2007

War Simulator:

ROBOTOTALWAR

Final Report

This document is prepared for Software Engineering Course project; War Simulator: RoboTotalWar. In this document you will find the final step of our Software project.

Erol TOKALAÇOĞLU
Osman A. BIÇAKCI
6/13/2007
Submitted to: M. Borahan TÜMER, Ph.D.

1. Table of Figures

2. Requirement Analysis Part

2.1. Introduction

RoboTotal War is a war simulator that enables to the users in first step establishing war simulations as defining robot teams and the environment that the robots fighting. For the future aspect we try to add the environment step that users also write their own robots that are extends our ancestor robot that has some specifications for adapting the environment that we defined before.

2.1.1. Purpose

This document provides detail surrounding the functionality of the RoboTotal War simulator software. Furthermore, the functional and non-functional requirements are discussed in depth. The document illustrates the functionality of the software through the use case diagrams. System constraints and required software and hardware are presented emphasizing the contribution to the system as a whole. Finally, a time plan for design and implementation is provided.

2.1.2. Scope

RoboTotal War is mainly working by computer itself but adjusting the teams, maps, conditions gives user to effect game. In the later steps, adding own robots and the learning robots gives user to fight his own robot with others. The subsystem is the environment that enables the user to write the java code with extending our robot class or for security reasons just adding the library that we defined for this project. This project gives chance to users that they can write their own robots with AI for better wars, as like robocode but the difference between our platform and robocode is we define the rules for robots and we give some specifications and and the limits are restricted.

2.1.3. Definitions, Acronyms and Abbreviations

RoboTotal War simulator we define first an environment that enables user to fight the robots and we give up the name EnWar for the environment. We have types of robots, and every robot has a defined specific features and for future aspect user can define more within the limits and environment rules. We will use an IDE (INTEGRATED DEVELOPMENT ENVIRONMENT) for designing GUI(GRAPHICAL USER INTERFACE). Mainly we think to write our code in JAVA Language for using the features of OOP(OBJECT ORIENTED PROGRAMMING) that enables us to extend new robots from the ancestors and for user part for future user can also define his own robots as like we do now. Also Java gives us the chance to run our environment in all OS(OPERATING SYSTEM) platforms as like LINUX, WINDOWS, SOLARIS, MacOSX etc. For future user can easily understand our platform with using API(APPLICATION PROGRAM INTERFACE) that enables user to see which classes we have and which relations that they have.

2.1.4. Overview

Section 1.2 of this document provides an overall description of the Software system, including the main functionality and ideas.
Section 1.3 illustrates the main functionality of the system using several simple activity diagrams and use case diagrams.
Section 1.4 explains the functional requirements of the system in detail grouped in order of importance by each of the components mentioned in Section 1.3.
Section 1.5 presents the non- functional requirements of the system. Section 1.6 User Interface that we develop. Section 1.7 contains the constraints we would be having in the making of our system Section 1.8 lists the hardware and the software to be used. Section 19 finally has the references we have used in our research followed by the glossary written.

2.2. Overall Description

2.2.1. Overview of the System

Our environment works on JAVA. The system includes two steps: first is user interface to adjust the specifications for the game and the second is starting the game. The game means autonym working robots that are hard coded by us and for future aspects we try to give chance to end user for designing own robots but our project in first step gives user to adjust conditions, robots, maps etc. before starting game.

2.2.2. Thesis Statement

RoboTotal War simulation software's main aim is to bring end user to define own robots within the limits of our environment. That means end user will use own hardcoded robots, as using the AI algorithms but actually in the growing procedure of the system shows that the first steps of the environment required a unhandled robot environment. End user uses only the robots that we define for simulation. But adjusting is also a big constraint for the end user to change the way of the war.

2.2.3. Product Functions

The product functionality can be divided according to the components that make up the system which include the user and the robot component.

User Functionality: User can compose the system and adjust the features of the game simulation. Can either handed the general options and game options, will start, save, open the simulator.

Robot Functionality: System gives robots to fight automatically but the before starting the game they are hardcoded. The user handle them by GUI and this changes are mainly seem in simulation.

2.3. Uses Cases and Activity Diagrams

2.3.1. End User Use Case

End user can adjust the environment for the next game. The features of the main functionality can be adjusted by the Simulator options menu and the game options are seperated mainly 4.

- Game general options
- Match Type
- Map Options
- Robots and creating Teams

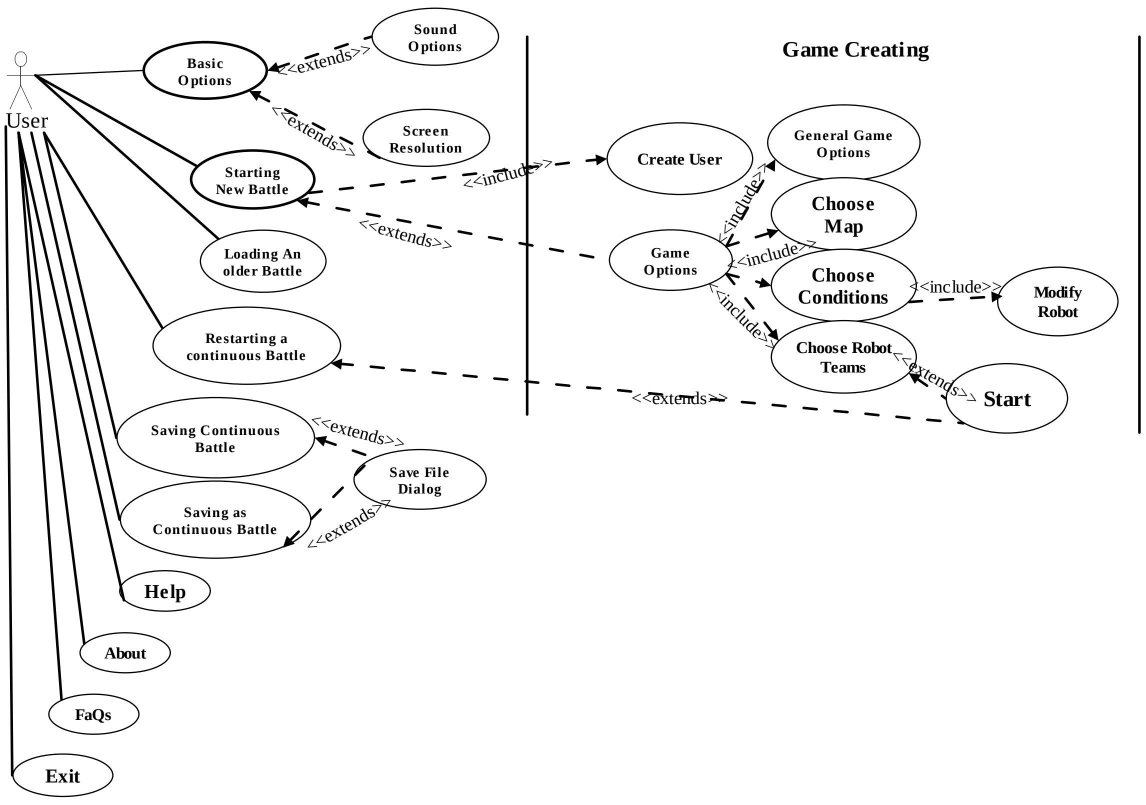


Figure 1.3.1 1: Use-Case of End User

2.3.2. Robot Use Case

Robots have lots of functionality. They are hardcoded system droids that they work by themselves. User only effect the conditions, team members, the guns, maps etc.

The other parts are worked by the robots and their functions are defined as like below use case.

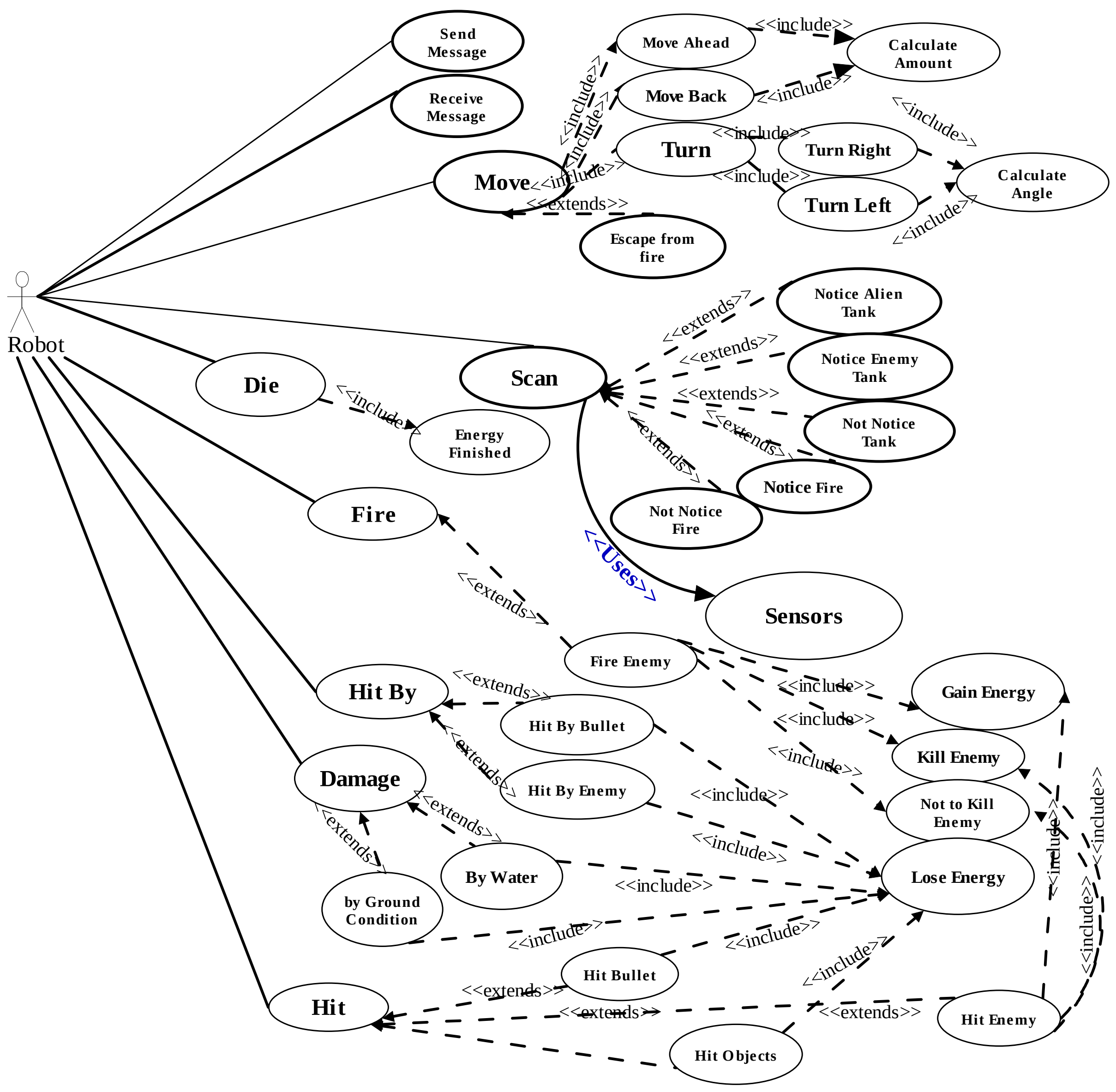


Figure 1.3.2 1: Use-Case of Robot

2.3.3. State Activity Diagrams

User can control the main environment. So user has the ability to create game, open game, restart game, save and save as game, exit game and adjust the main functions for the environment.

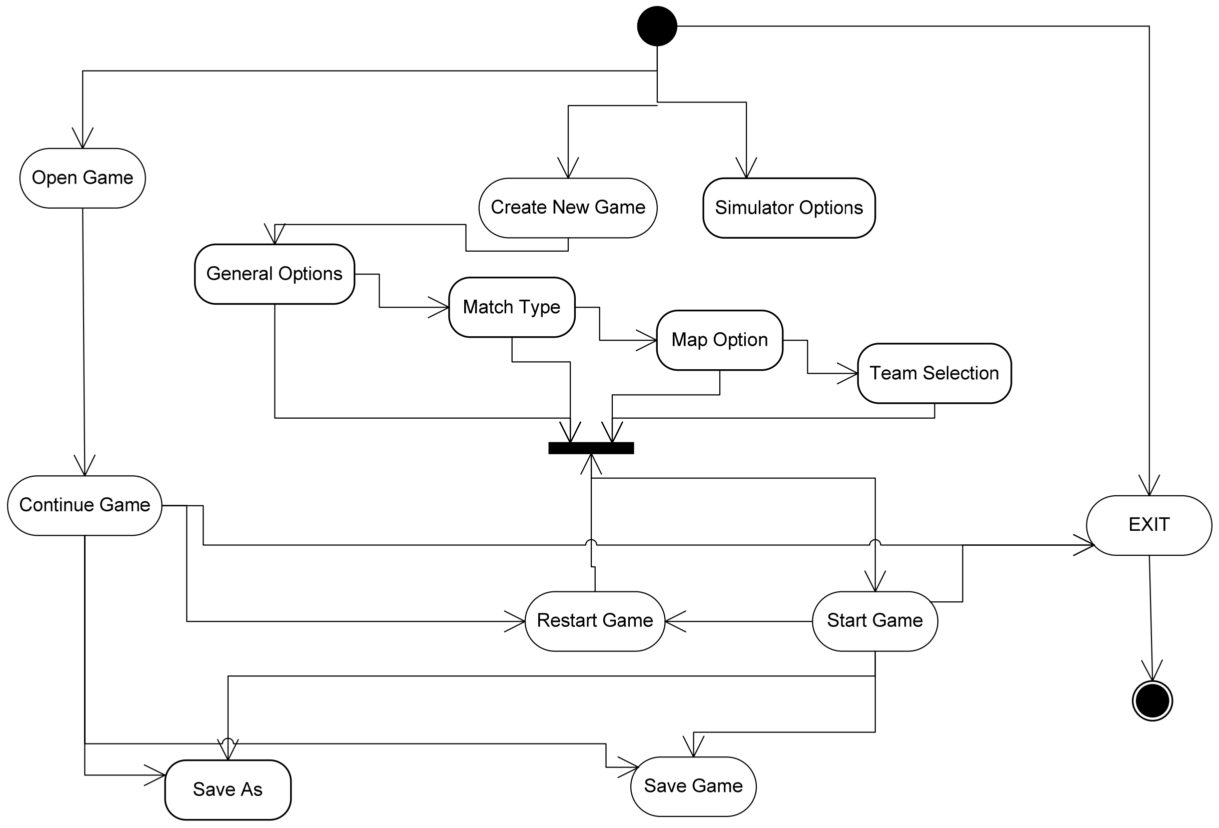


Figure 1.3.3 1: State/Activity Diagram of User

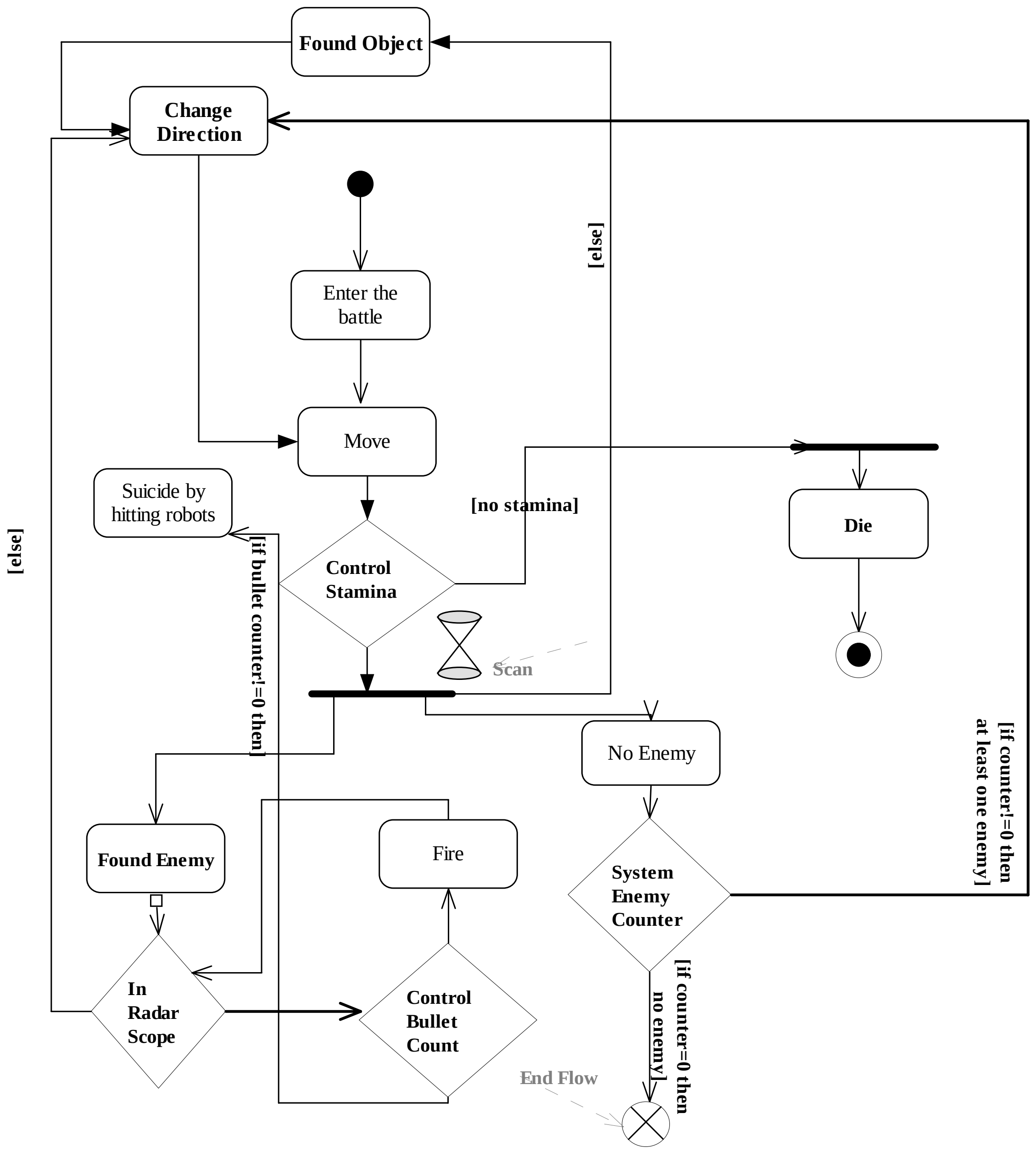


Figure 1.3.3 2: State-Activity Diagram of Robot

This is the lifecycle of the robot. Robot has the ability to work by itself but the communication skills are also added by the environment. After user adjusts robot fight in the conditions of the environment. The rules are defined by Robot class and this is our ancestor robot it has the defined and non changeable functions.

2.3.4. Sequence Diagram

This is the timeline and objective sequences throughout the robot.

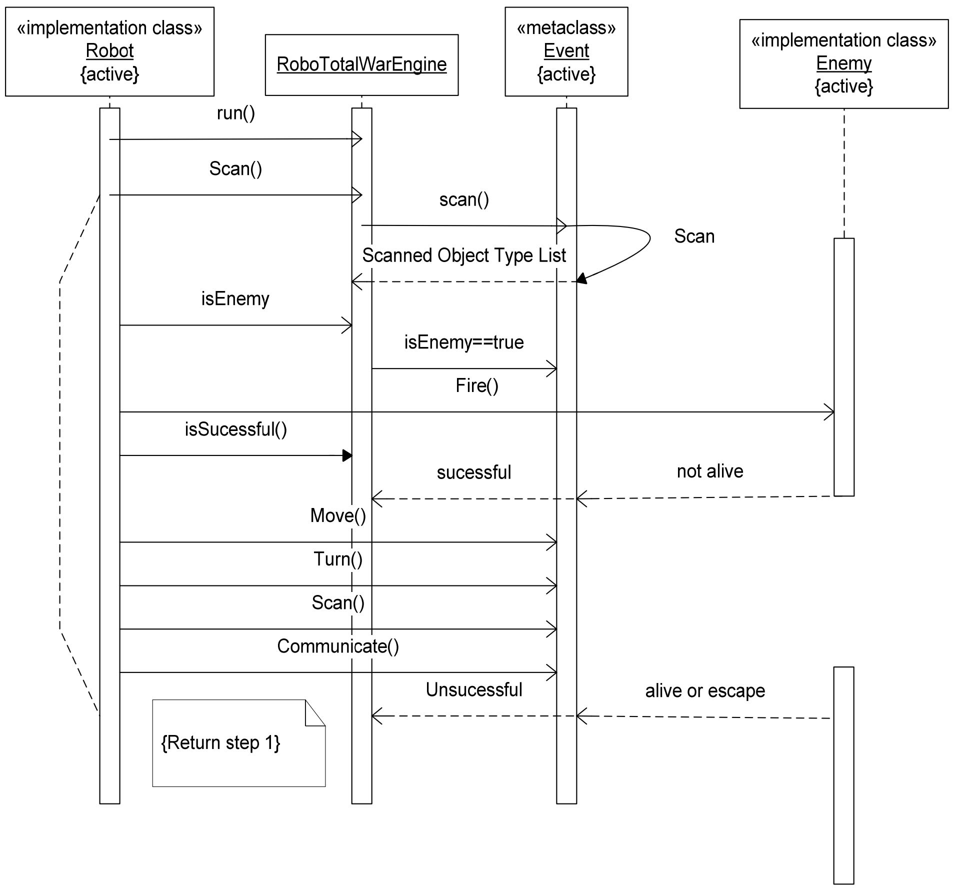


Figure 1.3.4 1: : Sequence Diagram 1 of Robot

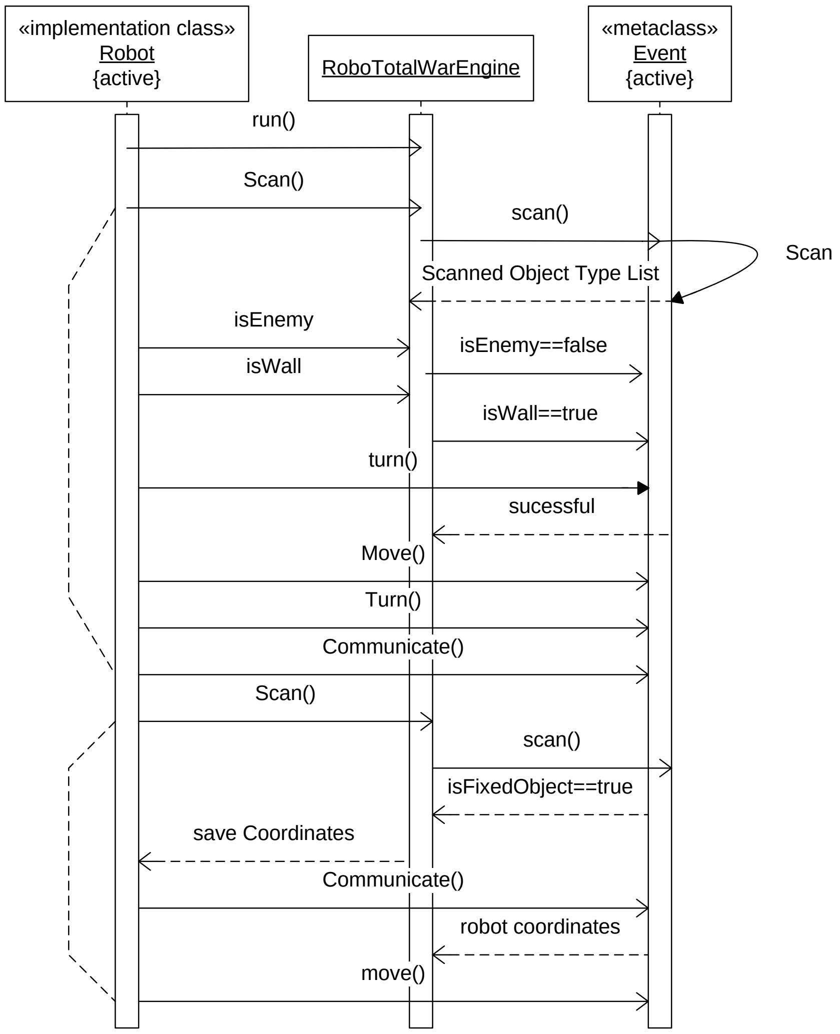


Figure 1.3.4 2: Sequence Diagram 2 of Robot

2.4. Functional Requirements

2.4.1. Environment

2.4.1.1. Board

2.4.1.1.1. Borders

There are exact limits that they are seen but actually there are limits for extreme conditions to stop escaping or abnormal actions.

2.4.1.1.2. Shape

The limits are seems as like rivers, mountains or other enviromental limits that the robots can not break the limits in real too.

2.4.1.1.3. Surface

The board is in a defined shape as a rectangle, for different screen resolutions. And it is in the center of the screen**.**

2.4.1.1.4. Topology

The surface is in 2D so there are no 3D objects in the surface but there are mountains, rivers then the type of the surface will be vary as like: stone,water areas.

2.4.1.2. Conditions

The shape of the map and the things on the map are whole form topology. The surface area is 2D but actually the effects over the enviromental things are the same as real so there is a topological surface. For example a robot cannot go over on mountain or sea, it must to stop and determine another direction for moving.

2.4.1.2.1. Light Condition

Light conditions are in general two types as day and night, also we set conditions to effect the robots' vision features oriented with light conditions.

2.4.1.2.2. Weather Condition

Weather conditions are in general sunny rainy, snowy.The weather condition effects the robots as like enviromental conditions, block or dicrease the attacks etc.

2.4.1.2.3. Ground Condition

Oriented with weather conditions the types of ground will be varied, if the weather is rainy then the soft sand is going to be swampy and robots will spend energies in moving.

2.4.2. Capabilities

2.4.2.1. Fire

2.4.2.1.1. Types

In a instance there are 3 types of fires.

The instance types of fires are;

- Pistol (lower speed, small effect, low cost)
- Laser(middle speed, big effect, highest cost)
- Plasma (higher speed, big effect, high cost)

2.4.2.1.2. Damage Types

The main idea should be defect the enemy robots. The damages should be in the types of;

- Only one robot can be effected
    - Should lose power
    - İf have, should lose first Shield than power in an order

2.4.2.1.3. Range

Actually the range of the bullets are the same just the speed is different, and for view parts it is working as targeting and getting results in a fixed range. The range is only vary with the condition of the battle, if it is nearly to be complete than the range is larger to finish in a small time instance.

2.4.2.1.4. Cost of Fire

There will be types of fires then the effects of these fires should not be the same also now we should know that the cost of fires must be different, but for owner of the fire it means nothing just for the hitted robot the cost means as a constant to delete power or shield.

2.4.2.1.5. Speed of Bullet

There must be a speed of a bullet, to catch the robots because every robot has a speed of motion and there will be fires to effect them or for escaping skills there must be a speed it also depends on the types of fire.

2.4.2.2. Motion

2.4.2.2.1. Angular

Robots have the capability to move with a angle in 2D plane, it helps them to find any object or sensing all map view.

2.4.2.2.2. Linear

Linear motion is the capability to move in a linear way.

2.4.2.2.3. Speed of Motion

The motion has a speed, it also controlled by the system users need as shifting the scroll on the robot speed part, and the main speed is the same with every each other robot.

2.4.2.2.4. Distance

Robots are fighting in a 2 dimensional plane so they have to move from a point to another point that has a vectoral amount means a distance. We calculate the distance as using coordinates system and angles.

2.4.2.2.5. Cost

For every motion, robots lose power, this means robots pay for the motion as like fire and they have power limits so they must spend the motion power in optimum way also the costs depend on type of robot, type of the way to the target(mountains, rivers has may spend more etc), the total distance etc.

2.4.2.3. Endurance

2.4.2.3.1. Health

We have two ideas for power level:

- Team Power :
  - The total Value is the same for every team in start condition, here value means the total capability of the robots(power,shield and bullet type) and it is working similar as robot power for each team.
- Individual Robot power:
  - For every robot the power level may be vary so for a defence robot the power limit may be low for an attack robot it will be more.

2.4.2.4. Sensing

2.4.2.4.1. Vision

With vision sensing robots can differ the enemy, collegues or objects. Robots has a limit of vision sense for instance just a limited angle of visibility they will have, or in bad weather condition so they can not see properly.

2.4.2.4.2. Motion

Robots shall sense motion for a range we learned that some conditions the vision sensing technic can not work so the there are some other technics one of them is motion sensing.

Motion sensing has the range, in a limited range robot has a radar for sense the things that they are in motion, then if the vision sensing technic works for differ the object.

2.4.2.5. Messaging

2.4.2.5.1. Location

Robots give each other the location infos about enemies and their own maybe infos about the topology too.

2.4.2.5.2. Sensor info

Robots have lots of sensors to get info about fire, weather condition,ground condition.

2.4.2.5.3. Range of Messaging

Communication property is using in a wide area but it also should have a range, for some condition SOS call shall not have a range.

2.5. Non-Functional Requirements

Heterogeneity: Types of Robots determined by capabilities. In general there are 6 types of robots where each type is dominant of one capability. The total power capacity of the teams are the same, but it will be distort to all robots not in the same level.

Objects: Objects which fixed at the first of the game. Rivers, mounts and other geographical objects can be defined as fixed object. These objects can be attached to the defined topologies.

2.6. User Interface Requirements

Our environment uses JAVA so we use our own User Interface this includes a main menu and some sub forms to adjust and in general view of the system has a map view that has the resolution and can be adjusted by the user.

2.7. Dependencies and Constraints

- Our platform contains an area that we defined the map where our hardcoded robots can battle against each other.
- Each Robot has similar and different equipments also they are changeable by the user before starting game. User can limit the gun, health and decide to play in these conditions. The equipments are mainly, radar, gun, moving ability, energy tank etc.
- Robots have energy limits and this will be defined by user before game and when the energy becomes zero the robot dies.
- Robots have ability to move. Moving are mainly 4 functions moving back, forward, turning left and right.
- Robots have ability to fire the enemy if it is in the range of the gun.
- The enemy can be scanned by the robot within the radar range.
- User only affects the game before starting game as like adjusting the conditions, robot selections, map options.
- Player define the robot number within the map limits
- Player define the total health, total gun power and max for each robot.
- Player will change the graphics.
- Player can restart, load, save, save as the game.

2.8. Hardware and Software to be used

2.8.1. Hardware

- PC or MAC
- Sound system for sound effects
- At least 64 MB of free memory

2.8.2. Software

- Java Runtime Environment (at least JRE 1.5.0)
- OS (Platform Independent so LINUX, SOLARIS, WINDOWS etc.)
- At least 1024x768 Screen Resolution

3. Design Part

3.1. Introduction

3.1.1. Purpose

This Document contains design information about our war simulator RoboTotalWar project. The main idea for preparing design report is describing how will the software system be implemented. Definition of the layers, GUI layer and Game Engine layer also selected design model will be introduced. Definitions of the Components, scenarios to be used for engine, architectural decisions and selected design model will be introduced also the key point of this part is describing how will the system implemented.
The main reason behind the design and development of RoboTotalWar is mainly due to the great advancements in the fields of OOP and AI technologies. The project is composed of two main components that are necessary for deployment. First, UI that should be user friendly and the second is RoboTotalWar Game Engine that uses some small parts of AI techniques.

3.1.2. Related Technologies

Robocode is also an environment of robot development in Java with using OOP techniques.[1]

3.1.3. Scope

The main aim of this project is to create the application of OOP development to simulate an environment of a robot war game with small steps of artificial intelligence in order to evolve a user controller for robot in a simulated environment. The purpose is to develop an environment that users add their own robots within the rules that are finalized when the development time.

3.1.4. Document Overview

In this document, software development model, design model, architectural design of the main system, UI descriptions and strategic desicions for robots, detailed information about objects, classes and their relationships are available.

3.2. Project Development

3.2.1. Software Development Model

The most appropriate development technique model is Component-based Software Engineering, so requirements easily tested and will be handling for future also future developments are also easier. Also using OOP language JAVA enables us to develop Components easily, well undersandable and dynamically.

3.2.2. Design Model

In this software project, we use OO strategy because RoboTotalWar is in JAVA platform that is an OOP language. The design design model of our project has an OO structure,.
Our Simulator project is similar with Robocode that is one of the most popular strategy game. We have a good experience on JAVA programming. The requirements to be implemented are concerned as object and they are assumed as problems which will be transformed into solution objects. The requirements such as walls, maps, robots, mountains guns etc. are thought to be objects, so objects oriented design is suitable for the project.

3.2.3. Software Development Tools

We develop our project in JAVA language as I mentioned before, this is a language requirement for our project because we develop our environment in a OO design structure.
We use NETBEANS IDE, IntelliJ IDEA for development side also we develop the UML Diagrams with Poseidon for UML tool and FreeMind brain storming tool for design the all project.

3.3. Architectural Design

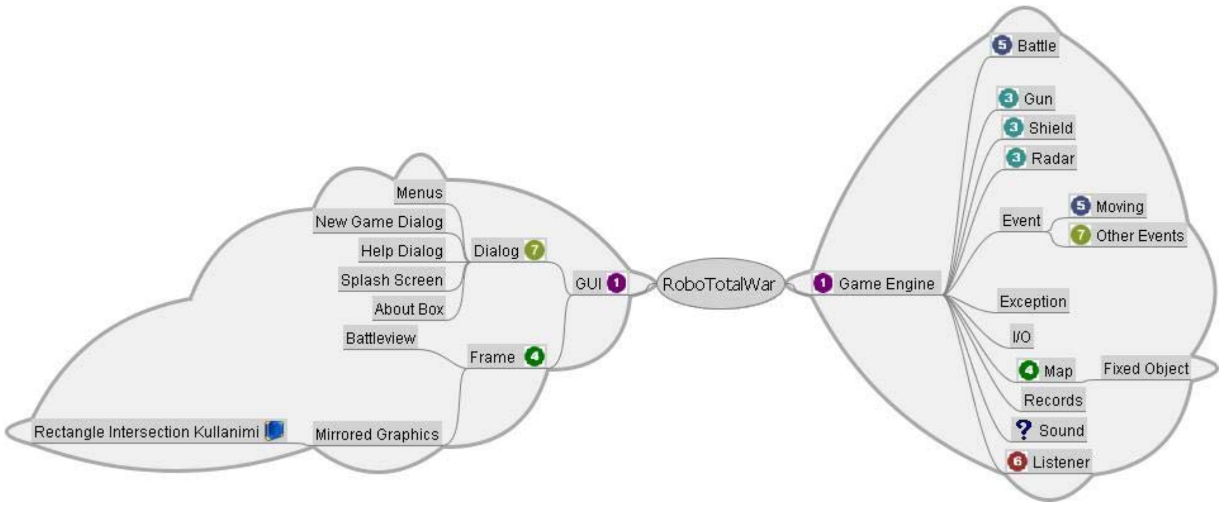


Figure 2.3 1: Brain Stroming Map of Project RoboTotalWar

3.3.1. Brain Storming

We use a brainstorming tool for designing the best architecture for our software. This tool helps us to design the steps of the design level. We use OOP and, have components the levels of the architecture includes these levels of the project.

3.3.2. Object Oriented Decomposition

After selecting the design model of the project, the next step is to decompose it into independent smaller objects and identify the classes, attributes and methods of those objects. Our project is decomposed firstly 2 Sub-Systems, these are also our project's layers and names UI and Game Engine, that are independtly reusable within other projects. We use mainly 13 sub packages and 1 main package, these helps us to seperate the each component about their features.

3.3.3. General Constraints

The main constraint is the rules of the environment that must be obeyed by new robots that are added by the users later.
Programmers should obey the rules of our platofrm that may restrict users but provide standardizing robots.
RoboTotalWar is opensource program. Programmers should be careful to maintain the memory and other capacity limitations in order to get high performance.
For detailed information, requirement specification report can be checked.

3.3.4. Goals and Objectives

The goal of our project is to develop an environment that have maintainability features as like adding new robots within the specific constraints that are also the rules of the environment. By applying the software development phases, we learn how to develop a designed project that means an architectural design is our best profit.
And the main objective aim of final project is to build an effective reusable and mainainable environment.

3.4. Class Diagrams

In this part user will find the descriptions of every important classes in this project, Which class do what, work integrated with which classes etc. For detailed class descriptions and method explanations please check RoboTotalWar Design Report[2].

3.4.1. RoboTotalWar Package and Included Important Classes

This package is main package of our project that also includes the other packages inside. Main controls of the program are also controlled in this package.

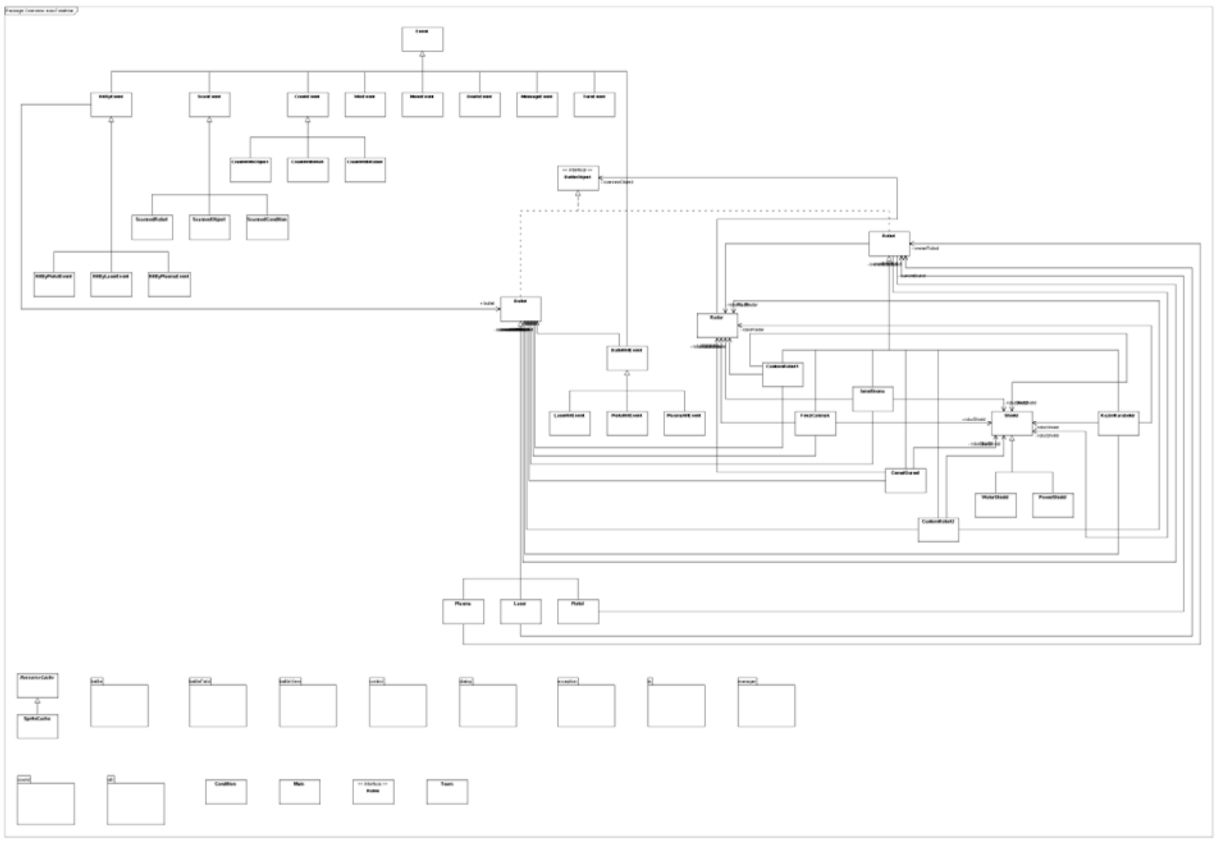
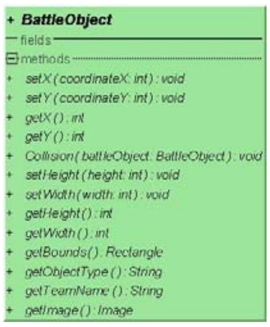


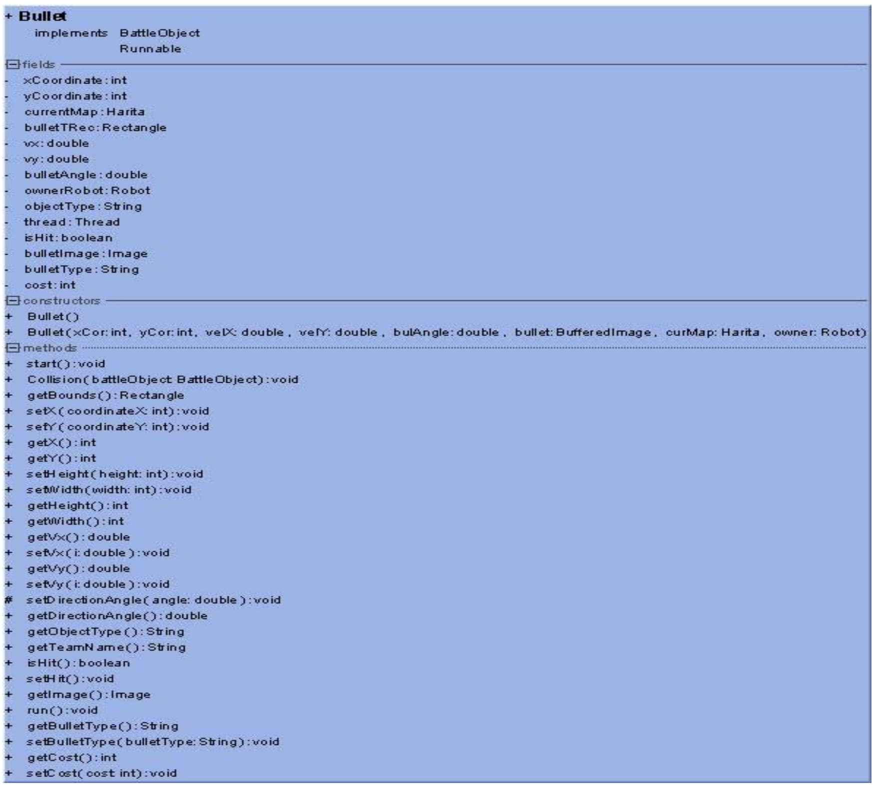
Figure 2.4 1: RoboTotalWar Package overview

3.4.1.1. BattleObject Interface


+ BattleObject
fields
methods
+ setX(coordinateX: int) : void
+ setY(coordinateY: int) : void
+ getX() : int
+ getY() : int
+ Collision(battleObject: BattleObject) : void
+ setHeight(height: int) : void
+ setWidth(width: int) : void
+ getHeight() : int
+ getWidth() : int
+ getBounds() : Rectangle
+ getObjectType() : String
+ getTeamName() : String
+ getImage() : Image
Figure 2.4 2: BattleObject Interface

This class is an interface for every object that included in the battle, for instance Robots, Mountains, Rivers, Bullets etc. Also this class mainly have the methods of the min. Object requirements that helps us in the future design of other objects.

3.4.1.2. Battle Class

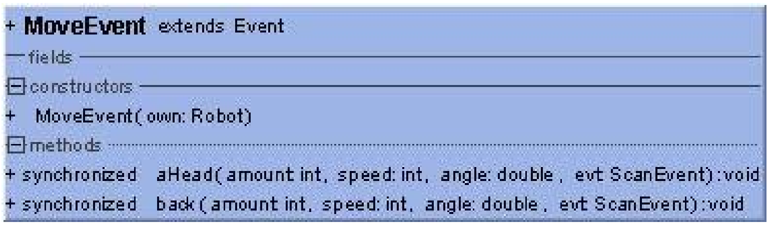


```
+ Bullet
    implements  BattleObject
                Runnable
⊟ fields
-  xCoordinate : int
-  yCoordinate : int
-  currentMap : Harita
-  bulletTRec : Rectangle
-  vx : double
-  vy : double
-  bulletAngle : double
-  ownerRobot : Robot
-  objectType : String
-  thread : Thread
-  isHit : boolean
-  bulletImage : Image
-  bulletType : String
-  cost : int
⊟ constructors
+  Bullet()
+  Bullet(xCor: int, yCor: int, velX: double , velY: double , bulAngle: double , bullet: BufferedImage , curMap: Harita , owner: Robot)
⊟ methods
+  start() : void
+  Collision( battleObject: BattleObject) : void
+  getBounds() : Rectangle
+  setX( coordinateX: int) : void
+  setY( coordinateY: int) : void
+  getX() : int
+  getY() : int
+  setHeight( height: int) : void
+  setWidth( width: int) : void
+  getHeight() : int
+  getWidth() : int
+  getVx() : double
+  setVx( i: double ) : void
+  getVy() : double
+  setVy( i: double ) : void
#  setDirectionAngle( angle: double ) : void
+  getDirectionAngle() : double
+  getObjectType() : String
+  getTeamName() : String
+  isHit() : boolean
+  setHit() : void
+  getImage() : Image
+  run() : void
+  getBulletType() : String
+  setBulletType( bulletType: String) : void
+  getCost() : int
+  setCost( cost: int) : void
```

Figure 2.4 3: Bullet Class

This class is our main bullet class that we develop three other types of bullet with extending it within the OO rules and constraints. This class works as an object for our bullet types, the types of our bullets are pistol, laser and plasma those have some constraints as hit power and speed.

3.4.1.3. MoveEvent Class

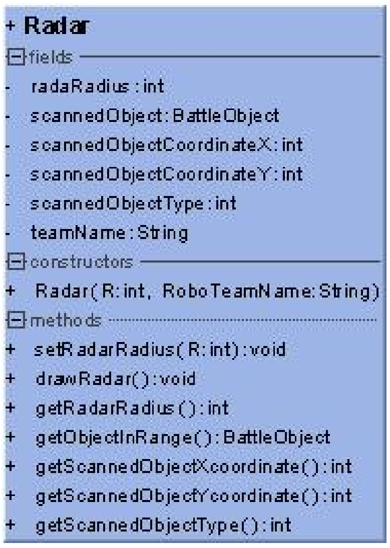


```
+ MoveEvent extends Event
fields
constructors
+ MoveEvent( own: Robot)
methods
+ synchronized aHead( amount int, speed: int, angle: double , evt ScanEvent) :void
+ synchronized back ( amount int, speed: int, angle: double , evt ScanEvent) :void
```

Figure 2.4 4: MoveEvent Class

This class is enable to move robots and bullets within the angle, speed and amount constraints, that also contains ScanEvent for robots to control when they are moving.

3.4.1.4. Radar Class

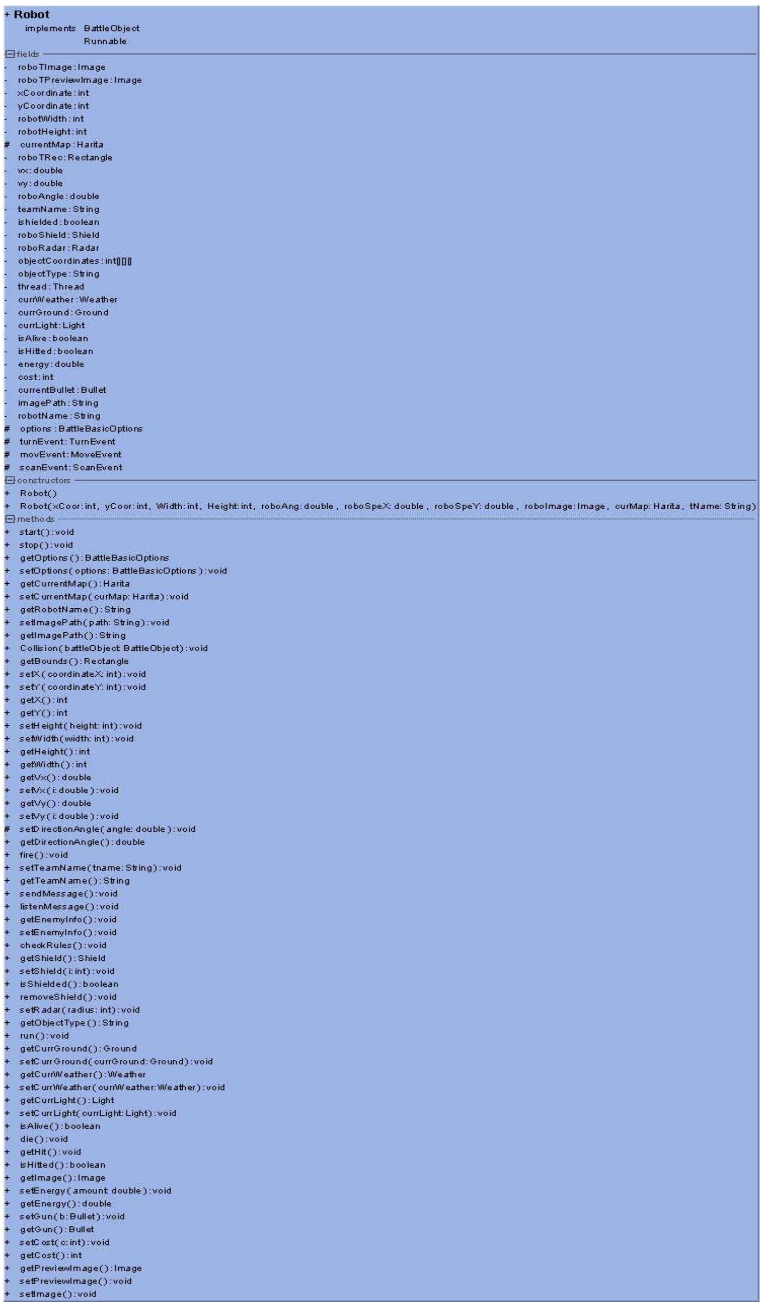


```
+ Radar
fields
- radaRadius : int
- scannedObject : BattleObject
- scannedObjectCoordinateX : int
- scannedObjectCoordinateY : int
- scannedObjectType : int
- teamName : String
constructors
+ Radar( R: int, RoboTeamName: String)
methods
+ setRadarRadius( R: int) : void
+ drawRadar() : void
+ getRadarRadius () : int
+ getObjectInRange() : BattleObject
+ getScannedObjectXcoordinate() : int
+ getScannedObjectYcoordinate() : int
+ getScannedObjectType() : int
```

Figure 2.4 5: Radar Class

This class enables robots to scan the environment within the scan lines, these scan lines are also increasing within the completed battle time.

3.4.1.5. Robot Class

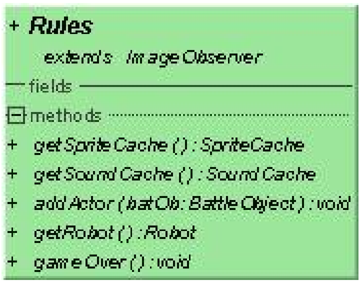


```
+ Robot
    implements  BattleObject
                Runnable
⊟ fields
-  roboTImage : Image
-  roboTPreviewImage : Image
-  xCoordinate : int
-  yCoordinate : int
-  robotWidth : int
-  robotHeight : int
#  currentMap : Harita
-  roboTRec : Rectangle
-  vx : double
-  vy : double
-  roboAngle : double
-  teamName : String
-  ishielded : boolean
-  roboShield : Shield
-  roboRadar : Radar
-  objectCoordinates : int[][][]
-  objectType : String
-  thread : Thread
-  currWeather : Weather
-  currGround : Ground
-  currLight : Light
-  isAlive : boolean
-  isHitted : boolean
-  energy : double
-  cost : int
-  currentBullet : Bullet
-  imagePath : String
-  robotName : String
#  options : BattleBasicOptions
#  turnEvent : TurnEvent
#  movEvent : MoveEvent
#  scanEvent : ScanEvent
⊟ constructors
+  Robot()
+  Robot(xCoor: int, yCoor: int, Width: int, Height: int, roboAng: double , roboSpeX: double , roboSpeY: double , roboImage: Image, curMap: Harita, tName: String)
⊟ methods
+  start() : void
+  stop() : void
+  getOptions () : BattleBasicOptions
+  setOptions( options: BattleBasicOptions ) : void
+  getCurrentMap() : Harita
+  setCurrentMap( curMap: Harita ) : void
+  getRobotName() : String
+  setImagePath( path: String ) : void
+  getImagePath() : String
+  Collision( battleObject: BattleObject) : void
+  getBounds() : Rectangle
+  setX( coordinateX: int) : void
+  setY( coordinateY: int) : void
+  getX() : int
+  getY() : int
+  setHeight( height: int) : void
+  setWidth( width: int) : void
+  getHeight() : int
+  getWidth() : int
+  getVx() : double
+  setVx( i: double ) : void
+  getVy() : double
+  setVy( i: double ) : void
#  setDirectionAngle( angle: double ) : void
+  getDirectionAngle() : double
+  fire() : void
+  setTeamName( tname: String ) : void
+  getTeamName() : String
+  sendMessage() : void
+  listenMessage() : void
+  getEnemyInfo() : void
+  setEnemyInfo() : void
+  checkRules() : void
+  getShield() : Shield
+  setShield( i: int) : void
+  isShielded() : boolean
+  removeShield() : void
+  setRadar( radius: int) : void
+  getObjectType () : String
+  run() : void
+  getCurrGround() : Ground
+  setCurrGround( currGround: Ground ) : void
+  getCurrWeather() : Weather
+  setCurrWeather( currWeather: Weather ) : void
+  getCurrLight() : Light
+  setCurrLight( currLight: Light ) : void
+  isAlive() : boolean
+  die() : void
+  getHit() : void
+  isHitted() : boolean
+  getImage() : Image
+  setEnergy( amount: double ) : void
+  getEnergy() : double
+  setGun( b: Bullet ) : void
+  getGun() : Bullet
+  setCost( c: int) : void
+  getCost() : int
+  getPreviewImage() : Image
+  setPreviewImage() : void
+  setImage() : void
```

Figure 2.4 6: Robot Class

This class is our Robot.Class that has some main functionalities that the later Robot developments that they must use and obey the constraints. Robot class is integrated with Bullet, Harita, Event, BattleObjects, Team classes.

3.4.1.6. Rules Interface

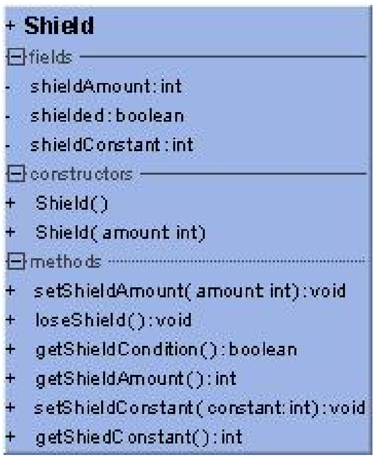


```
+ Rules
    extends ImageObserver
fields
methods
+ getSpriteCache () : SpriteCache
+ getSoundCache () : SoundCache
+ addActor (batOb: BattleObject) : void
+ getRobot () : Robot
+ gameOver () : void
```

Figure 2.4 7: Rules Interface

This class enables BattleView class to describe how the Observation of images usable in program, this is an interface and only used by BattleView.Class.

3.4.1.7. Shield Class

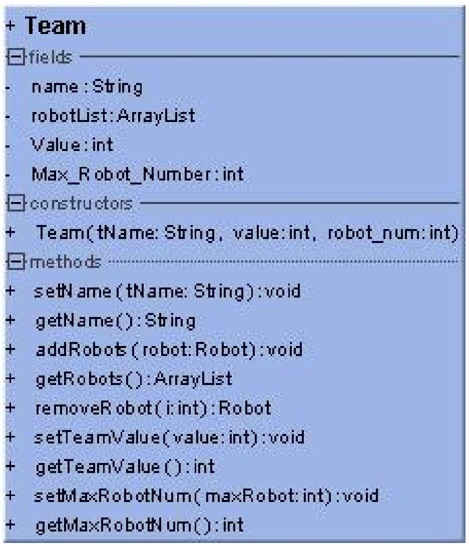


```
+ Shield
fields
- shieldAmount:int
- shielded:boolean
- shieldConstant:int
constructors
+ Shield()
+ Shield( amount int)
methods
+ setShieldAmount( amount int):void
+ loseShield():void
+ getShieldCondition():boolean
+ getShieldAmount():int
+ setShieldConstant( constant:int):void
+ getShiedConstant():int
```

Figure 2.4 8: Shield Class

This Class used by Robots, they have the cost of shield and shield works as an energy but the difference is it will be work for water or other conditions too. Types of shields are extending this class.

3.4.1.8. Team Class



```
+ Team
fields
-   name : String
-   robotList : ArrayList
-   Value : int
-   Max_Robot_Number : int
constructors
+   Team(tName: String, value: int, robot_num: int)
methods
+   setName(tName: String) : void
+   getName() : String
+   addRobots(robot: Robot) : void
+   getRobots() : ArrayList
+   removeRobot(i: int) : Robot
+   setTeamValue(value: int) : void
+   getTeamValue() : int
+   setMaxRobotNum(maxRobot: int) : void
+   getMaxRobotNum() : int
```

Figure 2.4 9: Team Class

This class is used by BattleProperties that includes teams when a new battle created also every team has an arraylist of robots that included for each team.

3.4.2. Battle Package and Included Important Classes

This package mainly includes the Battle.Class that works for every battle, every action.

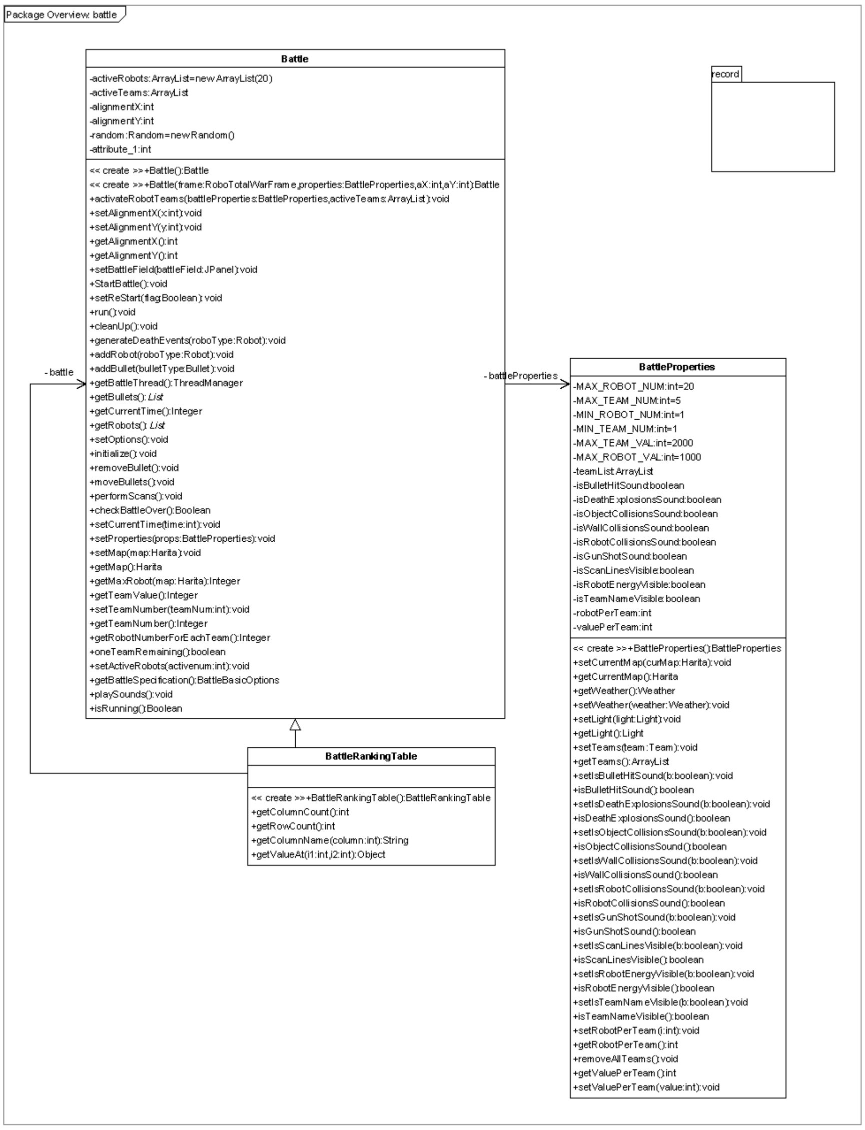
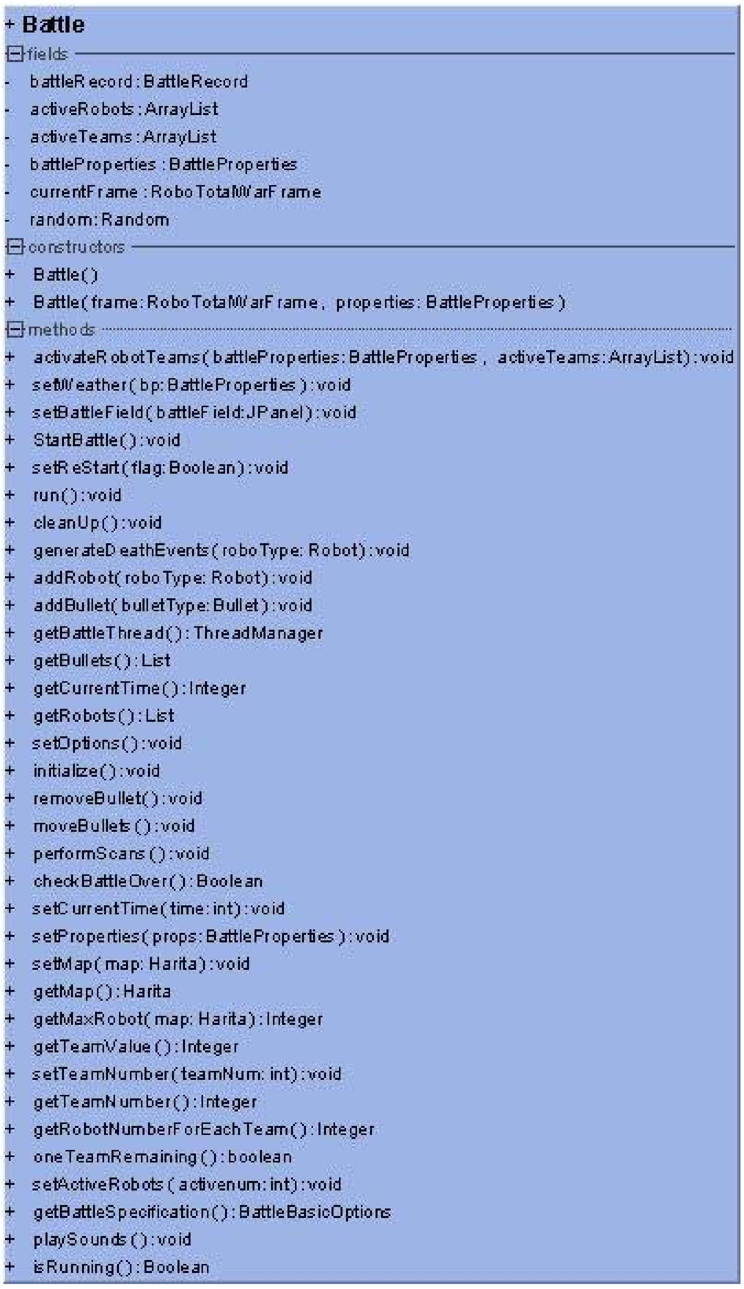

Package Overview: battle
Battle
-activeRobots:ArrayList=new ArrayList(20)
-activeTeams:ArrayList
-alignmentX:int
-alignmentY:int
-random:Random=newRandom()
-attribute_1:int
<< create >>+Battle():Battle
<< create >>+Battle(frame:RoboTotalWarFrame,properties:BattleProperties,aX:int,aY:int):Battle
+activateRobotTeams(battleProperties:BattleProperties,activeTeams:ArrayList):void
+setAlignmentX(x:int):void
+setAlignmentY(y:int):void
+getAlignmentX():int
+getAlignmentY():int
+setBattleField(battleField:JPanel):void
+StartBattle():void
+setReStart(flag:Boolean):void
+run():void
+cleanUp():void
+generateDeathEvents(roboType:Robot):void
+addRobot(roboType:Robot):void
+addBullet(bulletType:Bullet):void
+getBattleThread():ThreadManager
+getBullets(): *List*
+getCurrentTime():Integer
+getRobots(): *List*
+setOptions():void
+initialize():void
+removeBullet():void
+moveBullets():void
+performScans():void
+checkBattleOver():Boolean
+setCurrentTime(time:int):void
+setProperties(props:BattleProperties):void
+setMap(map:Harita):void
+getMap():Harita
+getMaxRobot(map:Harita):Integer
+getTeamValue():Integer
+setTeamNumber(teamNum:int):void
+getTeamNumber():Integer
+getRobotNumberForEachTeam():Integer
+oneTeamRemaining():boolean
+setActiveRobots(activenum:int):void
+getBattleSpecification():BattleBasicOptions
+playSounds():void
+isRunning():Boolean
record
- battle
- battleProperties
BattleProperties
-MAX_ROBOT_NUM:int=20
-MAX_TEAM_NUM:int=5
-MIN_ROBOT_NUM:int=1
-MIN_TEAM_NUM:int=1
-MAX_TEAM_VAL:int=2000
-MAX_ROBOT_VAL:int=1000
-teamList:ArrayList
-isBulletHitSound:boolean
-isDeathExplosionsSound:boolean
-isObjectCollisionsSound:boolean
-isWallCollisionsSound:boolean
-isRobotCollisionsSound:boolean
-isGunShotSound:boolean
-isScanLinesVisible:boolean
-isRobotEnergyVisible:boolean
-isTeamNameVisible:boolean
-robotPerTeam:int
-valuePerTeam:int
<< create >>+BattleProperties():BattleProperties
+setCurrentMap(curMap:Harita):void
+getCurrentMap():Harita
+getWeather():Weather
+setWeather(weather:Weather):void
+setLight(light:Light):void
+getLight():Light
+setTeams(team:Team):void
+getTeams():ArrayList
+setIsBulletHitSound(b:boolean):void
+isBulletHitSound():boolean
+setIsDeathExplosionsSound(b:boolean):void
+isDeathExplosionsSound():boolean
+setIsObjectCollisionsSound(b:boolean):void
+isObjectCollisionsSound():boolean
+setIsWallCollisionsSound(b:boolean):void
+isWallCollisionsSound():boolean
+setIsRobotCollisionsSound(b:boolean):void
+isRobotCollisionsSound():boolean
+setIsGunShotSound(b:boolean):void
+isGunShotSound():boolean
+setIsScanLinesVisible(b:boolean):void
+isScanLinesVisible():boolean
+setIsRobotEnergyVisible(b:boolean):void
+isRobotEnergyVisible():boolean
+setIsTeamNameVisible(b:boolean):void
+isTeamNameVisible():boolean
+setRobotPerTeam(i:int):void
+getRobotPerTeam():int
+removeAllTeams():void
+getValuePerTeam():int
+setValuePerTeam(value:int):void
BattleRankingTable
<< create >>+BattleRankingTable():BattleRankingTable
+getColumnCount():int
+getRowCount():int
+getColumnName(column:int):String
+getValueAt(i1:int,i2:int):Object

Figure 2.4.2 1: Battle Package overview

3.4.2.1. Battle Class

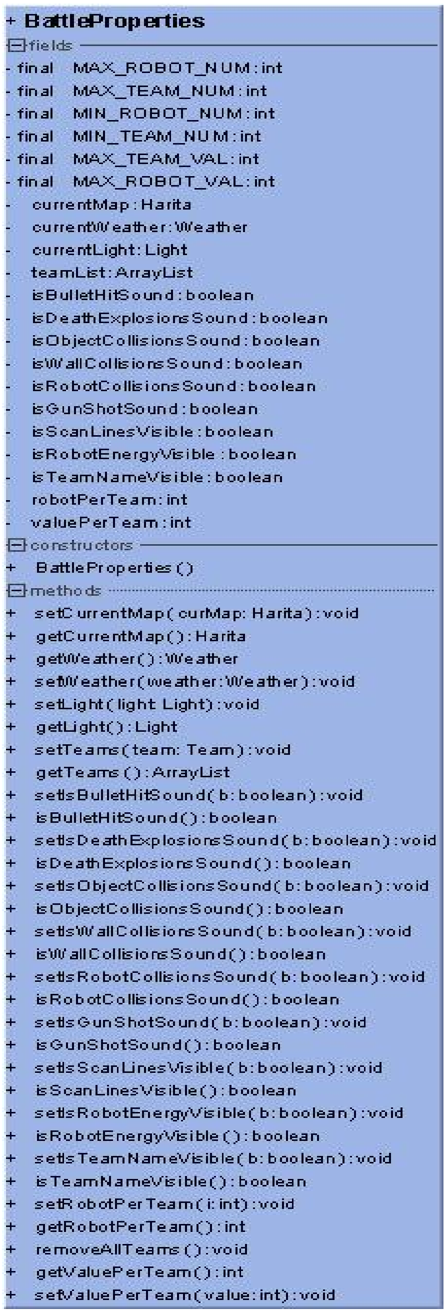


```
+ Battle
⊟ fields
-   battleRecord : BattleRecord
-   activeRobots : ArrayList
-   activeTeams : ArrayList
-   battleProperties : BattleProperties
-   currentFrame : RoboTotalWarFrame
-   random : Random
⊟ constructors
+   Battle()
+   Battle( frame : RoboTotalWarFrame,  properties : BattleProperties )
⊟ methods
+   activateRobotTeams( battleProperties : BattleProperties ,  activeTeams : ArrayList) : void
+   setWeather( bp : BattleProperties ) : void
+   setBattleField( battleField:JPanel) : void
+   StartBattle() : void
+   setReStart( flag : Boolean ) : void
+   run() : void
+   cleanUp() : void
+   generateDeathEvents( roboType : Robot) : void
+   addRobot( roboType : Robot) : void
+   addBullet( bulletType : Bullet ) : void
+   getBattleThread() : ThreadManager
+   getBullets() : List
+   getCurrentTime() : Integer
+   getRobots() : List
+   setOptions() : void
+   initialize() : void
+   removeBullet() : void
+   moveBullets () : void
+   performScans () : void
+   checkBattleOver() : Boolean
+   setCurrentTime( time : int) : void
+   setProperties( props : BattleProperties ) : void
+   setMap( map : Harita) : void
+   getMap() : Harita
+   getMaxRobot( map : Harita) : Integer
+   getTeamValue () : Integer
+   setTeamNumber( teamNum : int) : void
+   getTeamNumber() : Integer
+   getRobotNumberForEachTeam() : Integer
+   oneTeamRemaining () : boolean
+   setActiveRobots ( activenum : int) : void
+   getBattleSpecification() : BattleBasicOptions
+   playSounds () : void
+   isRunning() : Boolean
```

Figure 2.4.2 2: Battle Class

This class is the main class for every each battle, a battle includes options, rules, properties, teams, harita, objects(robots, bullets, fixedObjects etc.). Also battle will be restart, aborted, stop, pause and end.

3.4.2.2. BattleProperties Class



```
+ BattleProperties
fields
- final   MAX_ROBOT_NUM : int
- final   MAX_TEAM_NUM : int
- final   MIN_ROBOT_NUM : int
- final   MIN_TEAM_NUM : int
- final   MAX_TEAM_VAL : int
- final   MAX_ROBOT_VAL : int
-   currentMap : Harita
-   currentWeather : Weather
-   currentLight : Light
-   teamList : ArrayList
-   isBulletHitSound : boolean
-   isDeathExplosionsSound : boolean
-   isObjectCollisionsSound : boolean
-   isWallCollisionsSound : boolean
-   isRobotCollisionsSound : boolean
-   isGunShotSound : boolean
-   isScanLinesVisible : boolean
-   isRobotEnergyVisible : boolean
-   isTeamNameVisible : boolean
-   robotPerTeam : int
-   valuePerTeam : int
constructors
+   BattleProperties ()
methods
+   setCurrentMap( curMap: Harita ) : void
+   getCurrentMap() : Harita
+   getWeather() : Weather
+   setWeather( weather: Weather ) : void
+   setLight( light: Light ) : void
+   getLight() : Light
+   setTeams( team: Team ) : void
+   getTeams () : ArrayList
+   setIsBulletHitSound( b: boolean ) : void
+   isBulletHitSound() : boolean
+   setIsDeathExplosionsSound( b: boolean ) : void
+   isDeathExplosionsSound() : boolean
+   setIsObjectCollisionsSound( b: boolean ) : void
+   isObjectCollisionsSound() : boolean
+   setIsWallCollisionsSound( b: boolean ) : void
+   isWallCollisionsSound() : boolean
+   setIsRobotCollisionsSound( b: boolean ) : void
+   isRobotCollisionsSound() : boolean
+   setIsGunShotSound( b: boolean ) : void
+   isGunShotSound() : boolean
+   setIsScanLinesVisible( b: boolean ) : void
+   isScanLinesVisible() : boolean
+   setIsRobotEnergyVisible( b: boolean ) : void
+   isRobotEnergyVisible () : boolean
+   setIsTeamNameVisible( b: boolean ) : void
+   isTeamNameVisible() : boolean
+   setRobotPerTeam( i: int ) : void
+   getRobotPerTeam() : int
+   removeAllTeams () : void
+   getValuePerTeam() : int
+   setValuePerTeam( value: int ) : void
```

Figure 2.4.2 3: BattleProperties

BattleProperties class has the functions of properties of the new battle harita, robots has the usage of this class also Battle uses this class too for adapting the new battle, BattleView class uses the options of the battleProperties.

3.4.2.3. BattleRankingTable Class

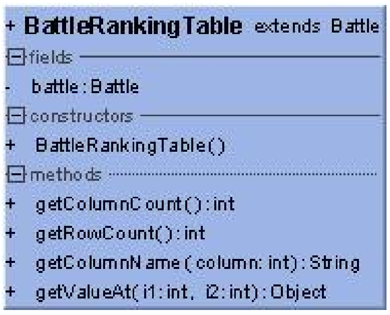

+ BattleRankingTable extends Battle
fields
- battle : Battle
constructors
+ BattleRankingTable()
methods
+ getColumnCount() : int
+ getRowCount() : int
+ getColumnName(column: int) : String
+ getValueAt(i1 : int, i2 : int) : Object

Figure 2.4.2 4: BattleRankingTable Class

This class can be used when the battle is ended, it gets the records from Battle Class and returns the results within this table, the battle end when there is only one team live.

3.4.3. Record Package and Included Important Classes

This Package contains the records of the bullet, robot and battle

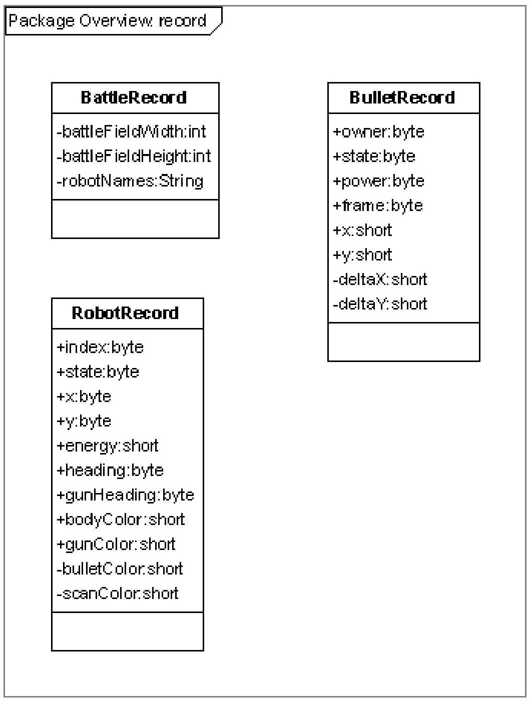


Figure 2.4.3 1 Record Package Overview

3.4.4. BattleField Package and Included Important Classes

This package contains 2 more packages and the maps that we use in our environment that are all contains special strategic points of our county. They all extend Harita class.

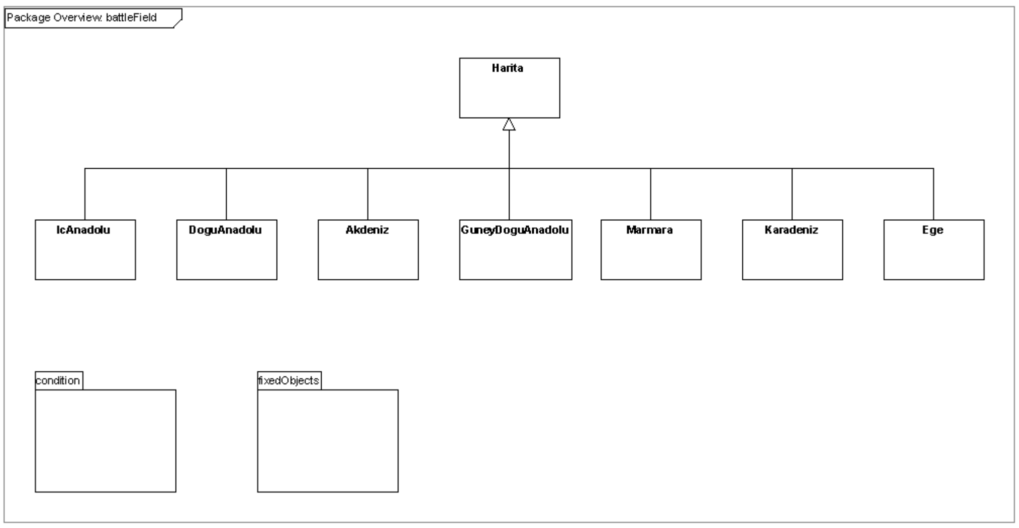
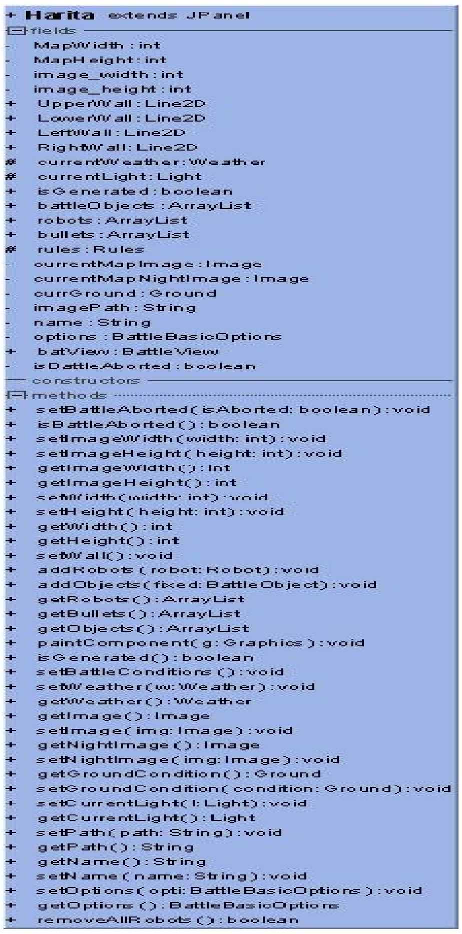


Figure 2.4.4 1: BattleField Package Overview

3.4.4.1. Harita Class



```
+ Harita  extends JPanel
fields
-   MapWidth : int
-   MapHeight : int
-   image_width : int
-   image_height : int
+   UpperWall : Line2D
+   LowerWall : Line2D
+   LeftWall : Line2D
+   RightWall : Line2D
#   currentWeather : Weather
#   currentLight : Light
+   isGenerated : boolean
+   battleObjects : ArrayList
+   robots : ArrayList
+   bullets : ArrayList
#   rules : Rules
-   currentMapImage : Image
-   currentMapNightImage : Image
-   currGround : Ground
-   imagePath : String
-   name : String
-   options : BattleBasicOptions
+   batView : BattleView
-   isBattleAborted : boolean
constructors
methods
+   setBattleAborted( isAborted: boolean ) : void
+   isBattleAborted() : boolean
+   setImageWidth( width: int ) : void
+   setImageHeight( height: int ) : void
+   getImageWidth() : int
+   getImageHeight() : int
+   setWidth( width: int ) : void
+   setHeight( height: int ) : void
+   getWidth() : int
+   getHeight() : int
+   setWall() : void
+   addRobots( robot: Robot ) : void
+   addObjects( fixed: BattleObject ) : void
+   getRobots() : ArrayList
+   getBullets() : ArrayList
+   getObjects() : ArrayList
+   paintComponent( g: Graphics ) : void
+   isGenerated() : boolean
+   setBattleConditions() : void
+   setWeather( w: Weather ) : void
+   getWeather() : Weather
+   getImage() : Image
+   setImage( img: Image ) : void
+   getNightImage() : Image
+   setNightImage( img: Image ) : void
+   getGroundCondition() : Ground
+   setGroundCondition( condition: Ground ) : void
+   setCurrentLight( l: Light ) : void
+   getCurrentLight() : Light
+   setPath( path: String ) : void
+   getPath() : String
+   getName() : String
+   setName( name: String ) : void
+   setOptions( opti: BattleBasicOptions ) : void
+   getOptions() : BattleBasicOptions
+   removeAllRobots() : boolean
```

Figure 2.4.4 2 Harita Class

This class extends Jpanel, it is a parent for all other maps, and has some general properties of a basic map.

,

3.4.5. Conditions Package and Included Important Classes

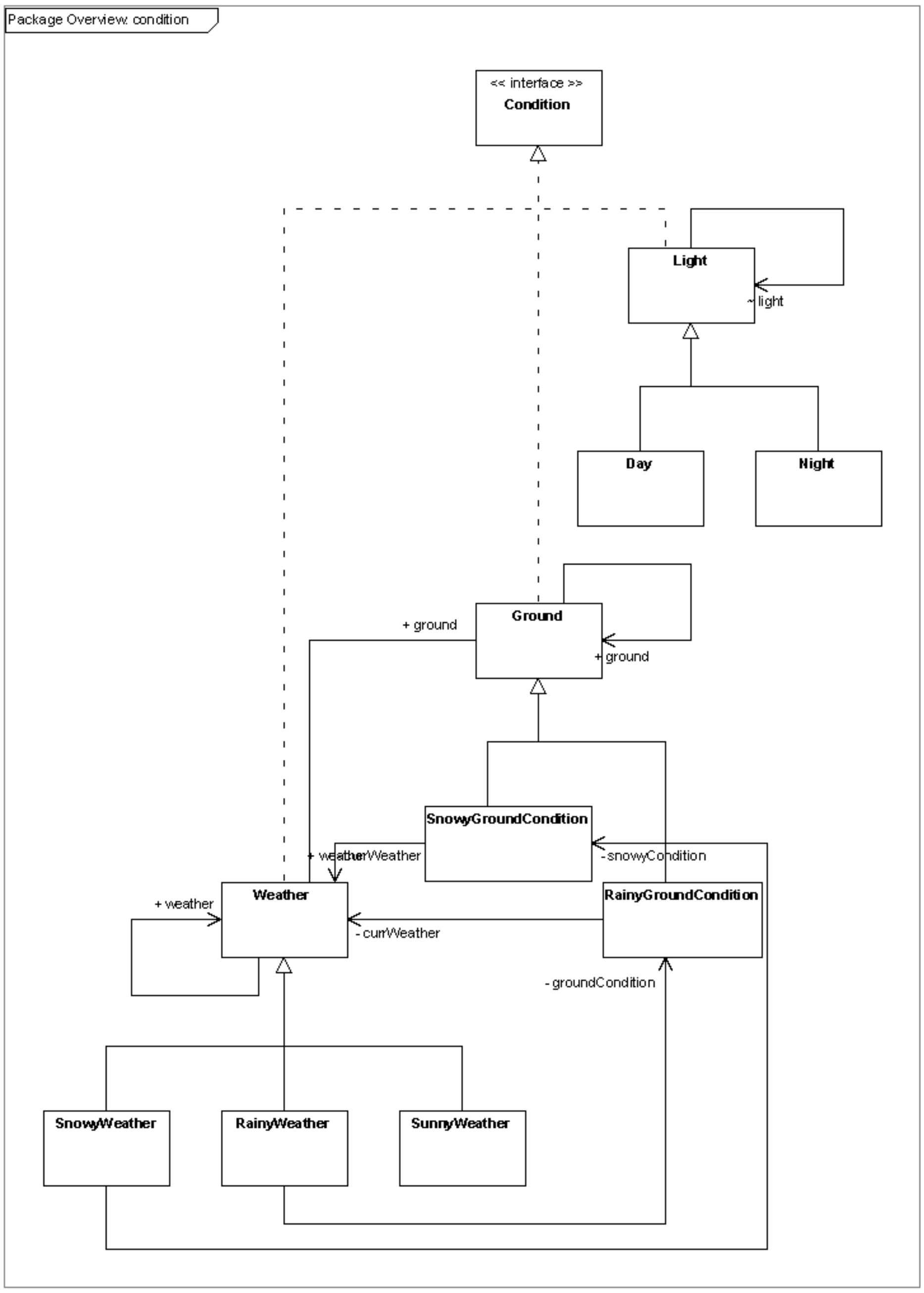


Figure 2.4.5 1 Conditions Package Overview

3.4.5.1. Condition Interface

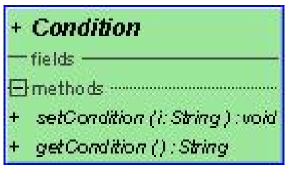


Figure 2.4.5 2

This is an interface for all conditions, such as Weather, Light, Ground.

3.4.5.2. Ground Class

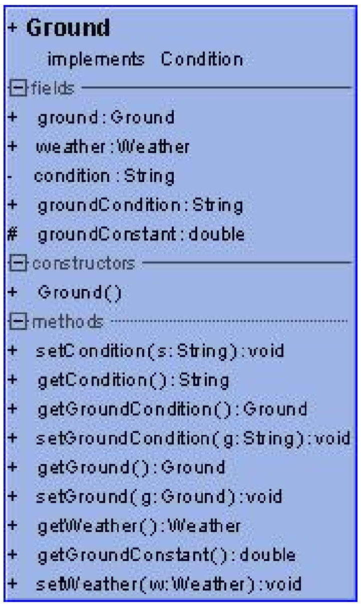

+ Ground
implements Condition
fields
+ ground : Ground
+ weather : Weather
- condition : String
+ groundCondition : String
# groundConstant : double
constructors
+ Ground()
methods
+ setCondition(s: String) : void
+ getCondition() : String
+ getGroundCondition() : Ground
+ setGroundCondition(g: String) : void
+ getGround() : Ground
+ setGround(g: Ground) : void
+ getWeather() : Weather
+ getGroundConstant() : double
+ setWeather(w: Weather) : void

Figure 2.4.5 3: Ground Class

Ground class is the unvisible but effective class for robots, it directly effects robots, interworking with weather condition.

3.4.5.3. Light Class

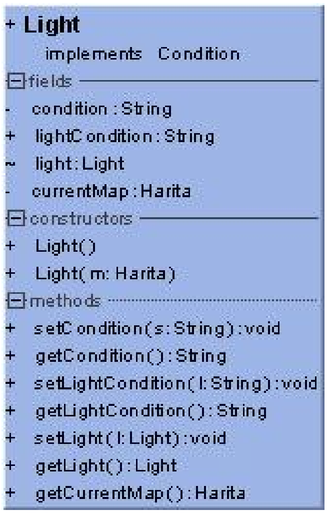

+ Light
implements Condition
fields
- condition : String
+ lightCondition : String
~ light : Light
- currentMap : Harita
constructors
+ Light()
+ Light(m: Harita)
methods
+ setCondition(s: String) : void
+ getCondition() : String
+ setLightCondition(l: String) : void
+ getLightCondition() : String
+ setLight(l: Light) : void
+ getLight() : Light
+ getCurrentMap() : Harita

Figure 2.4.5 4: Light Class

Light class contains 2 types, those extends this class named Night,Day this class in future will be used as an abnormal condition to effect robots, but now we only check the image and control the light constant.

3.4.5.4. Weather Class

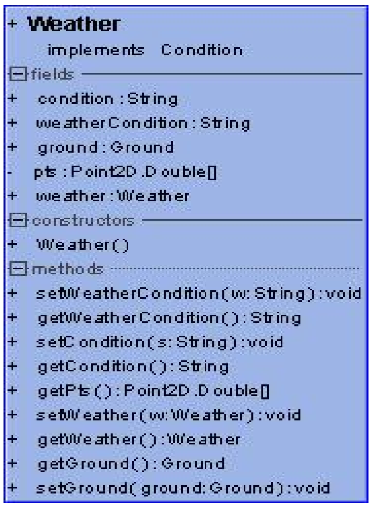

+ Weather
implements Condition
fields
+ condition : String
+ weatherCondition : String
+ ground : Ground
- pts : Point2D.Double[]
+ weather : Weather
constructors
+ Weather()
methods
+ setWeatherCondition(w: String) : void
+ getWeatherCondition() : String
+ setCondition(s: String) : void
+ getCondition() : String
+ getPts() : Point2D.Double[]
+ setWeather(w: Weather) : void
+ getWeather() : Weather
+ getGround() : Ground
+ setGround(ground: Ground) : void

Figure 2.4.5 5: Weather Class

This class is the parent class of RainyWeatherCondition, SnowyWeatherCondition and SunnyWeatherCondition that first two are runnable and effective objects and the third is normal condition class and has no more effect to the battle

3.4.6. FixedObjects Package and Included Important Classes

This package contains the objects of the maps, they are fixed and no more maintainable.

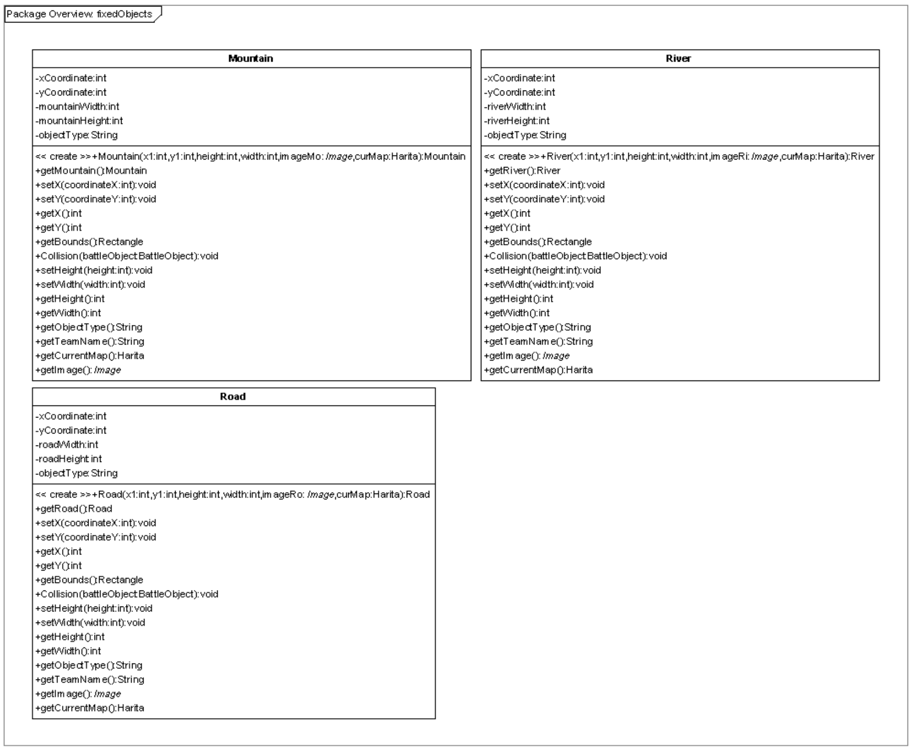
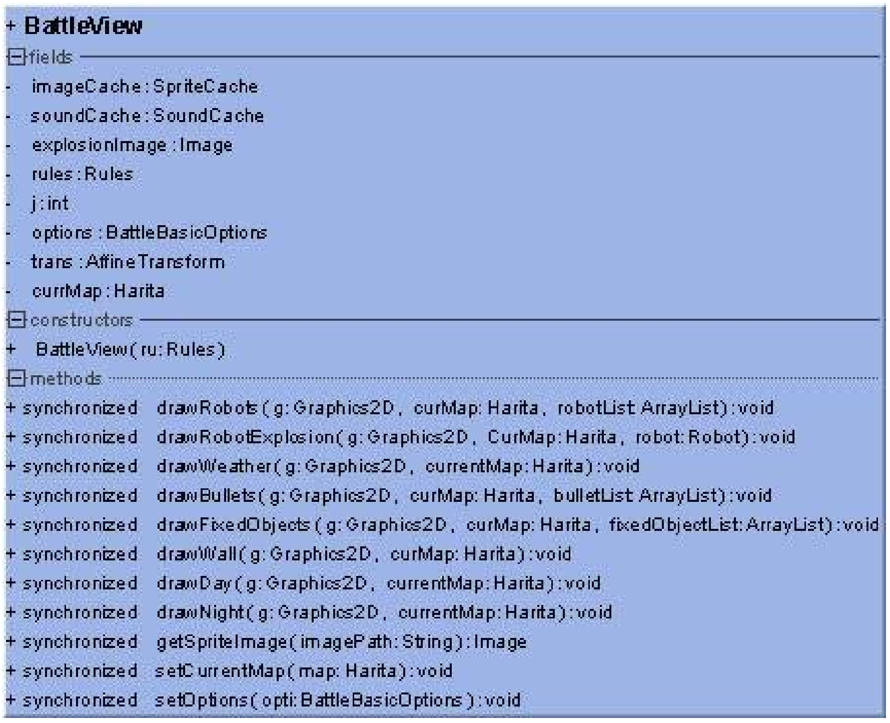


Figure 2.4.6 1: FixedObjects Package Overview

3.4.7. BattleView Package and Included Important Classes

This package only contains BattleView Class and this class works for every graphical object such as Harita, Robot, FixedObjects, Bullet etc.

3.4.7.1. BattleView Class



```
+ BattleView
⊟fields
-   imageCache : SpriteCache
-   soundCache : SoundCache
-   explosionImage : Image
-   rules : Rules
-   j : int
-   options : BattleBasicOptions
-   trans : AffineTransform
-   currMap : Harita
⊟constructors
+   BattleView( ru: Rules )
⊟methods
+ synchronized   drawRobots ( g: Graphics2D ,  curMap: Harita ,  robotList ArrayList) : void
+ synchronized   drawRobotExplosion( g: Graphics2D ,  CurMap: Harita ,  robot: Robot) : void
+ synchronized   drawWeather( g: Graphics2D ,  currentMap: Harita) : void
+ synchronized   drawBullets( g: Graphics2D ,  curMap: Harita ,  bulletList ArrayList) : void
+ synchronized   drawFixedObjects ( g: Graphics2D ,  curMap: Harita ,  fixedObjectList: ArrayList) : void
+ synchronized   drawWall( g: Graphics2D ,  curMap: Harita) : void
+ synchronized   drawDay ( g: Graphics2D ,  currentMap: Harita) : void
+ synchronized   drawNight( g: Graphics2D ,  currentMap: Harita) : void
+ synchronized   getSpriteImage( imagePath: String) : Image
+ synchronized   setCurrentMap( map: Harita) : void
+ synchronized   setOptions ( opti: BattleBasicOptions ) : void
```

Figure 2.4.7 1: BattleView Class

3.4.8. Control Package and Included Important Classes

This package controls Options, results of a battle and listeners and also contains the game engine that has the main functionalities of the game.

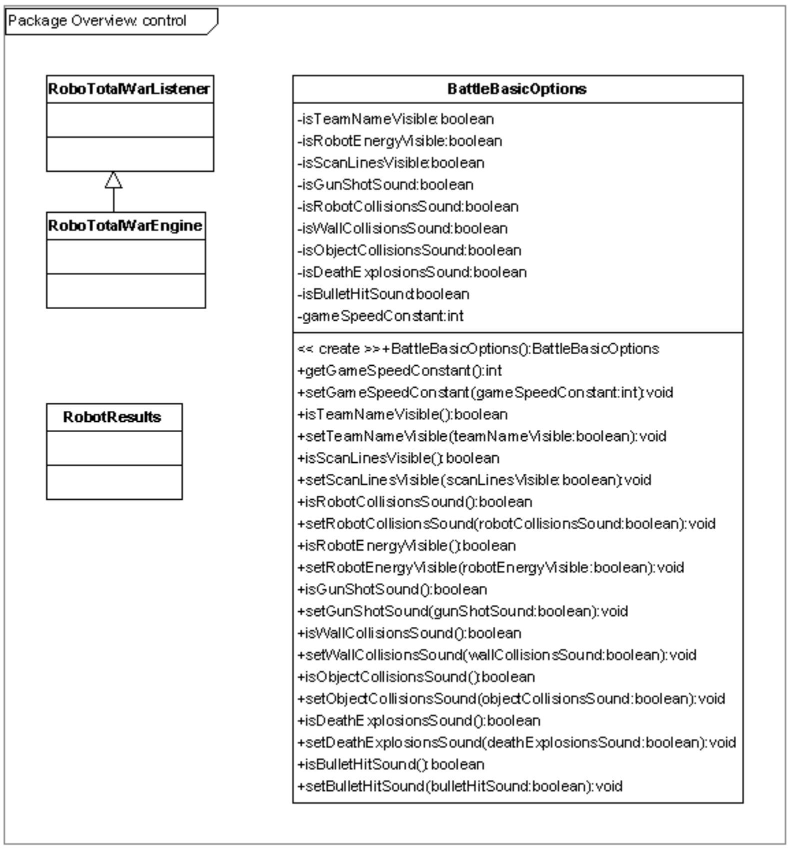
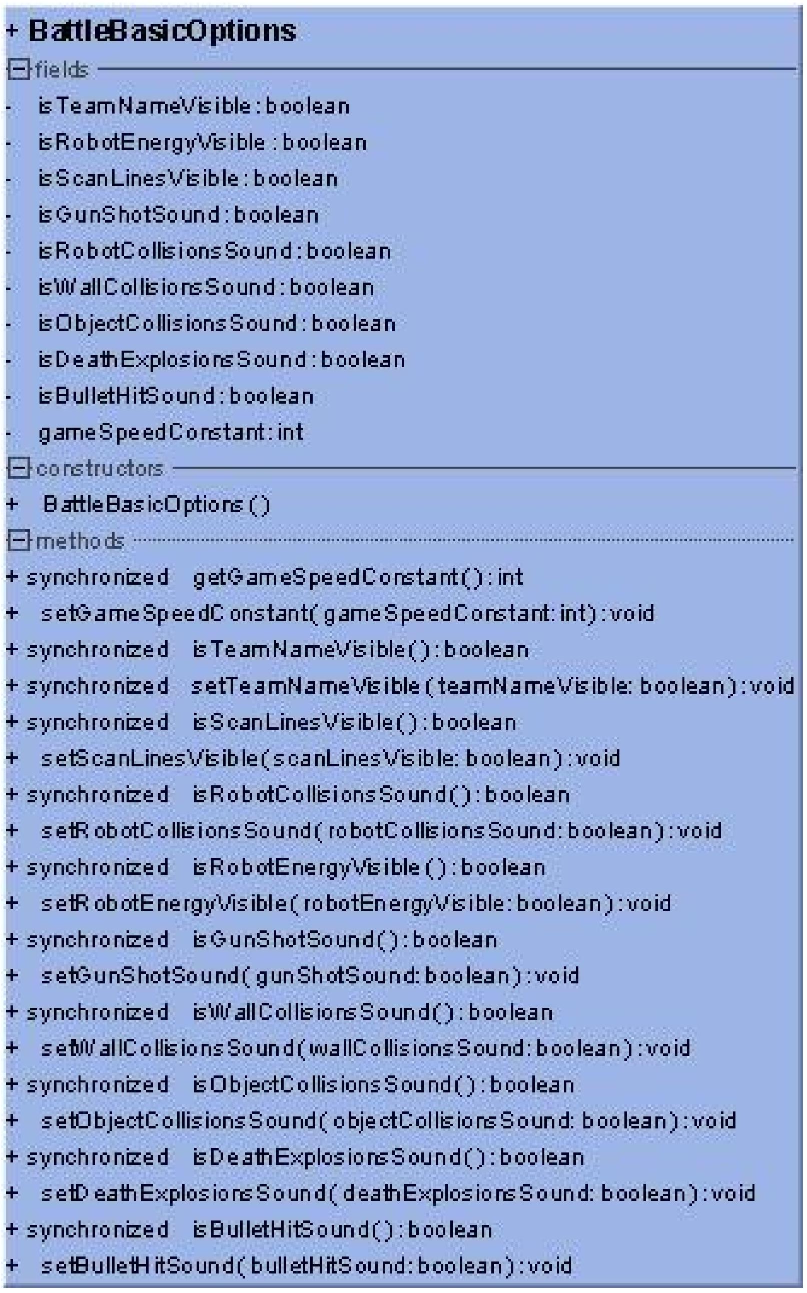

Package Overview control
RoboTotalWarListener
RoboTotalWarEngine
RobotResults
BattleBasicOptions
-isTeamNameVisible:boolean
-isRobotEnergyVisible:boolean
-isScanLinesVisible:boolean
-isGunShotSound:boolean
-isRobotCollisionsSound:boolean
-isWallCollisionsSound:boolean
-isObjectCollisionsSound:boolean
-isDeathExplosionsSound:boolean
-isBulletHitSound:boolean
-gameSpeedConstant:int
<< create >>+BattleBasicOptions():BattleBasicOptions
+getGameSpeedConstant():int
+setGameSpeedConstant(gameSpeedConstant:int):void
+isTeamNameVisible():boolean
+setTeamNameVisible(teamNameVisible:boolean):void
+isScanLinesVisible():boolean
+setScanLinesVisible(scanLinesVisible:boolean):void
+isRobotCollisionsSound():boolean
+setRobotCollisionsSound(robotCollisionsSound:boolean):void
+isRobotEnergyVisible():boolean
+setRobotEnergyVisible(robotEnergyVisible:boolean):void
+isGunShotSound():boolean
+setGunShotSound(gunShotSound:boolean):void
+isWallCollisionsSound():boolean
+setWallCollisionsSound(wallCollisionsSound:boolean):void
+isObjectCollisionsSound():boolean
+setObjectCollisionsSound(objectCollisionsSound:boolean):void
+isDeathExplosionsSound():boolean
+setDeathExplosionsSound(deathExplosionsSound:boolean):void
+isBulletHitSound():boolean
+setBulletHitSound(bulletHitSound:boolean):void

Figure 2.4.8 1: Control Package Overview

3.4.8.1. BattleBasicOptions Class



```
+ BattleBasicOptions
fields
-   isTeamNameVisible : boolean
-   isRobotEnergyVisible : boolean
-   isScanLinesVisible : boolean
-   isGunShotSound : boolean
-   isRobotCollisionsSound : boolean
-   isWallCollisionsSound : boolean
-   isObjectCollisionsSound : boolean
-   isDeathExplosionsSound : boolean
-   isBulletHitSound : boolean
-   gameSpeedConstant : int
constructors
+   BattleBasicOptions ()
methods
+ synchronized   getGameSpeedConstant() : int
+   setGameSpeedConstant( gameSpeedConstant: int) : void
+ synchronized   isTeamNameVisible() : boolean
+ synchronized   setTeamNameVisible ( teamNameVisible: boolean) : void
+ synchronized   isScanLinesVisible() : boolean
+   setScanLinesVisible( scanLinesVisible: boolean) : void
+ synchronized   isRobotCollisionsSound() : boolean
+   setRobotCollisionsSound( robotCollisionsSound: boolean) : void
+ synchronized   isRobotEnergyVisible () : boolean
+   setRobotEnergyVisible( robotEnergyVisible: boolean) : void
+ synchronized   isGunShotSound() : boolean
+   setGunShotSound( gunShotSound: boolean) : void
+ synchronized   isWallCollisionsSound() : boolean
+   setWallCollisionsSound( wallCollisionsSound: boolean) : void
+ synchronized   isObjectCollisionsSound() : boolean
+   setObjectCollisionsSound( objectCollisionsSound: boolean) : void
+ synchronized   isDeathExplosionsSound() : boolean
+   setDeathExplosionsSound( deathExplosionsSound: boolean) : void
+ synchronized   isBulletHitSound() : boolean
+   setBulletHitSound( bulletHitSound: boolean) : void
```

Figure 2.4.8 2: BattleBasicOptions Class

This class contains the the controls of the basic options of game such as sound and visiual effects.

3.4.9. Dialog Package and Included Important Classes

This package contains the classes of UI and has integration of Engine and UI.

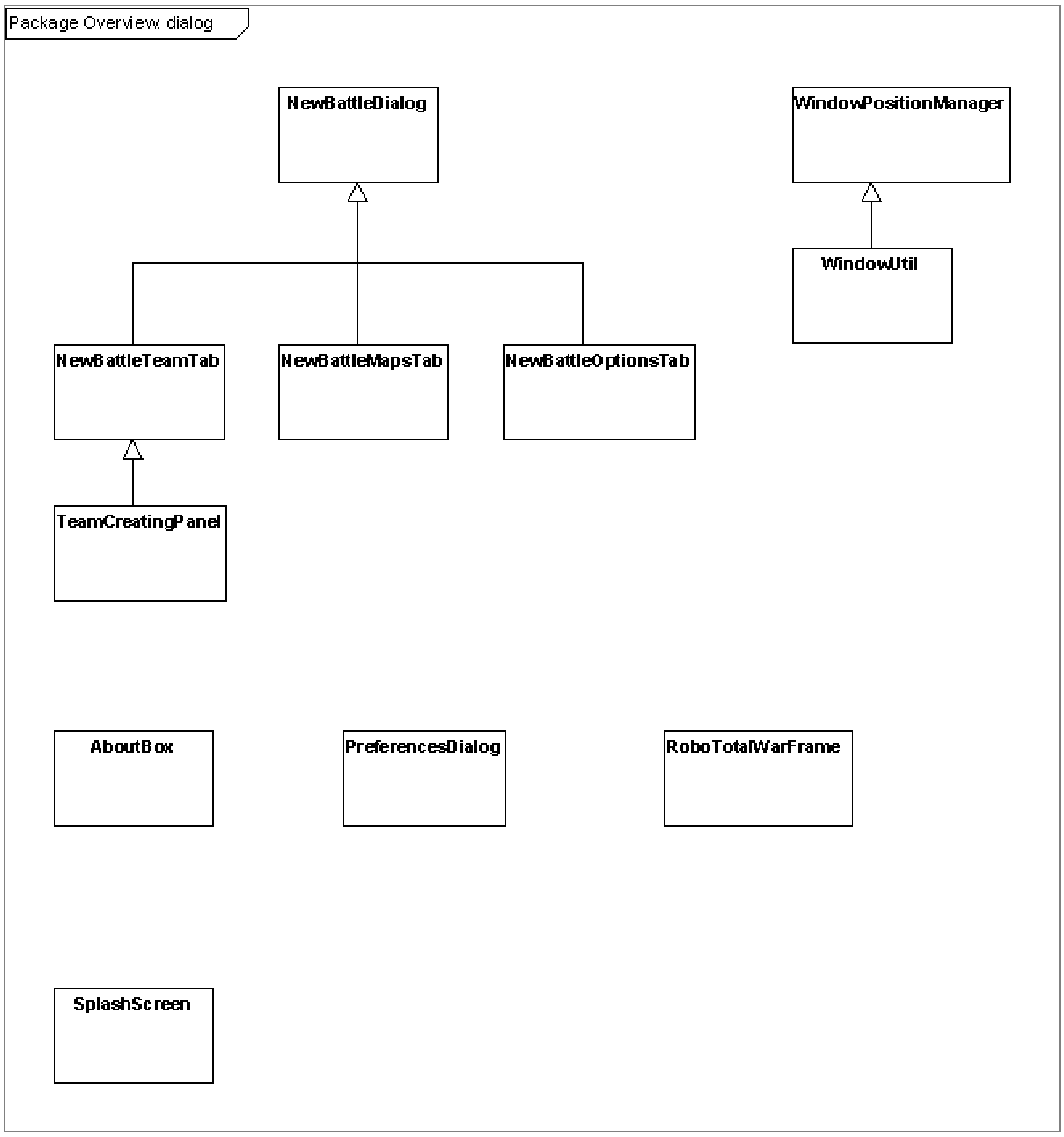


Figure 2.4.9 1: Dialog Package Overview

3.4.10. Exception Package and Included Important Classes

This Package enables us to handle exceptions of our environment other than JAVA exceptions, we define the exception types and solutions, and it helps us to improve and for future enables better maintainability.

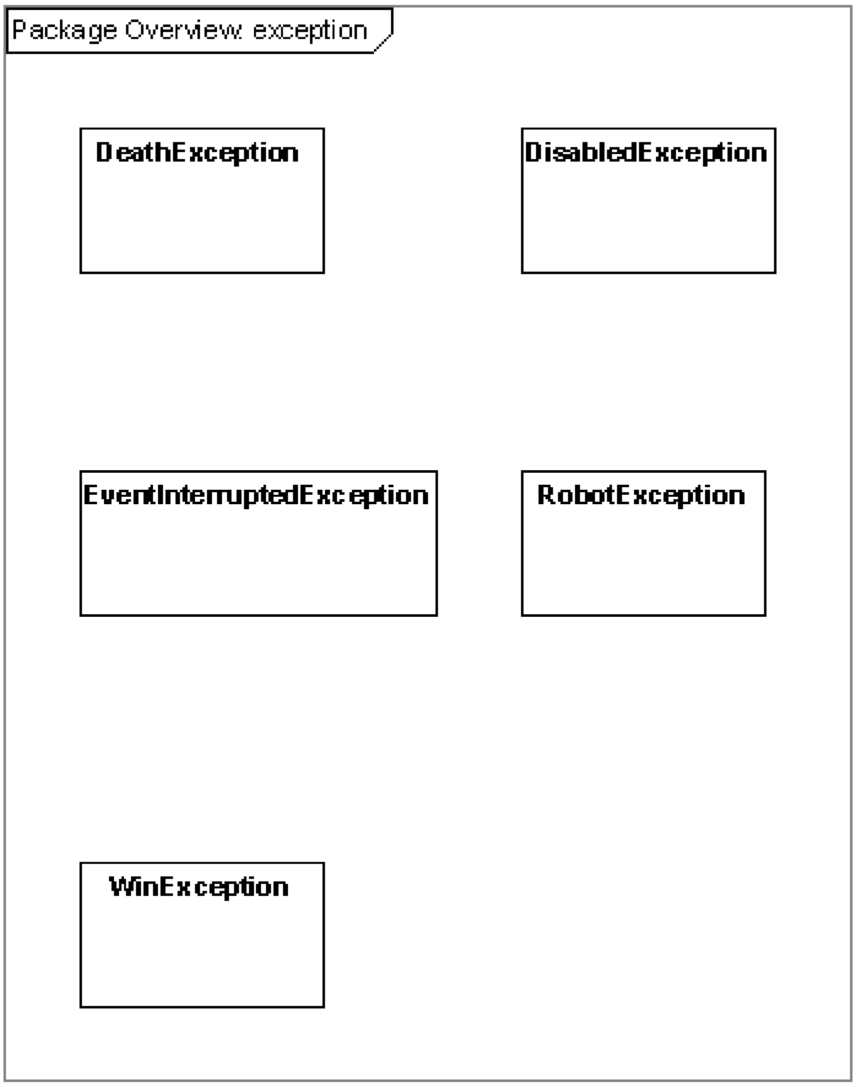


Figure 2.4.10 1: Exception Package Overview

3.4.11. IO Package and Included Important Classes

This Package is used for io processes for saving, loading a battle also we use small file io s for robot communication and AI.

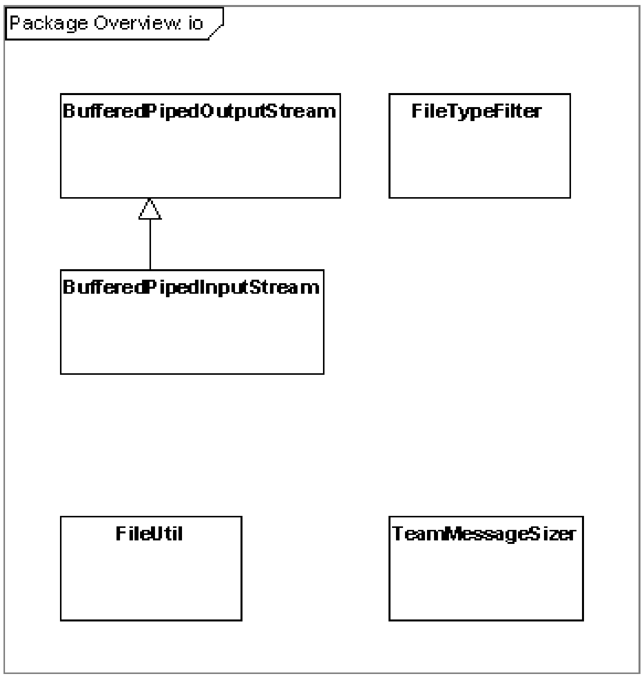


Figure 2.4.11 1: IO Package Overview

3.4.11.1. ResourceCache Abstract Class

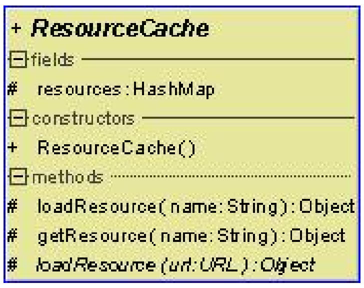

+ ResourceCache
fields
# resources : HashMap
constructors
+ ResourceCache()
methods
# loadResource(name: String) : Object
# getResource(name: String) : Object
# loadResource (url: URL) : Object

Figure 2.4.11 2: ResourceCache Abstract Class

This class is used by Sprite and Sound cache classes as like a prototype.

3.4.12. Manager Package and Included Important Packages

This Package contains the controllers of the game and frame properties etc.

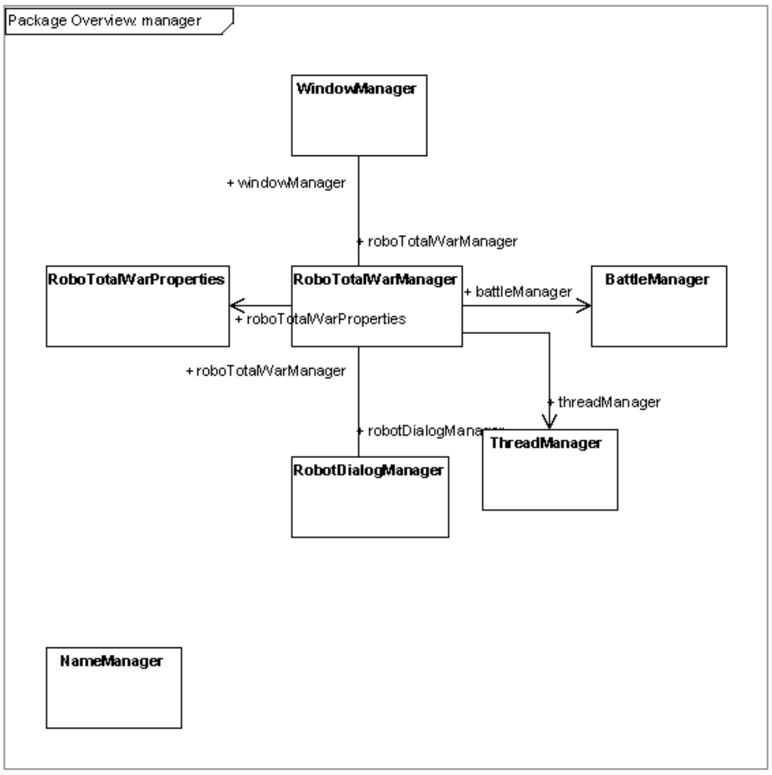
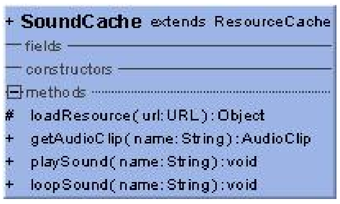


Figure 2.4.12 1: Manager Package Overview

3.4.13. Sound Package and Included Important Classes

This package contains SoundCache class and this class controls the imported sounds and use caching.

3.4.13.1. SoundCache Class



```
+ SoundCache extends ResourceCache
fields
constructors
methods
#  loadResource(url:URL):Object
+  getAudioClip(name:String):AudioClip
+  playSound(name:String):void
+  loopSound(name:String):void
```

Figure 2.4.13 1: SoundCache Class

This class controls the imported sounds and use caching.

3.4.14. Util Package and Included Important Classes

This packages used by other classes when it is needed as image resource, graphic states and utils etc.

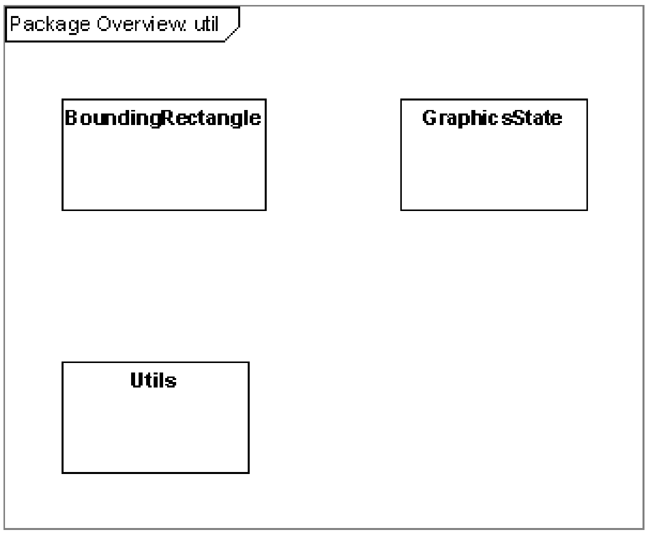
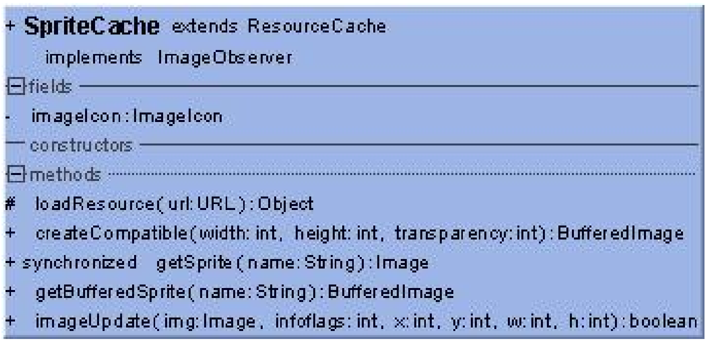


Figure 2.4.14 1: Util Package Overview

3.4.14.1. SpriteCache Class



```
+ SpriteCache  extends ResourceCache
    implements  ImageObserver
⊟fields
-   imageIcon:ImageIcon
— constructors
⊟methods
#   loadResource(url:URL):Object
+   createCompatible(width:int, height:int, transparency:int):BufferedImage
+ synchronized  getSprite(name:String):Image
+   getBufferedSprite(name:String):BufferedImage
+   imageUpdate(img:Image, infoflags:int, x:int, y:int, w:int, h:int):boolean
```

Figure 2.4.14 2: SpriteCache Class

This class is the resource class for the images every image returns from this class, has the buffers and observers for a better image cache results.

4. Testing Part

4.1. Introduction

This part of report is for Verification and Validation Plan for **RoboTotalWar**. The purpose of Verification and Validation Plan is to ensure the assurance of project against customer and users. Verification is intended to show that a program meets its specification. Validation is intended to show that the program does want the user requires.[3] So this part indicates the mutual connection between Requirements Specifation[2] and Design Specification[2].

One of the most important parts of the software process is testing to deliver a complete, qualified, efficient product to the customer. An efficient methodology to testing software product can be implemented considering the existing software testing methods and configuring them.

We will use the testing plan outlined in this part to validate that our software meets user requirements what previously defined in Requirement Specification.[2]

4.2. Testing Scope

Testing of **RoboTotalWar** covers white box testing and testing of the main units of the project and testing user-friendliness of the project. However, testing framework of the project is embedded to unit testing and integration testing

4.3. Testing Strategy

Testing process of **RoboTotalWar** can be divided into three parts:

- Unit testing
- Integration
- Performance Testing

RoboTotalWar can be classified into two units Graphical User Interface and Game Engine. In unit testing the part of the units are tested and the whole unit is tested In testing phase, all units will be tested and integration test will be applied to the system in order to see the persistence of the system.

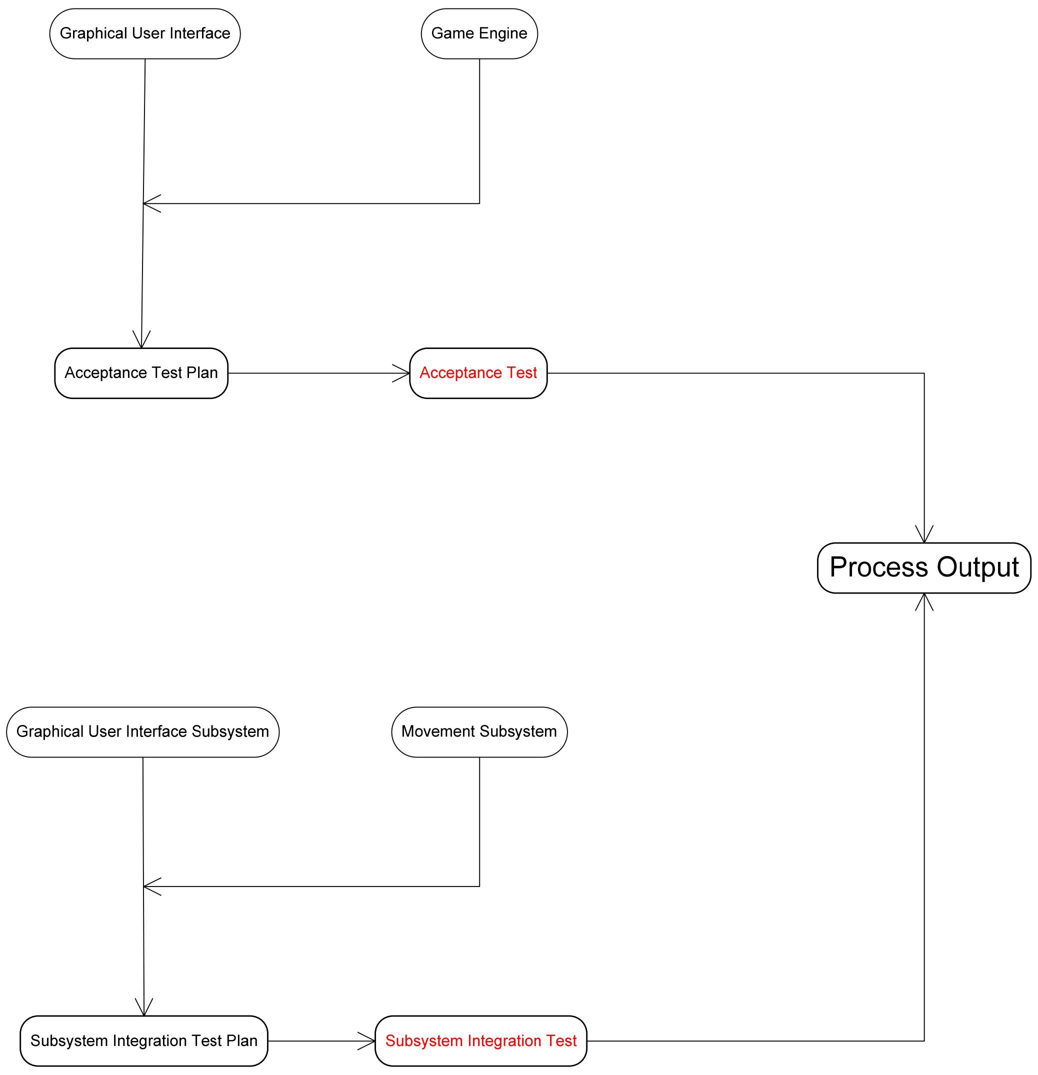


Figure 3.3 1: Testing Strategy Plan

4.3.1. Unit Testing

4.3.1.1. Graphical User Interface Unit

Graphical User Interface unit is responsible for providing connection between game engine and user. This unit designed with fastidiousness using user-friendly software development approach.

Graphical User Interface testing consists of testing smaller parts of Graphical User Interface:

- **New Battle Dialog**

UnitTestID:	**01**
Class(es) to be tested:	**RoboTotalWarFrame, NewBattleDialog, Battle, BattleProperties, Team, Robot, Harita, Light, Weather**
Testing Approach:	Create new Battle, selecting multiple teams with rainy, day light conditions in Akdeniz map.
Pass/Fail Criteria:	**Does the GUI created as expected with selected team and robots?**

Table 3.3.1.1.1

- **Load Battle Dialog**

UnitTestID:	**02**
Class(es) to be tested:	**RoboTotalWarFrame, Battle, Team, Robot, Harita, Light, Weather**
Testing Approach:	Loading previously saved battle.
Pass/Fail Criteria:	**Does the system created the battle from saved battle?**

Table 3.3.1.1.2

- **Save Dialog**

UnitTestID:	**03**
Class(es) to be tested:	**BattleProperties, RoboTotalWarFrame**
Testing Approach:	Saving the running battle for afterward loading.
Pass/Fail Criteria:	**Does the system save the battle as loadable?**

Table 3.3.1.1.3

- **Save As Dialog**

UnitTestID:	**04**
Class(es) to be tested:	**BattleProperties, RoboTotalWarFrame**
Testing Approach:	Saving a loaded battle with a different name.
Pass/Fail Criteria:	**Does the system save the battle as loadable with different name?**

Table 3.3.1.1.4

- **Game Options Dialog**

UnitTestID:	**05**
Class(es) to be tested:	**BattleBasicOptions, GameOptionsDialog**
Testing Approach:	Changing game options in runtime of battle**.**
Pass/Fail Criteria:	**Does the system reacts as expected and changes the options?**

Table 3.3.1.1.5

- **Help Dialog**

UnitTestID:	**06**
Class(es) to be tested:	**RoboTotalWarFrame**
Testing Approach:	Looking for help file.
Pass/Fail Criteria:	**Does the opens help file in system?**

Table 3.3.1.1.6

4.3.1.2. Game Engine Unit

Game Engine unit is responsible for handling battle rules, battle objects, robots and all connected objects within battle. This unit designed for running with high performance and requiring fewer resources. Game Engine Unit testing consists of testing smaller parts of Game Engine using these scenarios:

- **Creating new robot**

UnitTestID:	**07**
Class(es) to be tested:	**Robot, Harita, Bullet, Event, Radar, Shield**
Testing Approach:	Create new IsmetInonu robot in Akdeniz map.
Pass/Fail Criteria:	**Does the system create robot with requested properties in map?**

Table 3.3.1.2.1

- **Creating new team**

UnitTestID:	**08**
Class(es) to be tested:	**Robot, Team**
Testing Approach:	Create new team with name Marmara, team value is 1000 and # of robots is two.
Pass/Fail Criteria:	**Does the system create team with requested properties in map?**

Table 3.3.1.2.2

- **Adding a robot to a team**

UnitTestID:	**09**
Class(es) to be tested:	**Team, Robot**
Testing Approach:	Adding IsmetInonu to the Marmara team.
Pass/Fail Criteria:	**Does the system add IsmetInonu in Marmara team?**

Table 3.3.1.2.3

- **Creating a map**

UnitTestID:	**10**
Class(es) to be tested:	**Harita, GroundCondition, LightCondition, BattleView, Team, Robot**
Testing Approach:	Create new Akdeniz map.
Pass/Fail Criteria:	**Does the system create Akdeniz map with all objects?**

Table 3.3.1.2.4

- **Setting weather conditions**

UnitTestID:	**11**
Class(es) to be tested:	**Weather, Harita**
Testing Approach:	Create new rainy weather in Akdeniz map.
Pass/Fail Criteria:	**Does the system sets the weather of Akdeniz as expected?**

Table 3.3.1.2.5

- **Initializing graphical battle view**

UnitTestID:	**12**
Class(es) to be tested:	**RoboTotalWarFrame, Robot, Bullet,BattleView, NewBattleDialog, Battle, BattleProperties, Team, Robot, Harita, Light, Weather**
Testing Approach:	Create new Battle, selecting Marmara team, rainy condition in Akdeniz map.
Pass/Fail Criteria:	**Does the system create graphical part as expected with selected team and robots?**

Table 3.3.1.2.6

4.3.2. Integration Testing

Integration testing part consists of testing two base units of **RoboTotalWar** together. **Graphical User Interface** and **Game Engine** units' functionalities and achievement of decisions will be tested. Also, system accomplishment over user requirements must be answered.

In this phase, our testing approach, will be to control whether system attitudes as we designed pass/fail criteria will be: Choosing two team with two robot for each team and after running battle and after 30 seconds restarting battle without stopping currently war. After running these conditions test will be assumed to be passed.

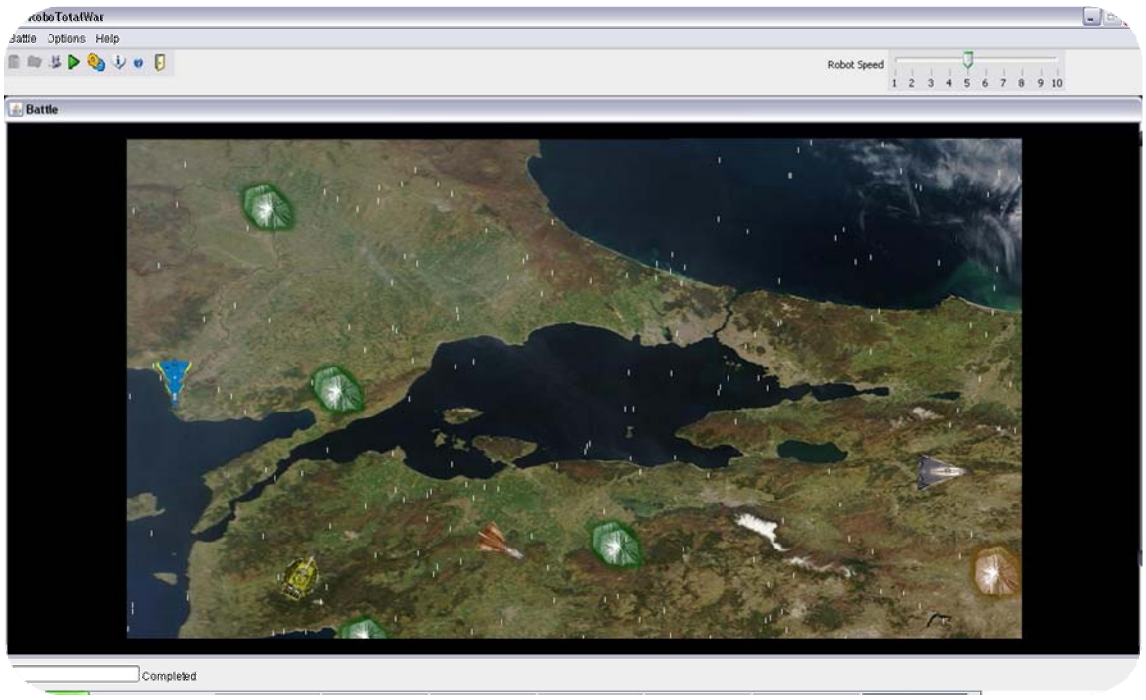


Figure 3.3.2 1: Integration Test Result

4.3.3. Performance Testing

Performance testing part consists testing in different operating systems and different system configurations. We will test the system in Ubuntu 6.11(Linux Distribution) [4] with 1024x768 screen resolution and 1280x800 screen resolutions. And also we will test in Windows XP SP2 [5] with 1024x768 and 1280x800.screen resolutions.

4.4. Testing Results

The Testing Results show connection between the effects the various testing activities described in the Testing Strategy. These can only show the presence of errors in a program. It cannot demonstrate that there are no remaining faults [3]. Testing strategies were planned to supply that the system and its components work as desired level thought the various development stages.

4.4.1. Unit Testing Results

As we designed in Testing Strategy, unit testing results according the selected criteria is as follows:

Unit Test ID	Criteria Test Result	Problems	Solutions
01	Pass	None	
02	Fail	The system doesn't load previously saved battle.	Working on solution.
03	Pass	System cannot load saved file properly	Changing the file structure of battle saving and work on solution in this way.
04	Pass	System cannot load saved file properly	Changing the file structure of battle saving and work on solution in this way.
05	Pass		
06	Pass		
07	Pass		
08	Pass		
09	Pass		
10	Pass		
11	Pass	In DoguAnadolu and IcAnadolu map, the initializing the map works incorrectly.	Working on solution.
12	Pass	In DoguAnadolu and IcAnadolu map, the initializing the map works incorrectly.	Working on solution.

Table 3.4.1: Unit Test Results

4.4.2. Integration Testing Results

In this test, sample two team with two robots for each team and after running battle and after 30 seconds restarting battle without stopping currently war. After running these conditions test is accepted as “Pass”.

4.4.3. Performance Testing Results

In performance tests, we aimed perceiving advantages of using cross-platform development language Java. So we tested our software on Linux distribution Ubuntu and Microsoft Windows XP. After testing we perceived our software runs platform independently.

5.Time Plan and Roadmap

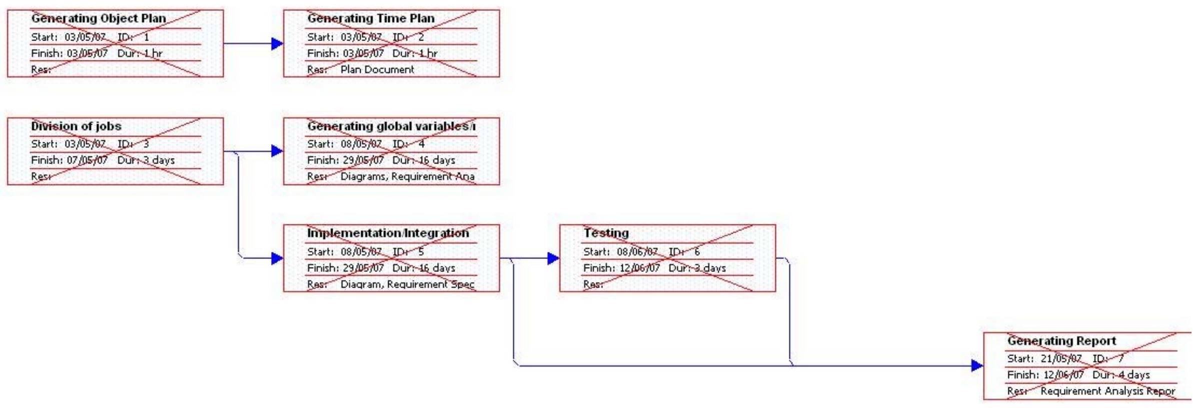
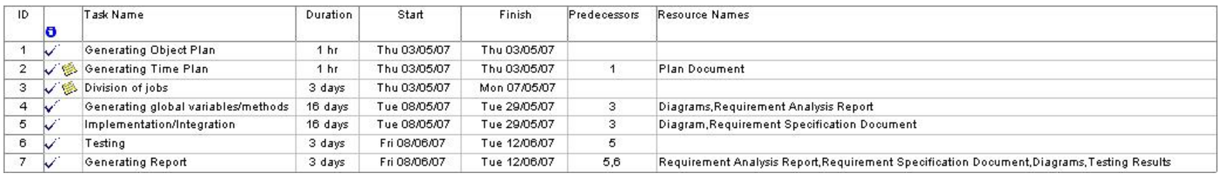

Generating Object Plan
Start: 03/05/07 ID: 1
Finish: 03/05/07 Dur: 1 hr
Res:
Generating Time Plan
Start: 03/05/07 ID: 2
Finish: 03/05/07 Dur: 1 hr
Res: Plan Document
Division of jobs
Start: 03/05/07 ID: 3
Finish: 07/05/07 Dur: 3 days
Res:
Generating global variables/r
Start: 08/05/07 ID: 4
Finish: 29/05/07 Dur: 16 days
Res: Diagrams, Requirement Ana
Implementation/Integration
Start: 08/05/07 ID: 5
Finish: 29/05/07 Dur: 16 days
Res: Diagram, Requirement Spec
Testing
Start: 08/06/07 ID: 6
Finish: 12/06/07 Dur: 3 days
Res:
Generating Report
Start: 21/05/07 ID: 7
Finish: 12/06/07 Dur: 4 days
Res: Requirement Analysis Repor

Figure 5 1: Project Road Map Flow Diagram



ID		Task Name	Duration	Start	Finish	Predecessors	Resource Names
1	✓	Generating Object Plan	1 hr	Thu 03/05/07	Thu 03/05/07		
2	✓	Generating Time Plan	1 hr	Thu 03/05/07	Thu 03/05/07	1	Plan Document
3	✓	Division of jobs	3 days	Thu 03/05/07	Mon 07/05/07		
4	✓	Generating global variables/methods	16 days	Tue 08/05/07	Tue 29/05/07	3	Diagrams,Requirement Analysis Report
5	✓	Implementation/Integration	16 days	Tue 08/05/07	Tue 29/05/07	3	Diagram,Requirement Specification Document
6	✓	Testing	3 days	Fri 08/06/07	Tue 12/06/07	5	
7	✓	Generating Report	3 days	Fri 08/06/07	Tue 12/06/07	5,6	Requirement Analysis Report,Requirement Specification Document,Diagrams,Testing Results

Figure 5 2: Project Timeline

6. Conclusion

This project enables java programmers to develop their own robots within the constraints of our environment project they will get the API from the web site(httt://www.sourceforge.net/robototal) and the system contains many constraints to maintain in a good structure and try to stop the fakes.
As the necessity of the component based model software developing, we developed this project in an order and as building a wall.

In Requirements Specification, we defined what kind of operations robots and end users should do. After that design of the process model architecture and V&V (testing) phase becomes. By applying these stages of the software process, we complete the integration of the components each other.

Actually main part as we have seen is design process, because it contains how the components work each other, how they will be more effective use, how the encapsulation should be done.
The layers have been used, are commonly basic but inside the engine layer contains many controls, because of the every other condition checks.

In concluison, we think that our environment has some weaknesses if we compare with Robocode, but also our project contains more conditions to be adaptable and more learning for robots that’s why end users(programmers) may have difficulties when they program their robots, it doesn’t mean to hard to understand, it is only hard to develop because of the more controls.

7. Future Expectations

For maintainability we use component based software engineering and used encapsulation that in most limits of usage, and try to get the best results from OOP. We have everything in objects, that helps us to control the development process, well understood the faulties etc.

For future expectations, we try to develop our project in JAVA 3D. Why Java3D because, it is in the same manner to use OOP, and getting best results of platform independancy. The end user have also here a role to get the well understood requirements to develop a robot in our platform.

8. References

[1] http://robocode.sourceforge.net

[2] http://robototal.sourceforge.net/

[3] Software Engineering, Ian Summerville, 8th edition

[4]http://www.ubuntu.com

[5]http://www.microsoft.com