PLANET BOUND

ADVANCED PROGRAMMING PROJECT REPORT

ANTONI FORZPAŃCZYK
a2019156557

Abstract

This report presents the description of the design and implementation of board game called “Planet bound v2” regarding the requirements provided during Advanced Programming subject. Undermentioned work consists of a brief description of the options and decision taken during the process of implementation, description and graphical representation of required state machine model, class relationship explanation and additionally for each feature or rule of the game the indication of the implementation with given certain methods to prove yet clarify the whole design.

Options and decisions taken in implementation

Game

Class *Game* represents the gateway for PlanetBound usages ready to use in UI model. Every game method just invokes and returns the result of the state methods, logic method and certain game constraints.

State machine

I decided to represent state machine by the usage of polymorphism. The main core class *Game* stores the current state which extends class *State*. Every class implements methods from beforehand design state machine diagram through the *IState* interface. This approach restricts the usage of states in *Game* class only for the methods to operate on state machine itself.

Logic

Every state has its logic equivalent which operates on current data held in *LogicConfig* static class to has globally available certain objects such as Ship, current position of ship etc. Previously I decided to force communication between static logic classes however some data exchanged provided the confusion in code. Therefor I implemented the non-static logic classes which obtain the necessary for computation data from one common Singleton class *LogicConfig*. This provides the state to have new logic object without any leftovers from previous computations.

Ship

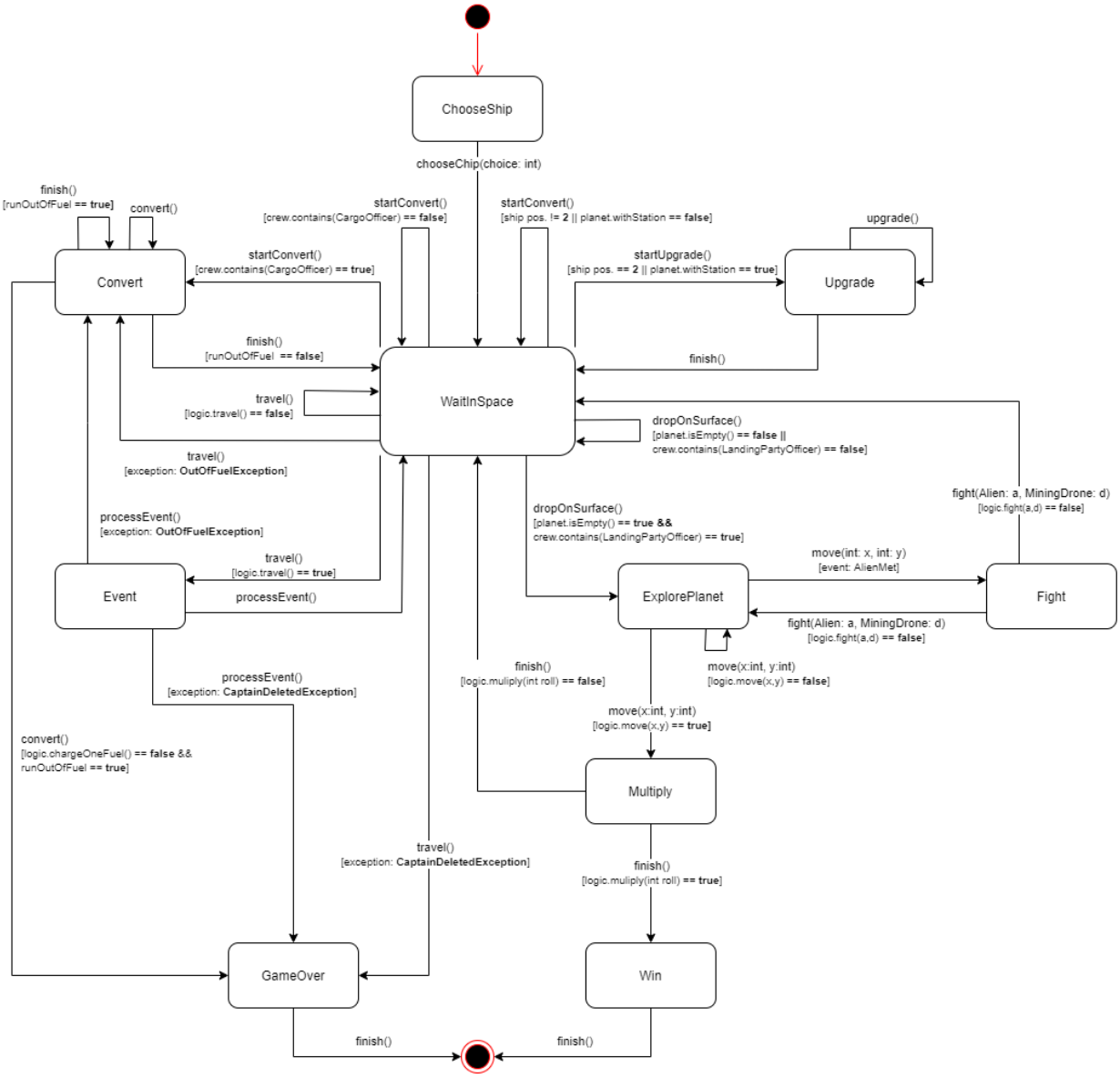
Ship itself has its own interface IShip which consists of methods used by logic and states in further computations. Furthermore ship equipment from game such as cargo, weapon system or shield has its own class representatives to provide its own functionalities through own interfaces like *IWeapon* and *ICargo*. I provided the polymorphism to divide equipment with its kinds for example *AdvancedCargo* and *BasicCargo* due to the diversity of initial attributes between them. Hence, the object Ship assures getters of certain ship equipment to access the functionalities from certain Interface. This approach is useful to notify UI module changes in some properties.

GlobalSender

Due to the necessity of having UI updated I needed to implement the mechanism to notify changes in *LogicConfig* class which is static. Consists of all receivers previously assigned to this class and stored in arrayList of objects implementing the *IGlobalReceiver* interface.

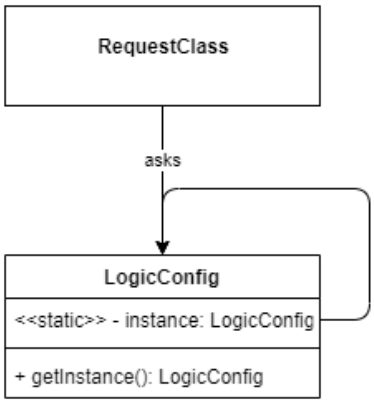
Programming patterns

State machine pattern <DAĆ TU OPIS>



Due to the poor quality I will enclose a diagram in svg extension to see it more in detail

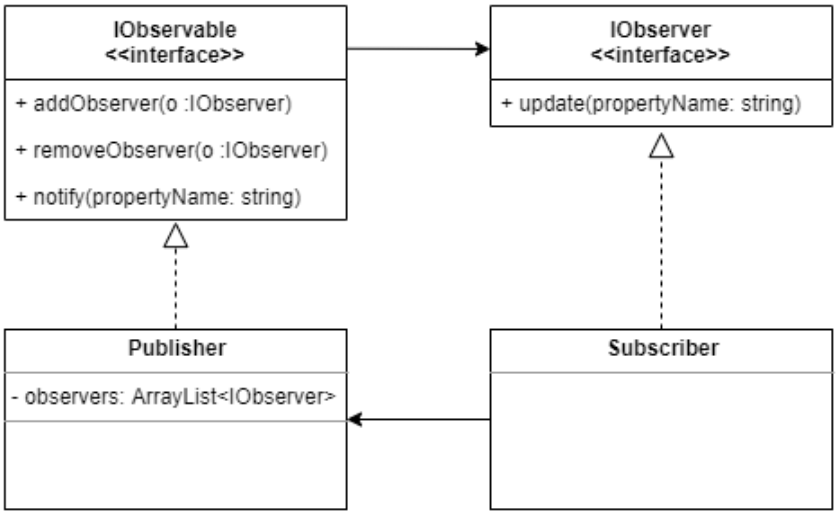
Singleton pattern



Provided pattern enables to instantiate the object by static method without creating its new equivalent. It provides the ability to store data globally but access it through variable by assigning the return constructor value by the getInstance() method.

```
private static LogicConfig instance;
    public static LogicConfig getInstance() {
        if (instance == null) {
            instance = new LogicConfig();
        }
        return instance;
    }
```

Observer pattern



I implemented the Observer programming pattern to notify the events in ExplorePlanetLogic class and process it in ExplorePlanet state class. In addition this will be used to notify changes and force update of the certain UI objects.

IObservable interface implementation

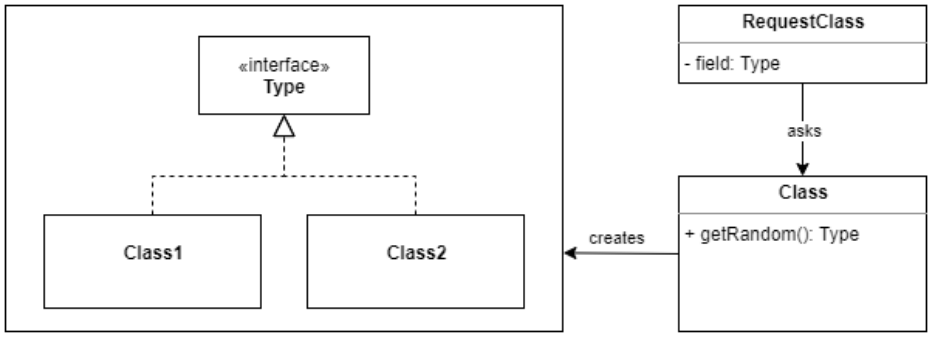
```
private final ArrayList<IObserver> observers = new ArrayList<>();
    @Override
    public void addObserver(IObserver observer) {
        observers.add(observer);
    }

    @Override
    public void removeObserver(IObserver observer) {
        observers.remove(observer);
    }

    @Override
    public void notifyChange(String property) {
        for (IObserver o : observers) {
            o.update(property);
        }
```

IObserver interface exemplary implementation

```
    @Override
    public void update(String property) {
        switch (property) {
            case "ResourceTaken":
                System.out.println("Resource taken");
                Game.setState(ExplorePlanet.getInstance());
                break;
            case "AlienMet":
                System.out.println("AlienMet");
                alienMet = true;
                Game.setState(Fight.getInstance());
                break;
            case "ExtractionPoint":
                System.out.println("ExtractionPoint");
                extractionPoint = true;
                Game.setState(Multiply.getInstance());
                break;
        }
}
```

Factory pattern

Planets such as aliens appear randomly in the game. The purpose of that pattern is to minimalize implementing random choosing repeatedly. Instead program contains classes with static method which returns already randomly chosen and adjusted object regarding rules of the game. My application contains PlanetFactory used to regenerate space object while travelling out the old one and also AlienFactory to obtain random alien at the surface of the explored planet by the mining drone.



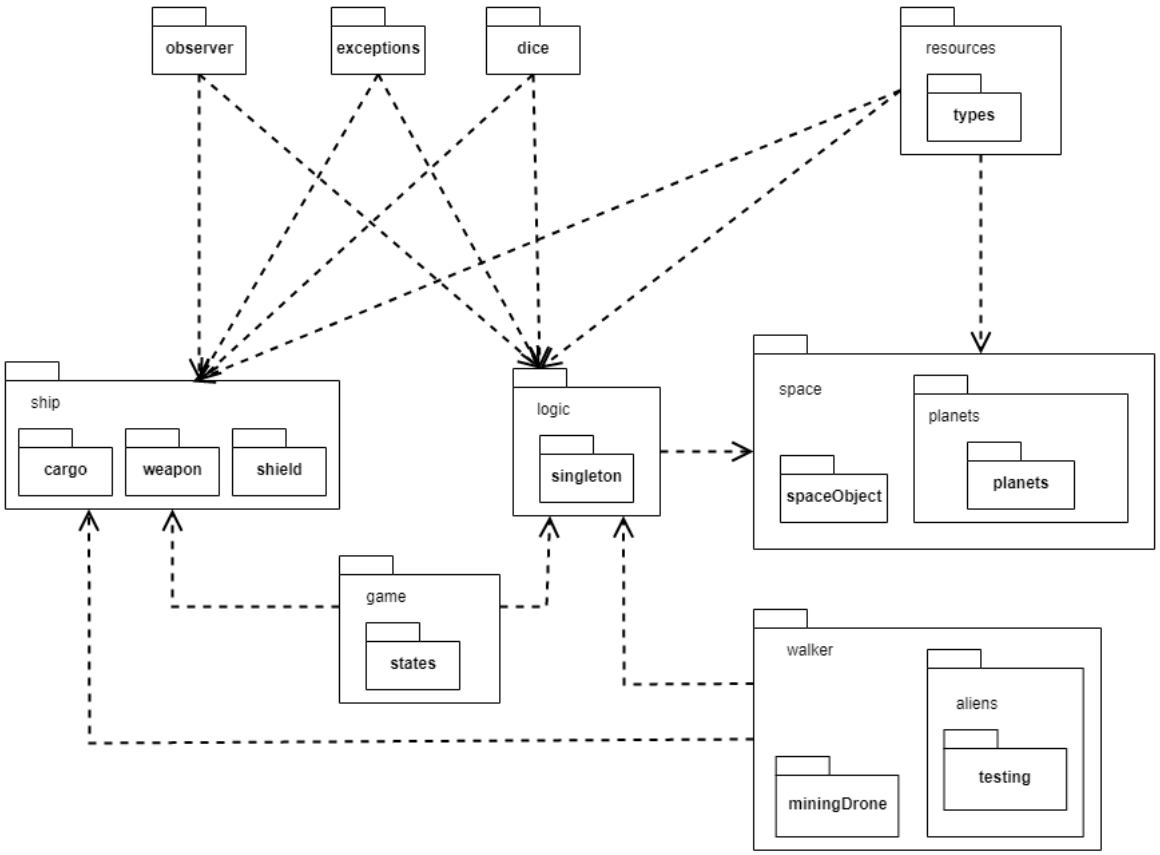
PlanetFactory exemplary implementation:

```
static public Planet getRandomPlanet() {

        Planet planet = null;
        Random random = new Random();
        switch (random.nextInt(4)) {

            case 0: {
                planet = new BlackPlanet();
            }
            break;
            case 1: {
                planet = new RedPlanet();
            }
            break;
```

```
            case 2: {
                planet = new BluePlanet();
            }
            break;
            case 3: {
                planet = new GreenPlanet();
            }
            break;
            default: {
                System.out.println("WRONG PLANET ID");
            }
        }

        //generate space.planet 70% without station and 30% with one
        int pChance = random.nextInt((100 - 1) + 1) + 1;

        if (planet != null) {
            if (pChance <= 30) {
                planet.setStation();
            }
        }
        return planet;
    }
```

Class description

Classes are divided into packages to provide better clarity in organizing the code. Below is placed package diagram to have better vision on the structure as well as the dependencies between them.



Dice package

Class name	Usability and purpose
Dice6	It is a representation of D6 dice and with method roll() it returns random number from 1 to 6.
Dice3	Equivalent of physical dice D3. By the method roll() returns random number from 1 to 3;

Exception package

Class name	Usability and purpose
CaptainDeletedException	Exceptionn thrown when captian is deleted. Specifically, when ship method looseOneCrewMember() is invoked while crew arrayList has size of 1.
OutOfFuelException	Exception thrown when ship has no fuel left. In detail, when ship's class method consumeFuel() is invoked and fuel will reach 0 the exception is thrown.
UnavailableException	Exception thrown when method from IState in states classes is not implemented, it is not available to perform in that state. It is used to provide the security in state machine flow.
WrongArgumentException	Exception thrown in default block in switch in methods implemented form IState interface. It helps to checks if all choices and arguments are passed correctly.

Game package

Class name	Usability and purpose
Game	This is the core of the application. It contains current state and methods which represents the IState methods signatures and returns the same values. It is used as the gateway of the backend of the application. It is public class with static methods especially made for UI connection to operate on game and state machine itself.

States package

Class name	Usability and purpose
IState <<interface>	This interface contains all methods used to operate on states in state machine. All methods returns boolean value which represents the result of used functionality.
ChooseShip	Representation of ChooseShip state. It implements method chooseShip(choice: int) which sets ship in globally accessible LogicConfig singleton class and changes the state to WaitInSpace.
Convert	Representation of Convert state. It implements two methods: convert(choice: int) and finish(). First one converts the resources into certain objects like new mining drone or new fuel. Method convert(int: choice) resets the Convert current state. Second one 's purpose is to exit the Convert state and enter the WainInSpace state.
Event	Representation of Event state. It implements only one method – processEvent(). Firstly, the event is randomly chosen and then processed. Depending on the result it usually returns to WaitInSpace state however when exceptions are thrown it can switch to Convert state or directly to GameOver state.

ExplorePlanet	Representation of ExplorePlanet state. It is used in aim of moving drone on the planet surface. It has implemented one method from IState interface which is move(x: int, y:int). This method is needed to move the drone in specific directions in behalf of move method in ExplorePlanetLogic class. In addition this class implements the beforehand mentioned Observer Patter. ExplorePlanet listens and processes the events sent from ExplorePlanetLogic class.
Fight	Representation of Fight state from state machine. It's responsibility is to cope with fight mechanism through FightLogic class. Depending on the boolean result which plays the role of the result of the fight. If drone has been destroyed it enters WaitInSpace state in other case it game continues the exploration of planet in ExploreSurace state.
GameOver	Representation of GameOver state in game. It has implemented method finish() which exits the application.
Multiply	Representation of Multiply state. It has implemented method finish() to take current resource collected by drone and load random number (Dice6.roll()) to ship's cargo unless artefact is collected. Additionally, it checks the amount of the artifacts in ship's cargo and in the situation when the collected artefact is the fifth one, it changes sate to Win state. In other case it just returns to WaitInSpace state.
Upgrade	Representation of Upgrade state in state machine. It has two methods implemented: upgrade(choice: int) and finish(). First one represents the functionalities from game while staying on the planet with space station. Second one is used to exit the state and switch to WaitInSpace one.
WaitInSpace	Representation of WaitInSpace state. Generally it is most switched and used state in whole state machine. It has implemented couple of methods: startConvert() switches state to Convert if ship has CargoOfficer on board; startUpgrade() switches state to Upgrade if ship is next to the planet with space station; DropOnSurface() switches the state to ExplorePlanet provided the ship is next to planet, LandingPartyOfficer is included into crew and there are any resources left on planet. The last method is travel() which shifts the position of the ship and forces generating new space sectors. Depending on the position of the ship it can return to WaitInSpace state or enter Event state when one is encountered.
Win	Representation of Win state in state machine. It places the role of thein the game. Basically, It has implemented one method – finish() which simply exits the application like GameOver state does.

Logic package

Class name	Usability and purpose
ConvertLogic	The aim of this class is to provide logical background of Convert state class. This set of four methods which firstly checks whether player satisfies the requirements to perform certain action in Convert state of the game.
EventLogic	When ship encounters the event in space this class provides functionalities to select event type randomly as well as process it depending of previously selected id.
ExplorePlanetLogic	Provides functionalities for ExploreSurface state class to move drone, alien, set randomly positions of extraction point coordinates as well as the

	resource's one. This class is connected to ExploreSurface state through the Observer pattern to notify the events during movement.
FightLogic	Provides functionalities for Fight state class to process outcomes of random Dice6 rolls in alien and drone attacks. It has one method fight() which depending on the result returns different Boolean values.
MultiplyLogic	Provides logical background for Multiply state class to load random number of resource found on the planet by mining drone. Also, if artefact is collected, it checks the quantity of that resource in the ship and notifies the state of that.
UpgradeLogic	Provides logical background for Upgrade state class to process all functionalities available for the player while staying next to the planet with space station.
WaitInSpaceLogic	This class plays role of the logical core switching between most of the states in the game. It checks the availability of entering the Convert state, Upgrade state, Explore state and also provides the functionality to move the ship in the space considering the encountering the space event.

Singleton package

Class name	Usability and purpose
LogicConfig	This class place role as global container for all necessary data for all logical state backgrounds. It consists of all objects processed in computations and comparisons through whole game lifetime.

Observer package

Class name	Usability and purpose
IObservable <<interface>>	Interface prepared for publisher class .It provides the set of method necessary to establish the connection between subscriber and publisher from Observer programming pattern.
IObserver <<interface>	It provides the methods for subscriber to invoke the certain functionalities from publisher side.

Resources package

Class name	Usability and purpose
IResource <<interface>>	Provides one necessary method for resource type comparison. Generally it's aim is to gather all resources under one class type.
ResourceType <<enum>>	Contains all available types of resources in the game. It is used to check the real type of the resource among bunch of IResource objects. It used enum type attribute stored in each of inherited IResource class to avoid class comparisons which would lead to confusion in code.

Types package

Class name	Usability and purpose
Artefact	Represents the artefact form the game.
BlackResource	Represents black resource form the game.
BlueResource	Represents blue resource form the game.
GreenResource	Represents green resource from the game.
RedResource	Represents red resource form the game.

Ship package

Class name	Usability and purpose
CrewMembers <<enum>>	It is enum which represents all available crew members in game. It is used in decisions whether player can have access to certain funcions such as dropOnSurface(), startConvert(). Ship class contains the List of the type CrewMembers representing current crew capacity.
IShip <<interface>	Provides all ship functionalities used in the game. It also provides the visible place to notify the change in the ship attributes' values processed by the UI in the future.
MiningShip	Represents the mining ship from the game.
MilitaryShip	Represents the military ship from the game.
Ship <<abstract>>	This class implements all methods from IShip interface, consists all common attributes inherited by MiningShip or MilitaryShip class.

Cargo package

Class name	Usability and purpose
AdvancedCargo	Represents advanced cargo hold from mining ship from the game.
BasicCrgo	Represents basic cargo hold from mining ship from the game.
Cargo <<abstract>>	This class implements all caro functionalities used in the game. As IShip interface it is also the prepared place to notify the changes thought the Observer pattern.
ICargo <<interface>>	Provides all necessary cargo methods invoked during all game lifetime.

Shield package

Class name	Usability and purpose
IShield <<interface>>	The purpose of this interface is to provide necessary and widely used functionalities for shield component of the ship.
Shield	Represents the shield component from the game.

Weapons package

Class name	Usability and purpose
AdvancedWeapon	Represents the weapon after upgrade for military ship
BasicWeapon	Represents basic weapon available for mining ship and military ship at the beginning of the game.
IWeapon <<interface>>	The purpose of this interface is to provide necessary functionalities for weapon component of the ship.
Weapon <<abstract>>	Consists all common attributes form derived classes. It also implements methods from IWeapon interface.

Space package

Class name	Usability and purpose
SpaceSector	This class represents the space card from the game adjusted to assignment requirements. It consists of Entrance, Event, Planet, Wormhole or WayOut class. The purpose of this entity is to operate on ship depending where it is in space void.

Planet package

Class name	Usability and purpose
IPlanet <<interface>>	This interface consists of the methods needed to operate on ExploreSurface state.
Planet <<abstract>>	This class is the base class for deriving planets classes of different type from the game's rules. It consists all common attributes and implements methods from IPlanet interface.
PlanetFactory	Class performing usability from Factory programming pattern used for obtaining random planet with randomly set attributes following assignment requirements.

Planets package

Class name	Usability and purpose
BlackPlanet	Representation of a black planet from the game
BluePlanet	Representation of a blue planet from the game
GreenPlanet	Representation of a green planet from the game
RedPlanet	Representation of a red planet from the game

SpaceObject package

Class name	Usability and purpose
Entrance	Representation of an entrance (left side of the space card) into the space sector
Event	Representation of a event on the space card form the game.
ISpaceObject	Consists of the method for exploiting the ship's resources such as fuel and shields while travelling through the space object on the space card in the game.
RouteOut	Representation of normal space route out the space sector (straight white line to the right side of the space card)
Wormhole	Representation of wormhole on the space card.

Walker package

Class name	Usability and purpose
Coordinate	Representation of the coordinates (x,y) used for moving mining drone and alien on surface.
IWalker <<interface>>	Consists of methods used for those classes representing walking objects on the planet surface.
Walker <<abstract>	This class contains all common attributes for Alien and MiningDrone class and implements methods imposed by the IWalker interface.

Alien package

Class name	Usability and purpose
Alien <<abstract>>	This abstract class is used in the polymorphism to be extended by various types of alien representatives: BlackAlien, GreenAlien, RedAlien, BlueAlien class.
AlienFactory	The aim of this class is to give random, entirely prepared, new alien.

Aliens package

Class name	Usability and purpose
BlackAlien	Representation of a black type of alien from the game.
GreenAlien	Representation of a green type of alien from the game.
RedAlien	Representation of a red type of alien from the game.
BlueAlien	Representation of a blue type of alien from the game.

Testing package

Class name	Usability and purpose
BadassAlien	For a testing purposes I needed to have alien which will always win.
WeakAlien	For a testing purposes I needed to have alien which will always loose.

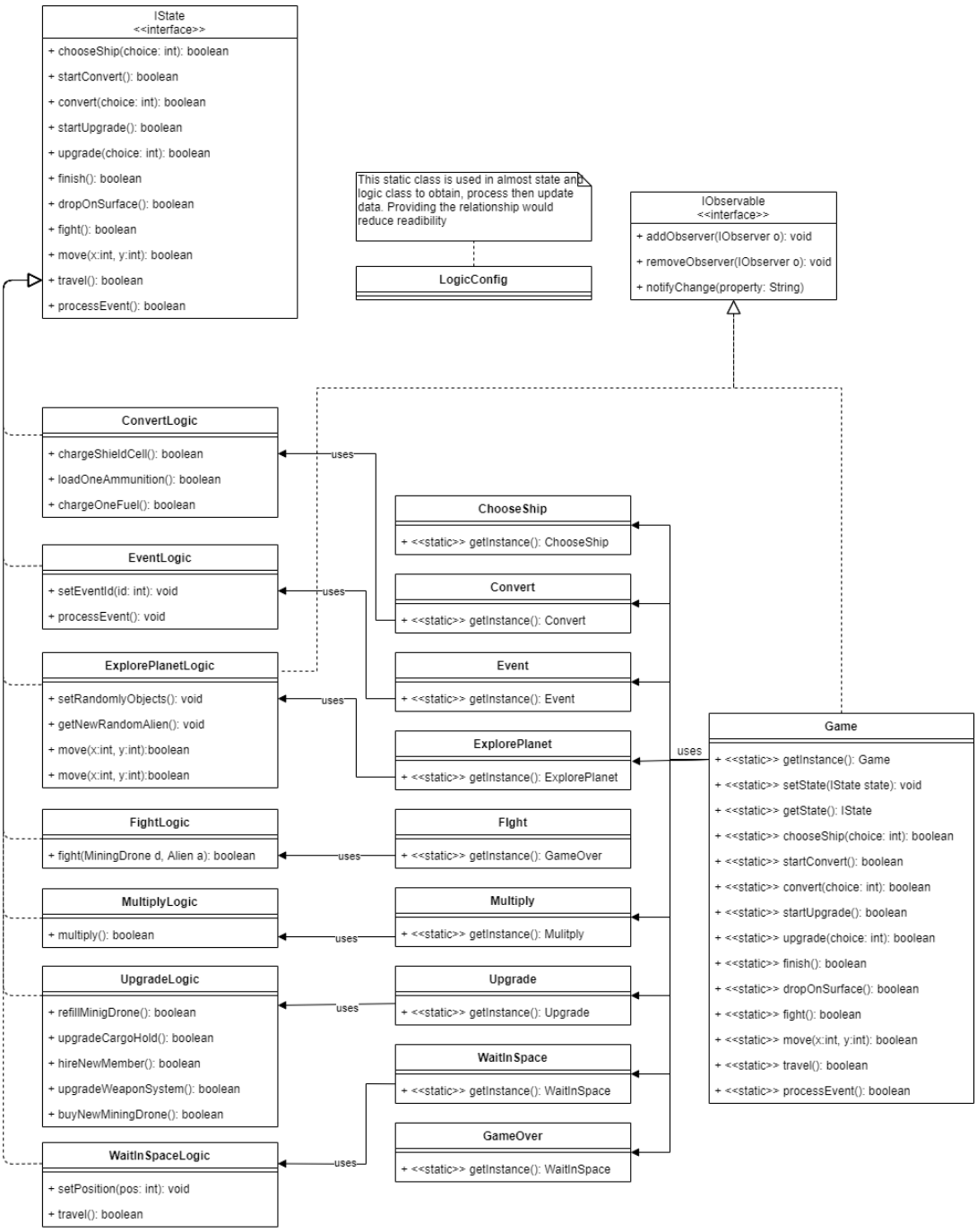
MiningDrone package

Class name	Usability and purpose
MiningDrone	Class representing functionalities of mining drone from the game. It is stored in ship class however it is in walkerPackage due to commonly used methods alike alien class.

Class relationships

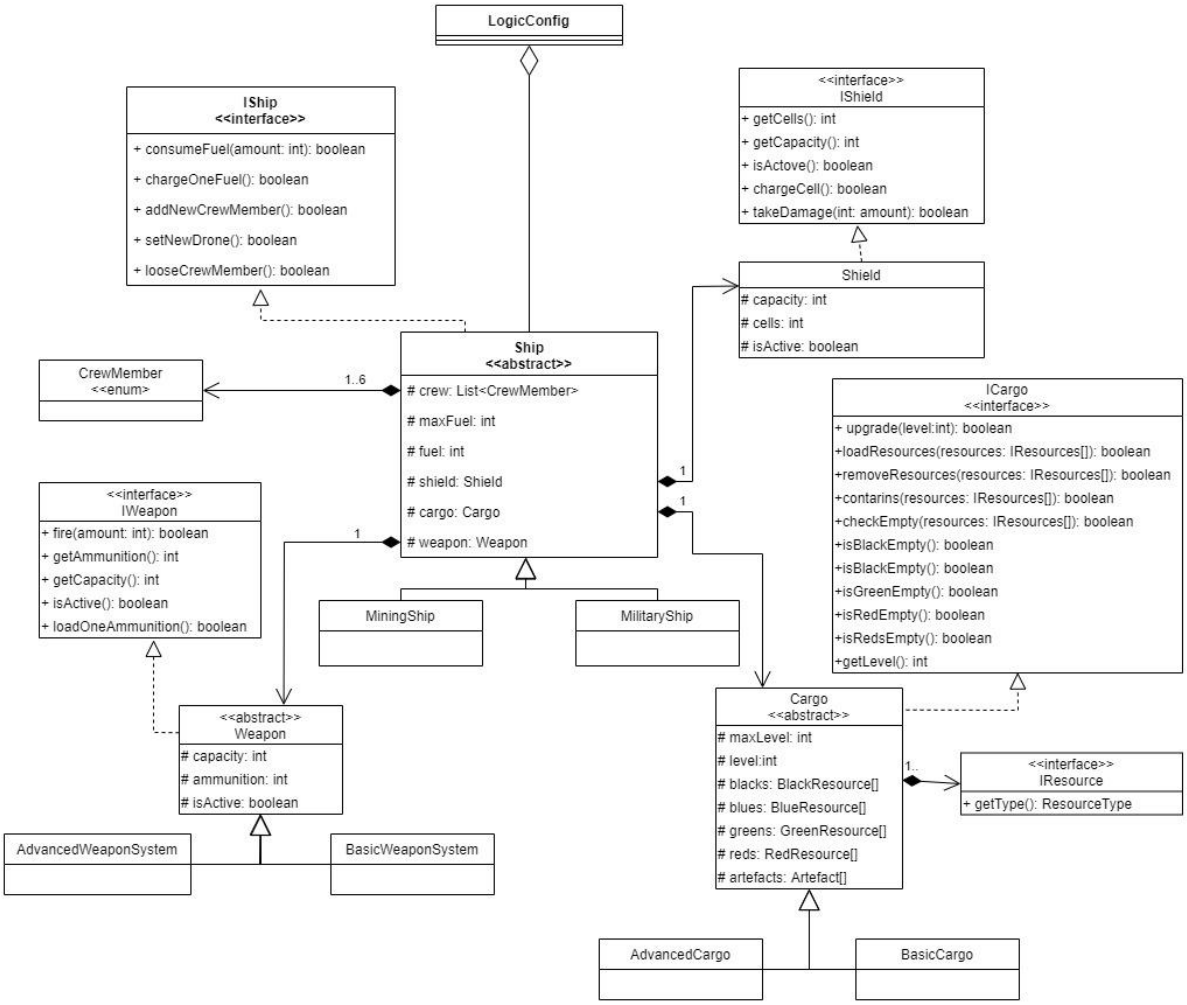
State machine with logic

Presented below diagram represents relationship in the state machine core of application. States implements IState interface and use logic classes to switch between states properly. Some classes to perform better communication implements IObserver and IObservable interface. Class LogicConfig stores all necessary data for the application and its instance is used among almost all classes that is why it is not connected in the diagram in order to sustain readability of the diagram. In addition, all state classes throws exceptions to provide security aspects with data flow as well as cover critical situations for game rules.



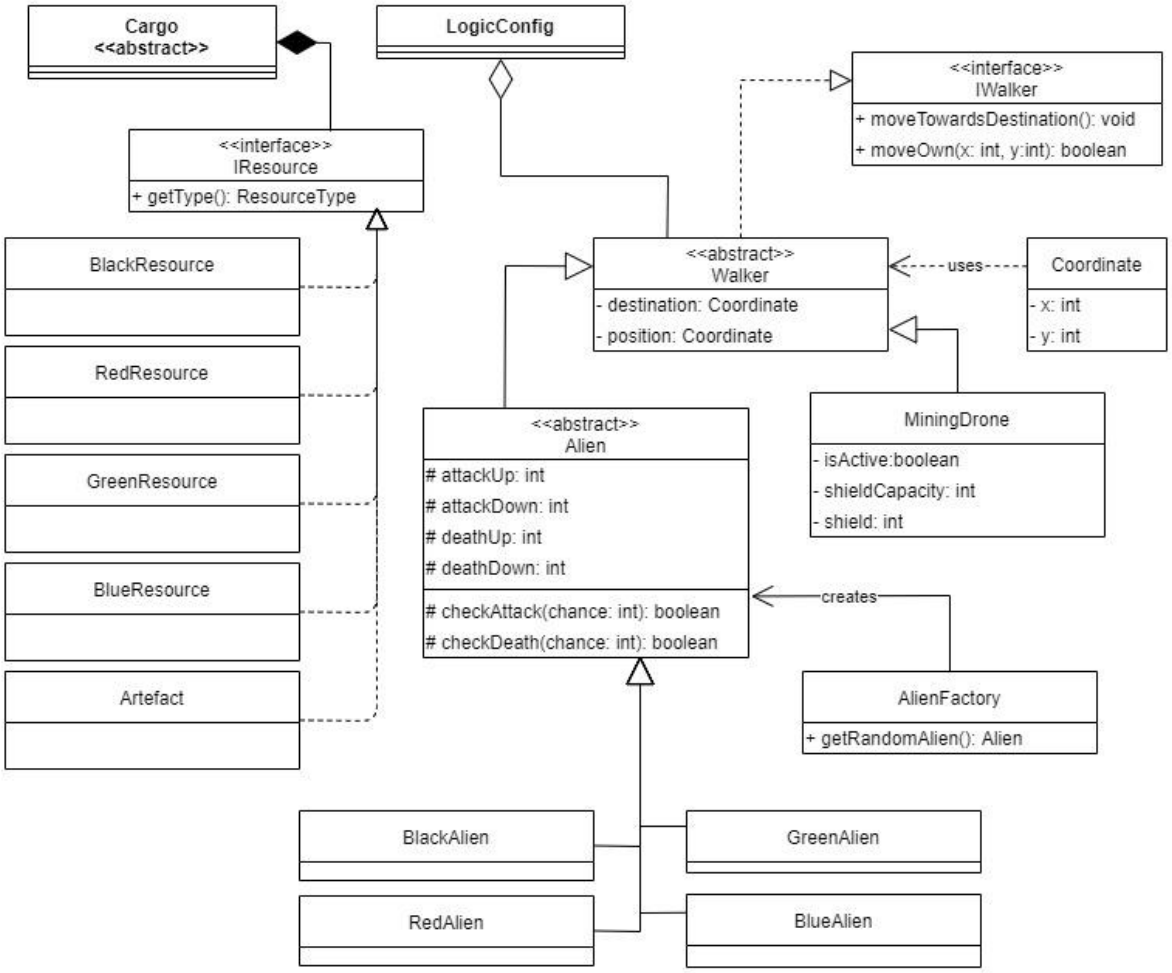
Ship

Class diagram presented below describes ship design. Ship class implements IShip interface providing necessary and widely used functionalities for that class. Ship class is derived by classes MilitaryShip and MiningShip. Ship contains and uses classes such as: Shield, Cargo Weapon which respectively implements its own interfaces and if is necessary derive from basic abstract class. Whole Ship class is stored in LogicConfig class which provides global access among while application to operate on.



Resources and walkers

This diagram presents composition and design of resources and surface walkers (regarding rules form the game). All resource representatives implements IResource interface. They are stored in Cargo class from Ship. Walker abstract class is derived by Alien and MiningDrone. Interface IWalker provides functionalities and methods to operate on coordinates while moving on planet's surface in ExploreSurface state.

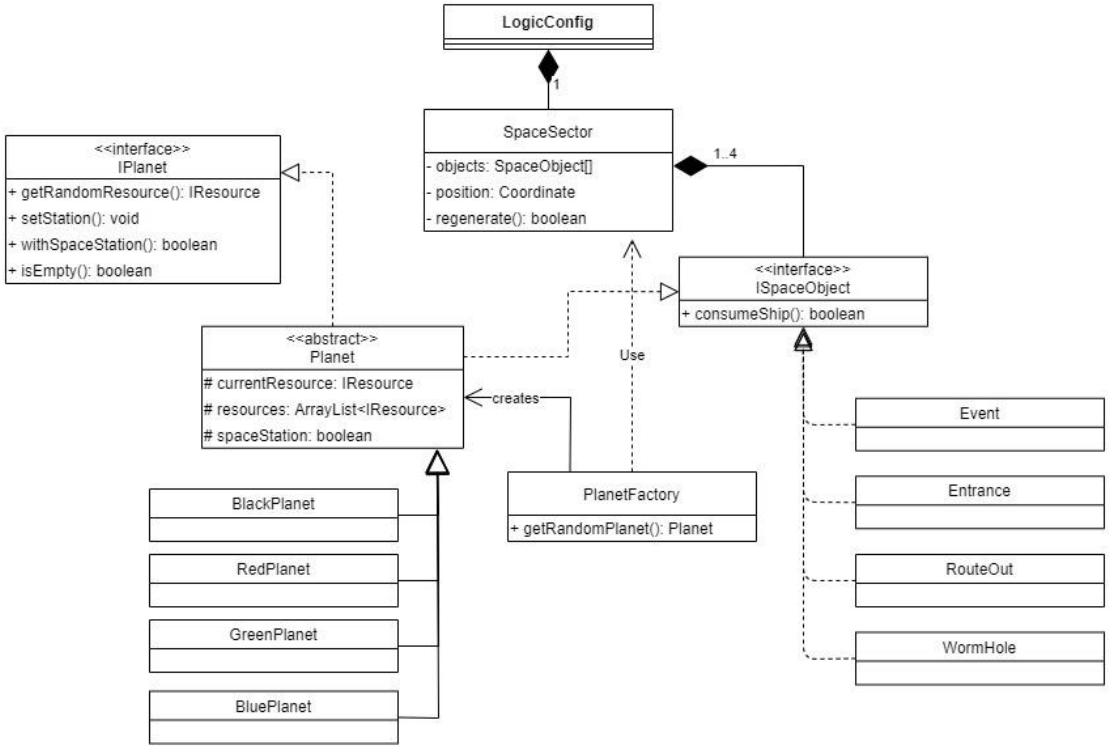


Space

Space class diagram presents the structure of classes responsible for positioning and providing methods concerning the travel though space as well as data storage for next ExploreSurface state.

IPlanet interface is implemented by Planet class hence Planet is extended by deriving classes such as: BlackPlanet, RedPlanet, GreenPlanet and BluePlanet which differs from each other regarding game's rules.

Event, Entrance, RouteOut and Wormhole are classes which represents other components from space card from the game.



Status of implementation

Feature	Status	Testing (Junit5)
Choosing the ship	Implemented	Fully tested
Proceeding to the next planet	Implemented	Fully tested
Drawing and resolving event	Implemented	Fully tested
Possibility to land on planet	Implemented	Fully tested
Possibility to land on planet again	Implemented	Tested but in GUI I have problem with old listeners which try to change state. (I need to cope with finalize() method)
Movement on the surface of the planet	Implemented	Fully tested
Drawing and spawning aliens	Implemented	Tested
Aliens moving towards drone	Implemented	Fully tested
Fighting with aliens	Implemented	Fully tested
Buying resources	Implemented	Fully tested
Upgrading systems on station	Implemented	Fully tested
Win and lose conditions	Implemented (not in GUI yet)	Fully tested
Possibility to play again	Not implemented	Not tested
GUI (without winning and losing view)	Implemented	-
Log system	Not implemented	Not tested