Missile Combat

Luis Carlos Mateos Cañas
wuiscmc@gmail.com

About this document

This document contains the documentation of the project Missile Combat. It contains the reasons behind most of the decisions taken during the development of the project and how it would be possible to improve the system in the future if needed.

Main structure

The project follows clearly the Model View Controller architectural pattern. This pattern allows to organize the code in three layers:

- View
  The responsibility of this layer is to display the entities of the system and to interact with the user.

- Controller
  Known as well as the expert of the application, the role controller provides a mechanism to the view so it can establish a communication with the model of the application.

  The controller could be bidirectional or unidirectional. In both cases, the controller separates the view from the model, the difference relies on the purpose of the controller. In the first approach, bidirectional, the controller receives the requests of the view and the responses from the model. In the second approach, unidirectional, the controller knows how to map the views requests into model actions, it does not receive any response from the model since it communicates with other elements through observers.

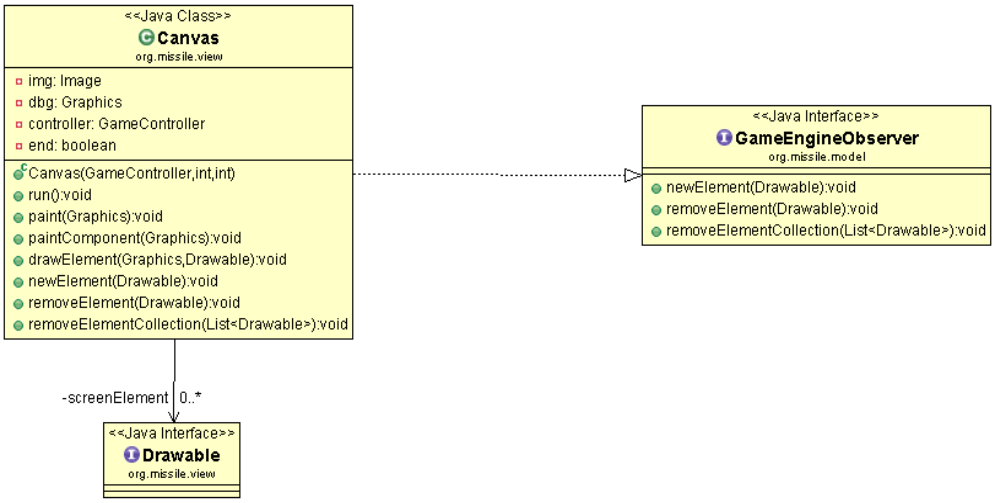
- Model
  Contains the logic of the application and the business models.

The MVC pattern is a good solution to maintain the different parts of the application decoupled. A system designed in this way will be a more scalable system and easier to maintain.

View

Main class: Canvas.java
Package : org.missile.view

As said previously, the view layer knows how to represent the different elements of the system and to interact with the user.



Canvas is a subclass of **JFrame**.

Double buffering

The technique used to refresh the screen is double buffering. This is handled in the functions **paint** and **paintComponent**.

Observer

The main class **Canvas** implements the interface **GameEngineObserver**. The view then listens the events which take place in the model of the application.

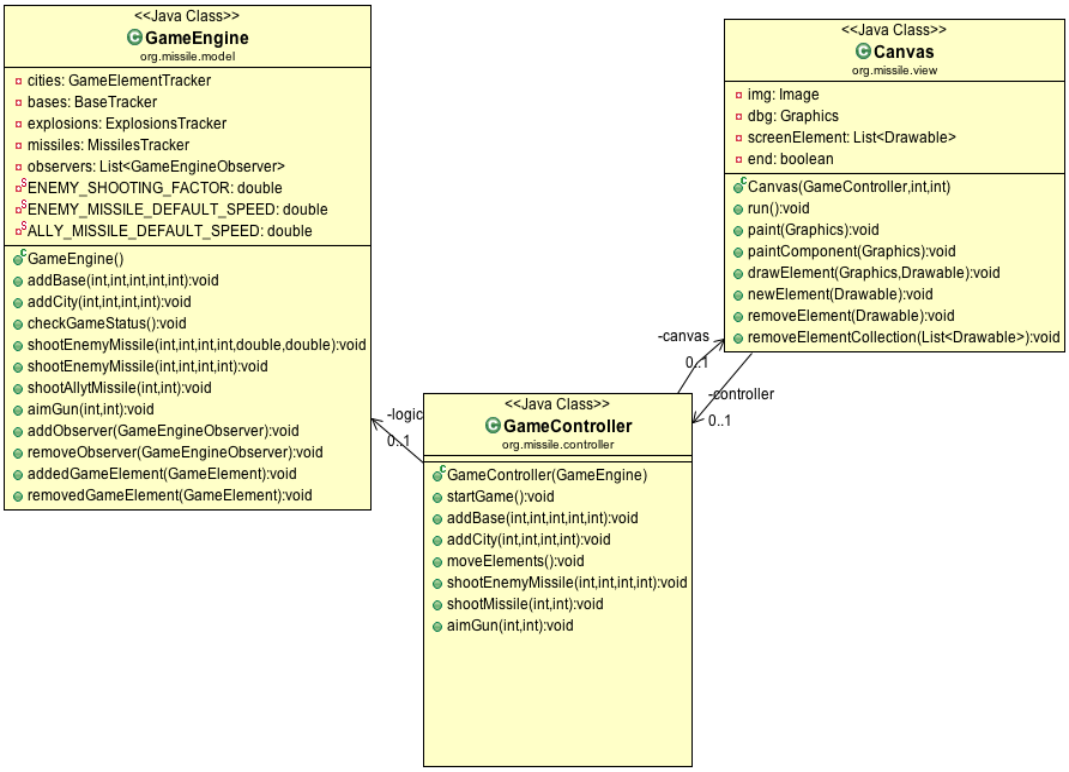
Painting an element

Once it is notified that a event in the system has occured, Canvas calls the function **drawElement**. Within that function we paint the components of the application.
Controller.

Controller

Main class: GameController.java
Package : org.missile.controller

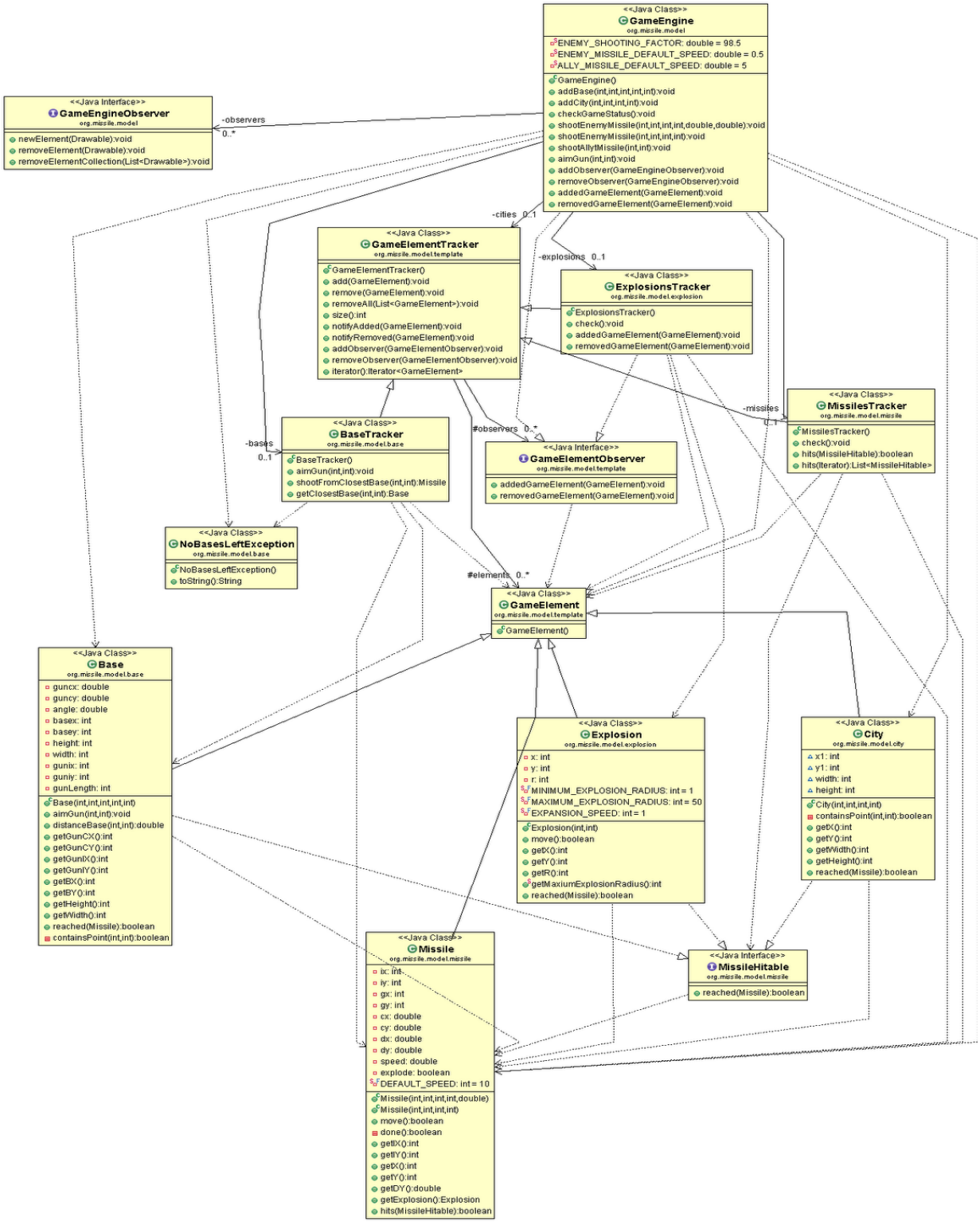
GameController is initialized in the main function of the system. It contains a pointer to the **GameEngine** (model) and also it creates the Canvas and subscribes it to the model listeners list.



The **GameController** has an association to **GameEngine** so it can send requests. Those requests will not travel back to the view from the model and go through the Controller, they will instead be notified by a observer.

The main responsibility of this class is to initiate the game.

Model


<<Java Class>>
GameEngine
org.missile.model
ENEMY_SHOOTING_FACTOR: double = 98.5
ENEMY_MISSILE_DEFAULT_SPEED: double = 0.5
ALLY_MISSILE_DEFAULT_SPEED: double = 5
GameEngine()
addBase(int,int,int,int,int):void
addCity(int,int,int,int):void
checkGameStatus():void
shootEnemyMissile(int,int,int,int,double,double):void
shootEnemyMissile(int,int,int,int):void
shootAllyMissile(int,int):void
aimGun(int,int):void
addObserver(GameEngineObserver):void
removeObserver(GameEngineObserver):void
addedGameElement(GameElement):void
removedGameElement(GameElement):void
<<Java Interface>>
GameEngineObserver
org.missile.model
newElement(Drawable):void
removeElement(Drawable):void
removeElementCollection(List<Drawable>):void
-observers
0..*
-cities 0..1
<<Java Class>>
GameElementTracker
org.missile.model.template
GameElementTracker()
add(GameElement):void
remove(GameElement):void
removeAll(List<GameElement>):void
size():int
notifyAdded(GameElement):void
notifyRemoved(GameElement):void
addObserver(GameElementObserver):void
removeObserver(GameElementObserver):void
iterator():Iterator<GameElement>
-explosions 0..1
<<Java Class>>
ExplosionsTracker
org.missile.model.explosion
ExplosionsTracker()
check():void
addedGameElement(GameElement):void
removedGameElement(GameElement):void
-missiles
0..1
<<Java Class>>
MissilesTracker
org.missile.model.missile
MissilesTracker()
check():void
hits(MissileHitable):boolean
hits(Iterator):List<MissileHitable>
-bases
0..1
<<Java Class>>
BaseTracker
org.missile.model.base
BaseTracker()
aimGun(int,int):void
shootFromClosestBase(int,int):Missile
getClosestBase(int,int):Base
#observers 0..*
<<Java Interface>>
GameElementObserver
org.missile.model.template
addedGameElement(GameElement):void
removedGameElement(GameElement):void
<<Java Class>>
NoBasesLeftException
org.missile.model.base
NoBasesLeftException()
toString():String
#elements 0..*
<<Java Class>>
GameElement
org.missile.model.template
GameElement()
<<Java Class>>
Base
org.missile.model.base
guncx: double
guncy: double
angle: double
basex: int
basey: int
height: int
width: int
gunix: int
guniy: int
gunLength: int
Base(int,int,int,int,int)
aimGun(int,int):void
distanceBase(int,int):double
getGunCX():int
getGunCY():int
getGunIX():int
getGunIY():int
getBX():int
getBY():int
getHeight():int
getWidth():int
reached(Missile):boolean
containsPoint(int,int):boolean
<<Java Class>>
Explosion
org.missile.model.explosion
x: int
y: int
r: int
MINIMUM_EXPLOSION_RADIUS: int = 1
MAXIMUM_EXPLOSION_RADIUS: int = 50
EXPANSION_SPEED: int = 1
Explosion(int,int)
move():boolean
getX():int
getY():int
getR():int
getMaxiumExplosionRadius():int
reached(Missile):boolean
<<Java Class>>
City
org.missile.model.city
x1: int
y1: int
width: int
height: int
City(int,int,int,int)
containsPoint(int,int):boolean
getX():int
getY():int
getWidth():int
getHeight():int
reached(Missile):boolean
<<Java Class>>
Missile
org.missile.model.missile
ix: int
iy: int
gx: int
gy: int
cx: double
cy: double
dx: double
dy: double
speed: double
explode: boolean
DEFAULT_SPEED: int = 10
Missile(int,int,int,int,double)
Missile(int,int,int,int)
move():boolean
done():boolean
getIX():int
getIY():int
getX():int
getY():int
getDY():double
getExplosion():Explosion
hits(MissileHitable):boolean
<<Java Interface>>
MissileHitable
org.missile.model.missile
reached(Missile):boolean

Main class: GameEngine.java
Package: org.missile.model

The model is in charge of processing the logic of the system. This model is built around the Facade[1] class **GameEngine**. The purpose of using that pattern is to provide a simplified interface which makes the system easier to understand.

To avoid dependencies with classes outside the model, **GameEngine** is conceived as an observable class which manages a collection of **GameEngineObserver** [2]. Any time GameEngine modifies its state, it will notify to its observers. This is how the **Canvas** receives the order to refresh the screen for instance. In the same way, this mechanism could be used to write modules which makes use of the model data, such as score system, statistics of use, etc. everything done in a decoupled way.

Regarding the business models, they have been grouped into a **GameElement** object. This object represents a single element in the game and it is managed by a **GameElementTracker**. The tracker is a collection handler class which is in charge of the management of a set of game elements, it knows how to add a element to a list, and it allows third classes to listen for the events that occur inside this class. If a missile has reached its goal or has impacted with a city or base, the listeners could handle the event and update its state somehow. The best explanation would be a sequence explanation:

1. MissileTracker performs a check of its status.
2. There is a Missile which has reached it's destiny, therefore it is removed.
3. After the removal, MissileTracker notifies its listeners that a element has been removed.
4. ExplosionsTracker, a listener of MissileTracker events, gets the notification and creates an explosion at the coordinates where the Missile was removed.

[1] Facade Design Pattern. http://en.wikipedia.org/wiki/Facade_pattern

[2] Observer Design Pattern http://en.wikipedia.org/wiki/Observer_pattern

Additional improvements

This design could be improved, especially the design of the view layer by using design patterns such as the Abstract Factory[3]. We would avoid dependencies between the view main class and the main business classes. Also we should then map the business classes with their representation classes in the view layer, like for example **Missile** and **MissileRepresentation**. The responsibility of MissileRepresentation would be to know how to paint a Missile on the screen.

Following such a structure, the Canvas would listen to the GameElement's events and would delegate the knowledge about how to represent a business model to a AbstractRepresentation class which would be initialized with the same Drawable object that we receive in the listener method.

[3] Abstract Factory Design Pattern. http://en.wikipedia.org/wiki/Abstract_factory_pattern