the base */
        public T getBase() {
            return this.base;
        }
    }
```

2. Observer patern implemented observer structure in model/Observers/ActObservcers, used to create a list of observers to then use it, used in model/ChangebleRole/Political/Political line 50



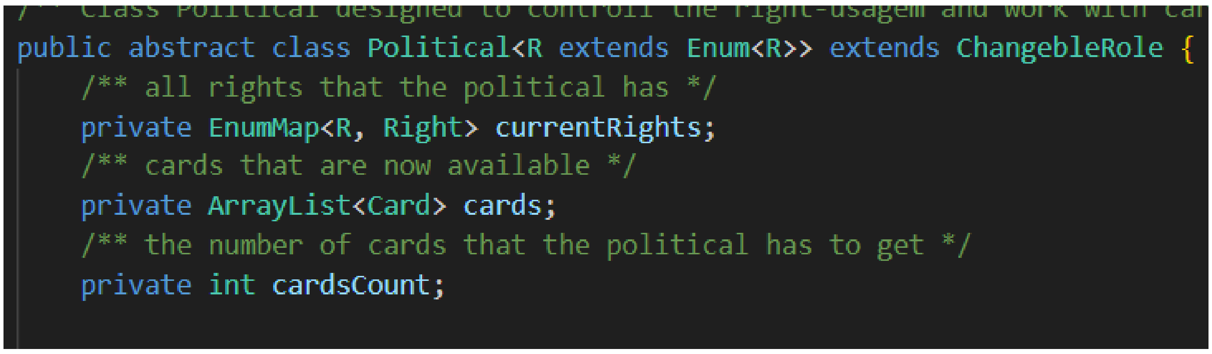
```java
public class ActObservers<T> implements ActObserversAccess<T> {
    @FunctionalInterface
    public static interface MethodToCall<T> {
        void execute(T t);
    }

    private ArrayList<AbstractMap.SimpleEntry<MethodToCall<T>, Integer>> followers;

    public ActObservers() {
        this.followers = new ArrayList<>();
    }

    /** subscrive for an action, with infinity call count of calls */
    public void subscribe(MethodToCall<T> follower) {
        this.followers.add(new AbstractMap.SimpleEntry<MethodToCall<T>,Integer>(follower, value:0));
    }

    /** subscrive for an action, with infinity some count of calls*/
    public void subscribe(MethodToCall<T> follower, Integer useCount) {
        this.followers.add(new AbstractMap.SimpleEntry<MethodToCall<T>, Integer>(follower, useCount));
    }

    /** unsubscribe from the action */
    public void unsubscribe(MethodToCall<T> follower) {
        for (int i=0; i<followers.size(); i++) {
            if (followers.get(i).getKey() == follower) {
                followers.remove(i);
                break;
            }
        }
    }
```

3. Command pattern, used to implement president and chancellor right, the command has to extend Right class and implement execute method, used in method model/ChangebleRole/Political/Political/useRIght to execute it. Command classes in model/ChangebleRole/President



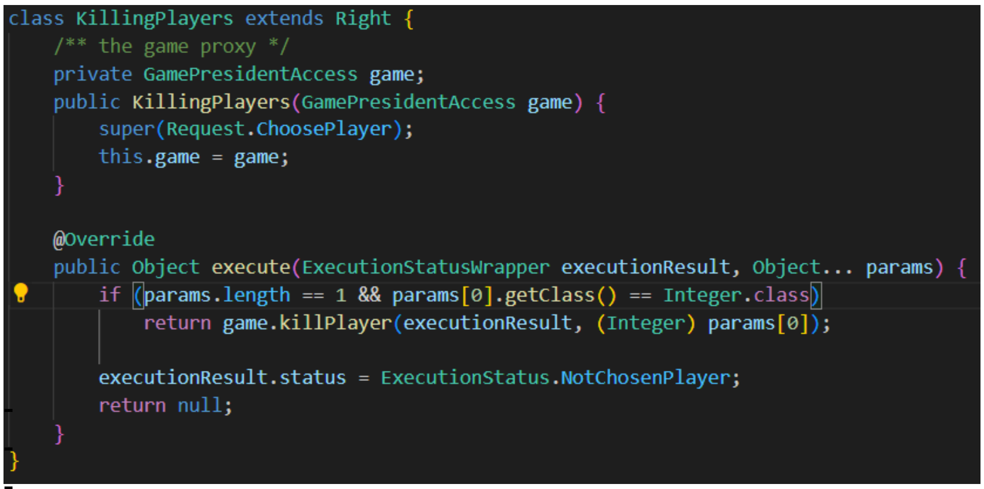
```java
/** the right ChoosingChancellorRight */
class CheckingUpperThreeCards extends Right {
    /** the game proxy */
    private GamePresidentAccess game;
    public CheckingUpperThreeCards(GamePresidentAccess game) {
        super(Request.None);
        this.game = game;
    }

    @Override
    public Object execute(ExecutionStatusWrapper executionResult, Object... params) {
        if (params.length == 0)
            return game.revealeUpperCards(executionResult, count:3);

        executionResult.status = ExecutionStatus.UnexpectedError;
        return null;
    }
}
```

- The gui was provided using the javafx library, can be seen during program run, all interface parts implemented in test_ui and PlayerGameManager/HumanPlayerGameManager
- Using generics - generic used in observers, model/ChangebleRole/Political/Political, model/Cards/ArrayShuffle/shuffle



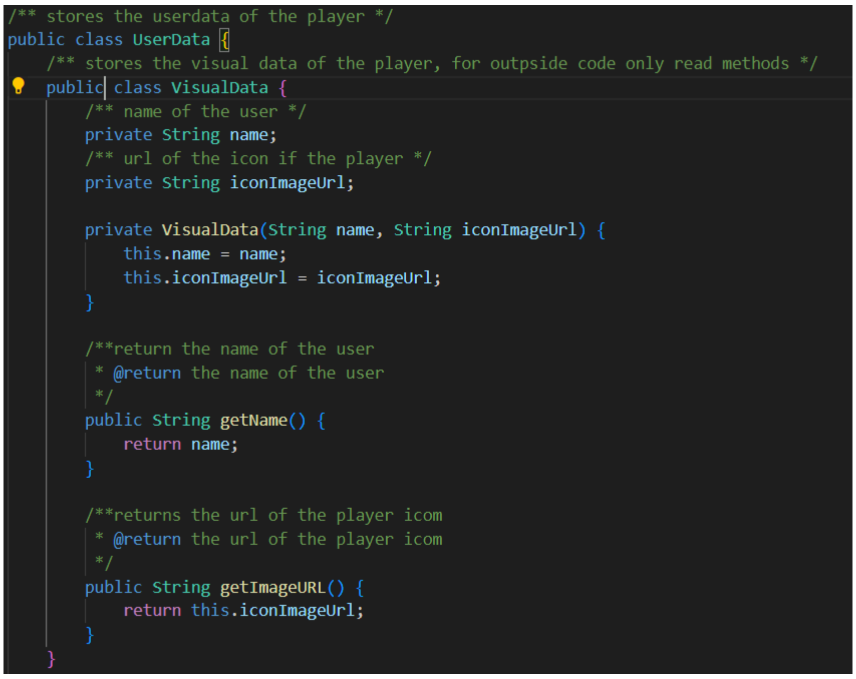
```java
/** class Political designed to controll the right-usagem and work with car
public abstract class Political<R extends Enum<R>> extends ChangebleRole {
    /** all rights that the political has */
    private EnumMap<R, Right> currentRights;
    /** cards that are now available */
    private ArrayList<Card> cards;
    /** the number of cards that the political has to get */
    private int cardsCount;
```

- explicit use of RTTI used in president command KillingPlayer



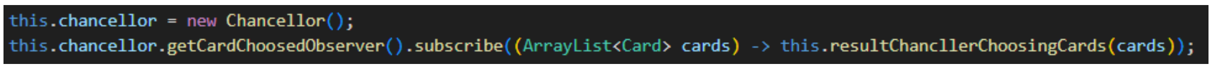
```java
class KillingPlayers extends Right {
    /** the game proxy */
    private GamePresidentAccess game;
    public KillingPlayers(GamePresidentAccess game) {
        super(Request.ChoosePlayer);
        this.game = game;
    }

    @Override
    public Object execute(ExecutionStatusWrapper executionResult, Object... params) {
        if (params.length == 1 && params[0].getClass() == Integer.class)
            return game.killPlayer(executionResult, (Integer) params[0]);

        executionResult.status = ExecutionStatus.NotChosenPlayer;
        return null;
    }
}
```

- Using nested classes and interfaces, used in UserData/Userdata to separate the visual data and easly send them without giving access to the whole obj



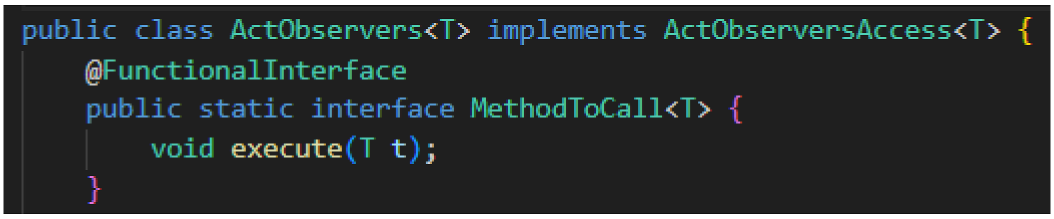
```java
/** stores the userdata of the player */
public class UserData {
    /** stores the visual data of the player, for outpside code only read methods */
    public class VisualData {
        /** name of the user */
        private String name;
        /** url of the icon if the player */
        private String iconImageUrl;

        private VisualData(String name, String iconImageUrl) {
            this.name = name;
            this.iconImageUrl = iconImageUrl;
        }

        /**return the name of the user
         * @return the name of the user
         */
        public String getName() {
            return name;
        }

        /**returns the url of the player icom
         * @return the url of the player icom
         */
        public String getImageURL() {
            return this.iconImageUrl;
        }
    }
```

- using lambda expressions or method references, used to create an observers and some functional interfaces
    1. lambda used in model/Game/Game/Constructor



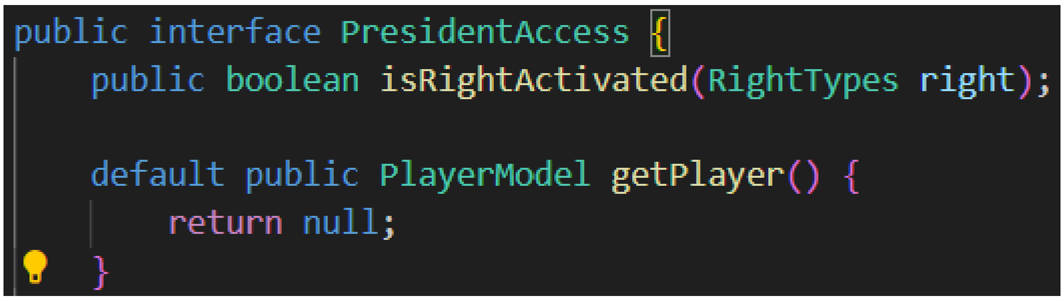
```
this.chancellor = new Chancellor();
this.chancellor.getCardChoosedObserver().subscribe((ArrayList<Card> cards) -> this.resultChanclerChoosingCards(cards));
```

    2. functional interface used in ActObservers



```
public class ActObservers<T> implements ActObserversAccess<T> {
    @FunctionalInterface
    public static interface MethodToCall<T> {
        void execute(T t);
    }
```

- default interface implementation used in model/ChangebleRole/PresidentAccess



```
public interface PresidentAccess {
    public boolean isRightActivated(RightTypes right);

    default public PlayerModel getPlayer() {
        return null;
    }
```