Emma Lindholm, Jacob Gideflod,
Simon Nielsen & Andreas Pegelow

Grupp 15, TDA367
Chalmers Tekniska Högskola

System Design Document for Meteor Defense

Contents

Version: 1.0

Date: 2014-05-20

Authors: Jacob Gideflod, Emma Lindholm, Simon Nielsen and Andreas Pegelow

This version overrides all previous versions.

1. Introduction

1.1 Design goals

The design should be as loosely coupled as possible to enable replacement of any module in the architecture. For usability see RAD.

1.2 Definitions, acronyms and abbreviations

- **GUI**, graphical user interface.
- **Java,** platform independent programming language.
- **JRE**, the Java Runtime Environment. Additional software needed to run a Java application.
- **MVC**, a way to partition an application with a GUI into distinct parts avoiding a mixture of GUI-code, application code and data spread all over.
- **Screen**, class from the library libGDX representing a **Controller** (see MVC pattern) in the applications hierarchy.
- **Renderer**, representing the **View** (see MVC pattern) in the applications hierarchy.
- **Visitor**, pattern to make the code open for extensions and to avoid the use of “instanceof”.
- **Armory**, a place where you can manage (sell, buy, upgrade) your armory items.
- **ArmoryItem**, items that the player use in game to destroy meteors and protect the city.
- **Continent**, it is the top level element in the level choosing hierarchy. A continent contains several cities.
- **City**, each city represent a different “level”. They have different amount of life and different kind of meteorshowers to achieve variation among the levels.
- **Meteor**, object falling from the sky that the player should prevent from hitting the city.
- **Projectile**, object fired by the cannon barrel that damages meteors.

2. System design

2.1 Overview

The application will use a modified MVC model. A modified version of the Visitor pattern is also used to enable extensions.

2.1.1 MainActivity

When the user starts the application the android system will call the onCreate method in the activity that is declared as the launcher activity in the AndroidManifest. In our case it is MainActivity in the MeteorDefense-android project. MainActivity initializes the libGDX project

MeteorDefense which will result in a call to the create method in the game class MeteorDefense.

2.1.2 MeteorDefense

This is the applications main controller which is responsible for the main flow of the application. It controls which screen that is active (see figure 1 for screen flowchart) and when the application state should be saved.

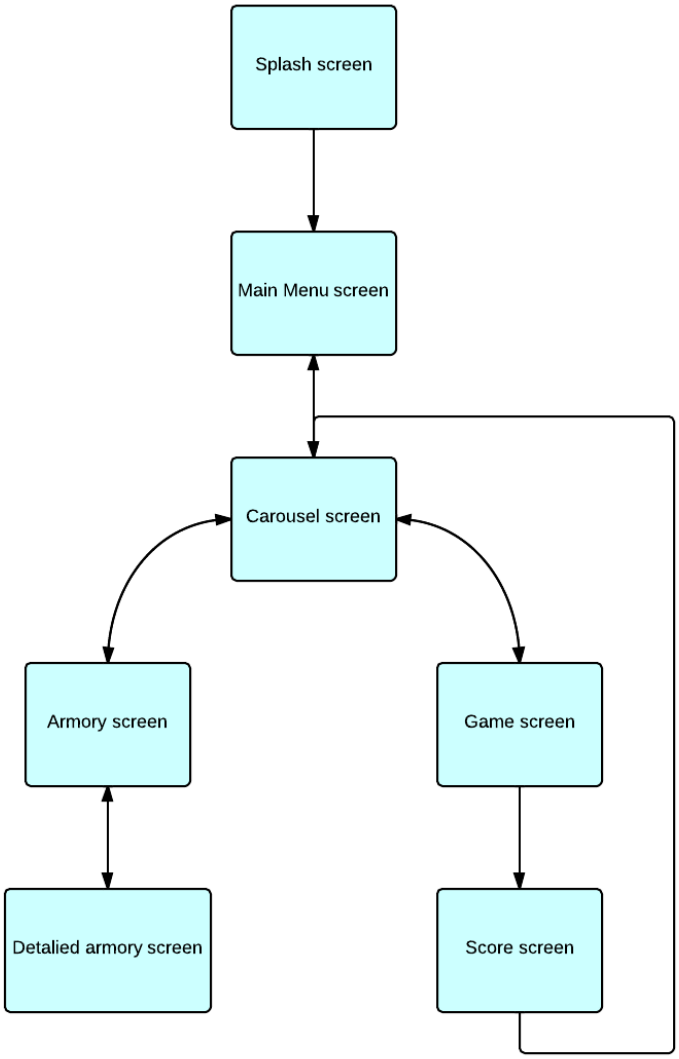

Figure 1, screen flowchart

2.1.3 Model

The models functionality will be exposed by the interface IGameModel (figure 2). This gives us a loose coupling between the model and the renderer.

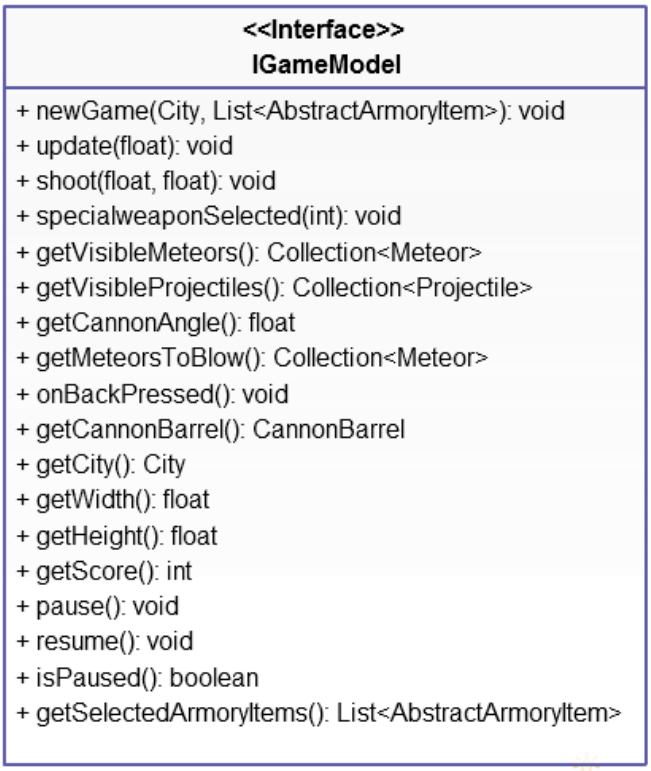

Figure 2, IGameModel

2.1.3 Renderer

All screens have their own renderer which contains all the view code. The view code can be written in the screen classes but to achive a program structure closer to the mvc model this code was extracted to separate renderers.

2.1.4 Factory

The factory package is responsible for the creation of all screens which involves creating the most of the application. It utilizes the facade pattern to hide the concrete factories and provide a well defined interface for the client to work towards. This makes the couplings between the classes loose which enables replacement of the concrete classes.

2.1.5 Service

The works in a similar way as the factory package. It is responsible for loading and saving application state.

2.1.6 MoveableGameObject

Because we have more than one object that are supposed to move on the gamescreen we decided to create the abstract class MoveableGameObject and let the classes projectile and meteor inherit it.

2.1.7 ArmoryItems

The armory items are divided into three different categories since they operate differently:
The defensive armory items needs an instance of the city, to be able to heal or defend it.
The physical effect type of armory items needs the list of visible meteors to be able to reverse their gravity or slow their speed down etc.
The armory items who fire a projectile does not need any object references to operate, but does however return a projectile which the cannon barrel should be loaded with.
These three types all define each abstract class; AbstractDefenseArmoryItem, AbstractEffectArmoryItem and AbstractProjectileArmoryItem.
The methods and attributes that these classes have incommon is defined by the abstract class AbstractArmoryItem which they all inherit from.

Since the list of chosen armory items to be used during the level has the generic type AbstractArmoryItem, but the different subclasses operates differently, the Visitor pattern is used to determine which subclass the fired armory item is an instance of. The pattern has been slightly altered by giving a return value to be able to retrieve the projectile sent by the AbstractProjectileArmoryItem objects.

2.2 Software decomposition

2.2.1 General

Package diagram (figure 2). For each package an UML class diagram in appendix

- **MeteorDefense** is class holding main-method, application entry point.
- **factory**, classes for composing of larger systems of classes.
- **model**, the core object model of the game. Model part of MVC.
- **armoryitem**, subpackage to model containing the various ArmoryItem classes for organisation only. Model part of MVC.
- **meteor**, subpackage to model containing the various Meteor classes for organisation only. Model part of MVC.
- **renderer**, view classes of the MVC.
- **screen**, control classes of the MVC.
- **service**, classes for loading and saving data.
- **test**, classes for testing of the model.
- **util**, various util classes for sound, fonts, constants etc.

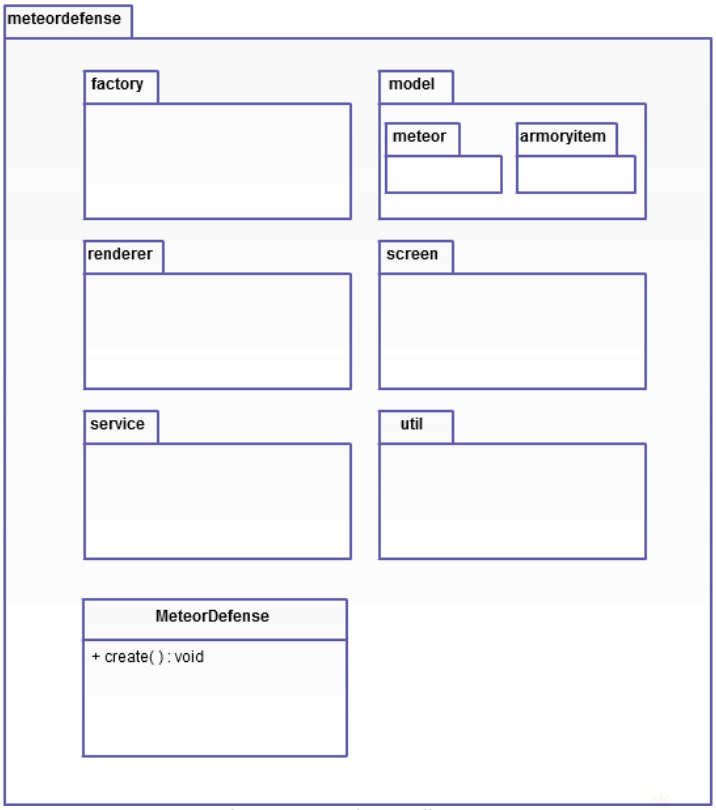

Figure 3, package diagram

2.2.2 Decomposition into subsystems

The only subsystems are service and factory.

2.2.3 Layering

The layering is as indicated in the figure below.

2.2.4 Dependency analysis

The dependencies are as shown in the figure 3. Only circular dependencies are between util and renderer, since util contains Constants and AssetsLoader.

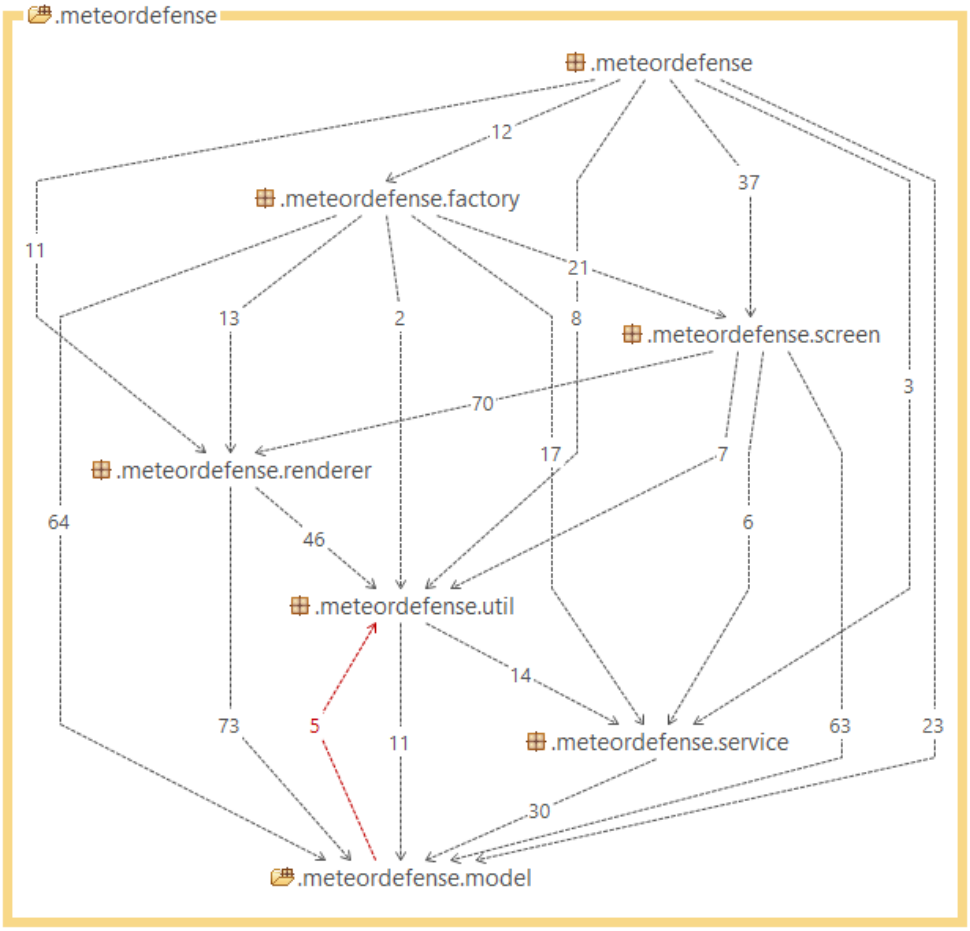

Figure 4, dependency analys

2.3 Concurrency issues

NA. Single threaded application. The thread is provided by the library libGDX that the application is dependent on. No concurrency issues will be raised.

2.4 Persistent data management

The persistent data will be stored in xml- and JSON-files both on desktop and Android with the use of the libGDX xmlwriter/-reader and JSON serializer. The following files will be included:
An xml document with all the filenames for the textures of each object in the game to be visualized.
Levels.xml with information about each level that has to do with the game logic such as life and the meteorshower.
A list of all the continents with their names and the cities/levels they contain.
The armoryitems, soundstate and wallet serialized with all their information.

2.5 Access control and security

NA, this application does not use any kind of internet communication nor does it use any sort of user accounts and therefore the security aspect is not relevant.

2.6 Boundary conditions

Due to lack of time the app will not be released on google play before deadline. Therefor the app will not be able to install from google play the normal way. Instead applications from unknown sources must be enabled on the targeted android phone. Then the MeteorDefense.apk file must be copied to the devices memory card. After that the “apk installer” application must be downloaded from google play and used to install the MeteorDefense.apk located on the memory card.

2.7 Sequence diagrams

There are two sequence diagrams, one for choosing a city to play and one for hitting and killing a meteor. Both are listed in the appendix.

3. References

MVC, see http://sv.wikipedia.org/wiki/Model-View-Controller (2014-05-13)
JSON, see http://www.json.org/ (2014-05-13)
libGDX, see http://libgdx.badlogicgames.com/ (2014-05-20)

APPENDIX

Sequence diagram for choosing a city to play:

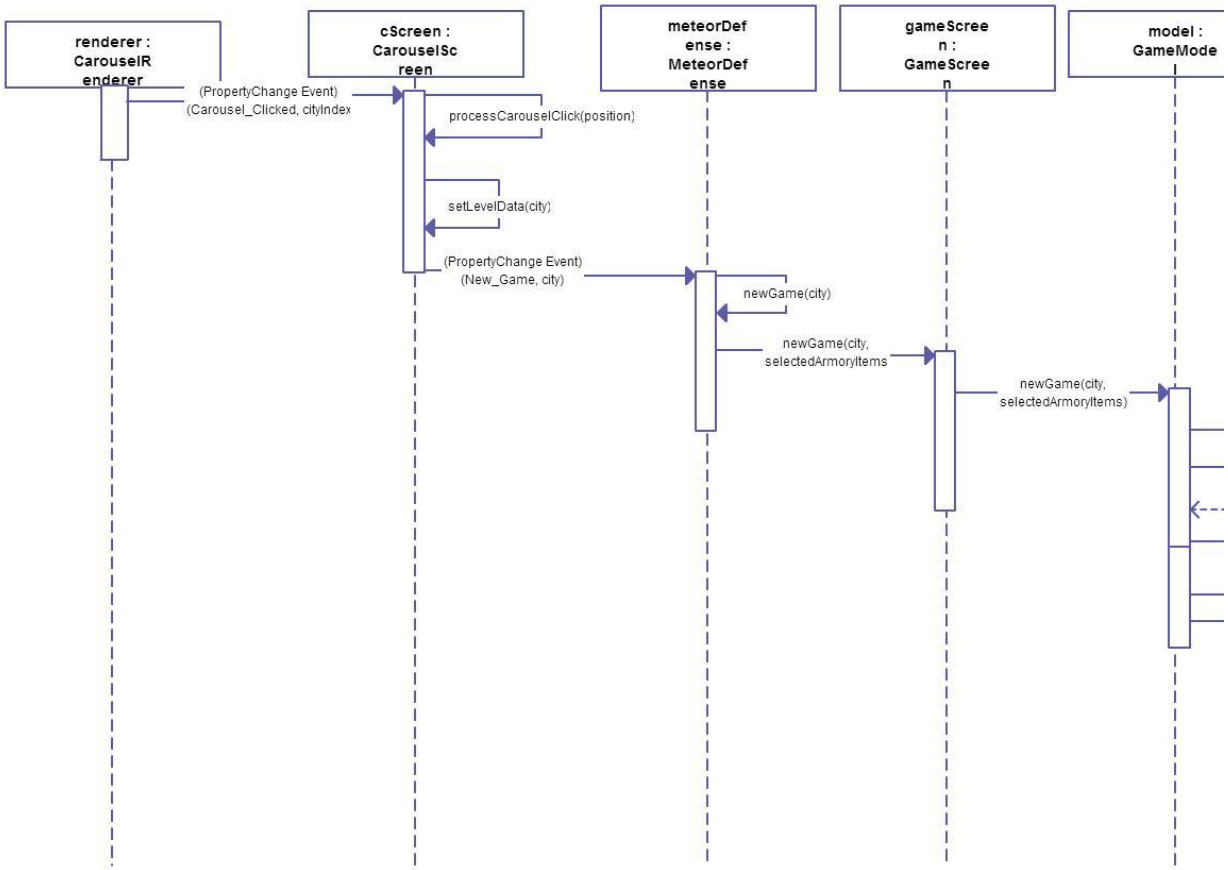

Figure 5, sequence diagram 1 part 1

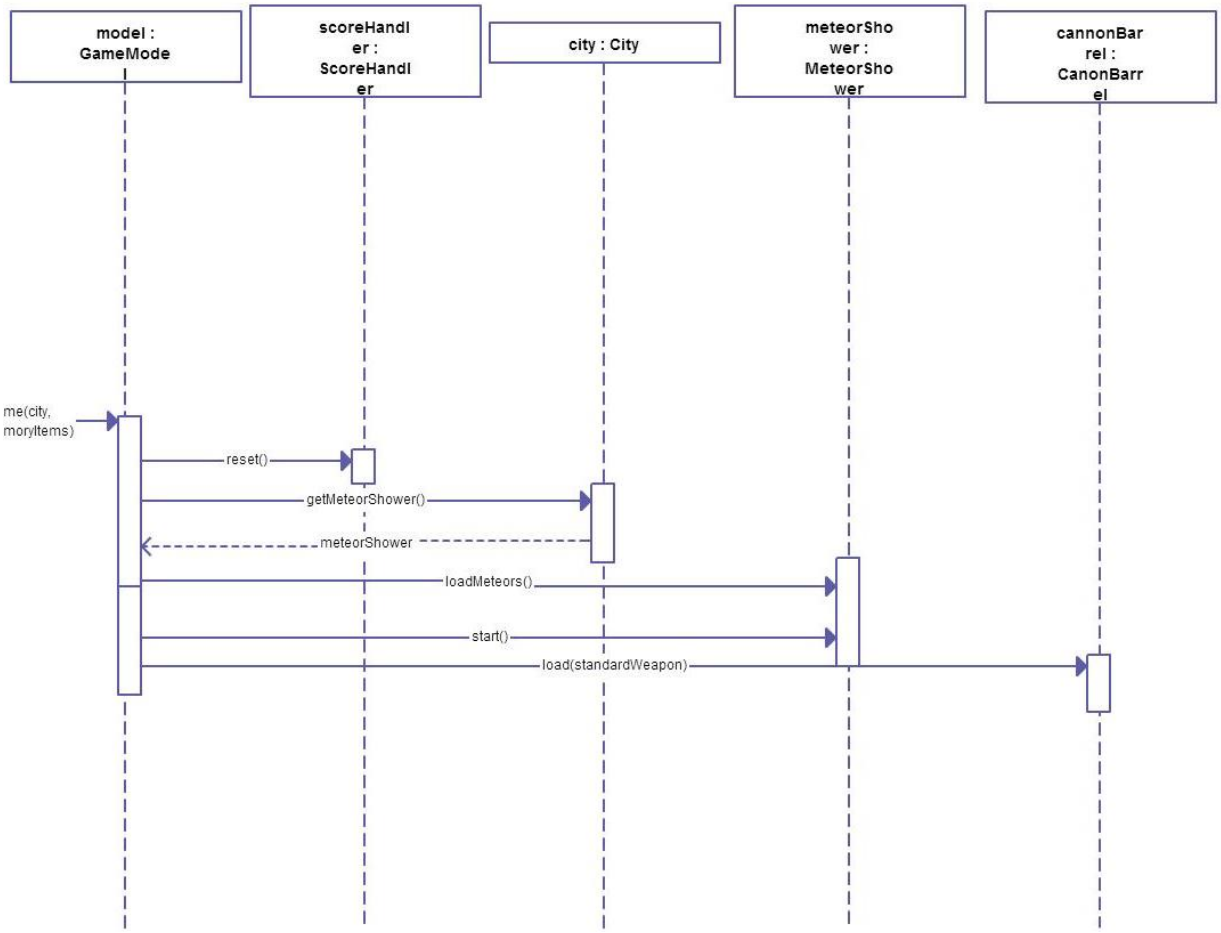

Figure 6, sequence diagram 1 part 2

Sequence diagram for hitting a meteor:

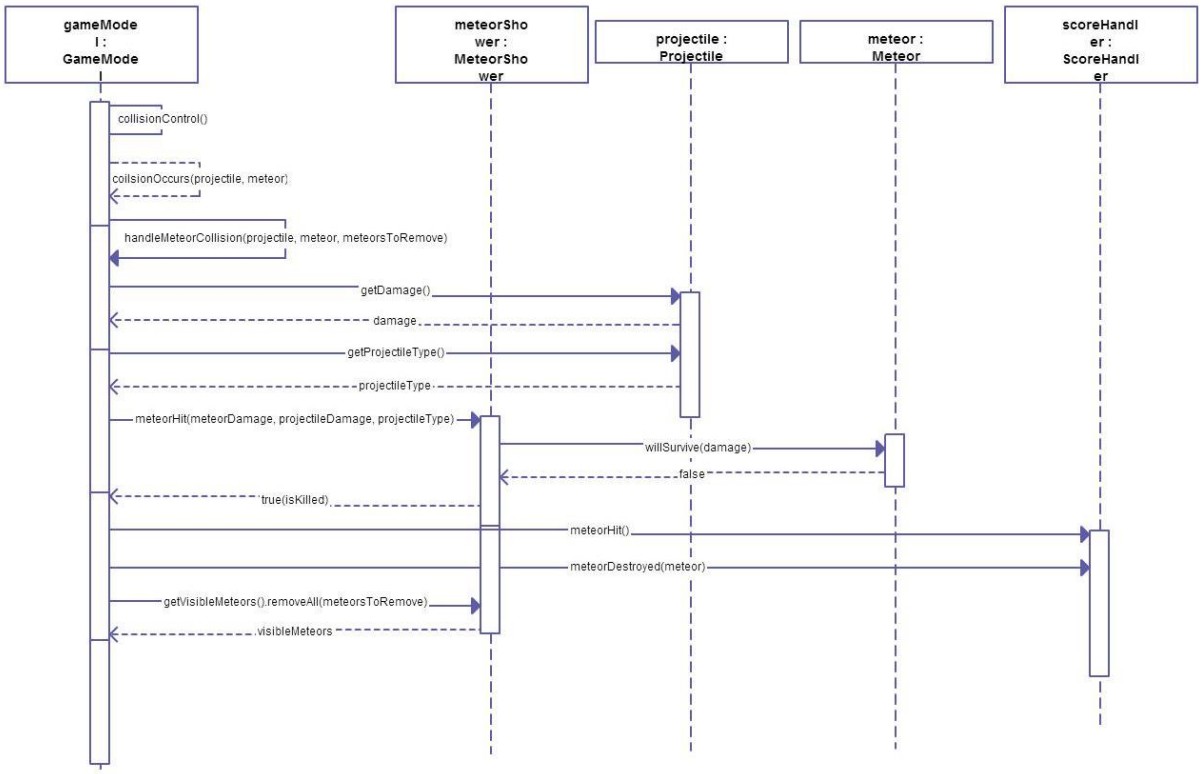

Figure 7, sequence diagram 2

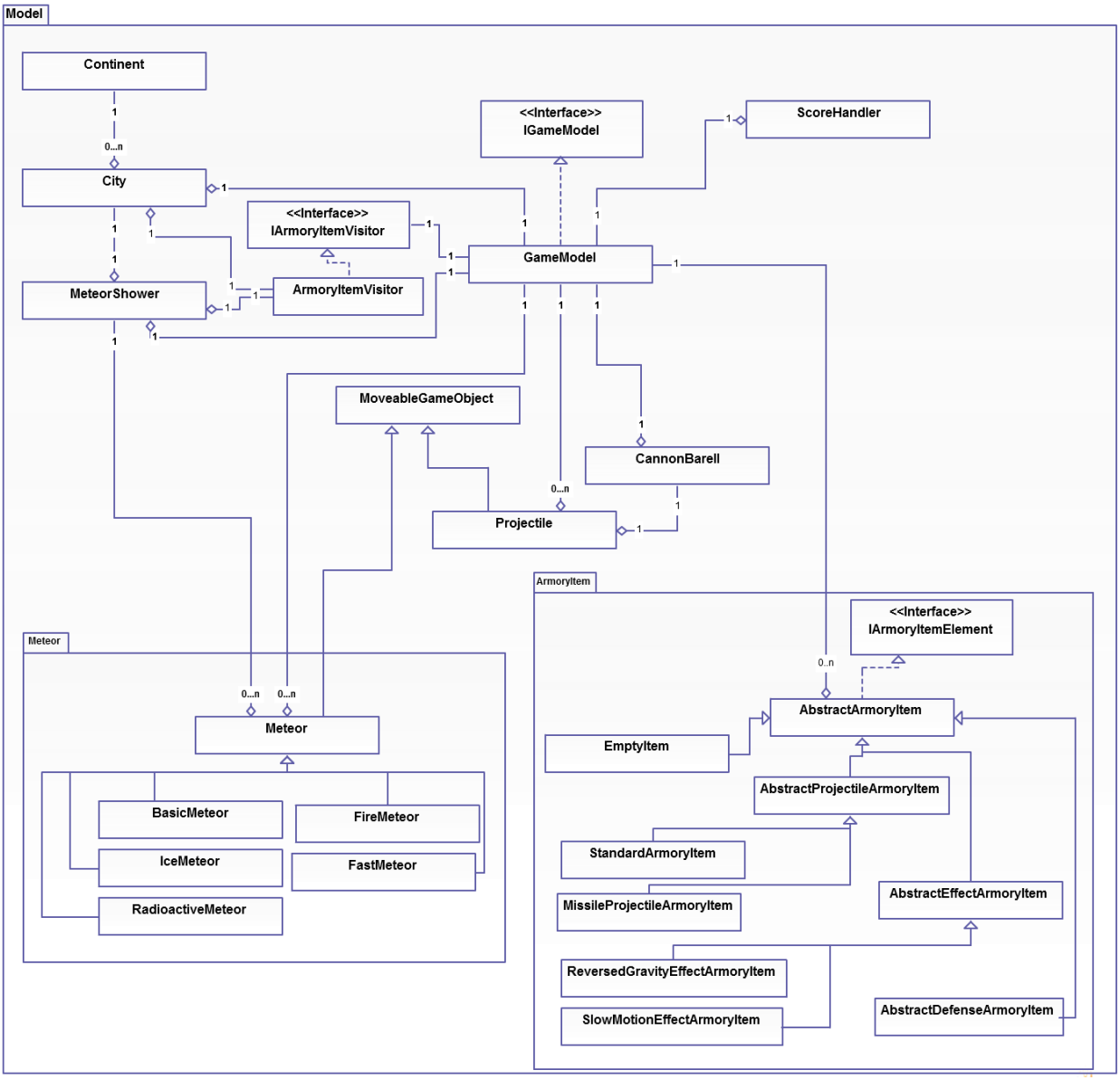

Figure 8, model package

- **Continent** is a container for cities
- **City** represents a level and contains all individual data.
- **MeteorShower** is responsible for keeping track of all meteors and deploying them at the right time. It is also partly involved in the gameover control.
- **Meteor** is a super class for all concrete implementation of meteors. It contains data such as damage and methods to implement different behaviours when being hit by a projectile.
- **GameModel** the core of the game logic where collision control and gameover control takes place among other tasks.
- **CannonBarrel** is responsible for deploying a projectile with the correct angle and startposition.

- **MoveableGameObject** is a super class for all objects that move across the screen. It contains data like speed, angle and bounds.
- **Projectile** is a movable object with type and power which will damage a meteor if a collision occurs between them.
- **AbstractArmoryItem** is a super class for all armory items. It contains the shared methods for all armory items such as upgrading, setting the state and resetting its attributes when item is sold.
- **AbstractProjectileArmoryItem** subclass to AbstractArmoryItem and super class to all armory items which fires projectiles. Containes attributes projectileSize and projectileType. The execute() method of this super class has no parameters but returns a reference to a Projectile object.
- **AbstractDefenseArmoryItem** subclass to AbstractArmoryItem and super class to all defensive armory items. The execute() method in this super class takes a reference to a City object as parameter, which in the concrete subclasses should be altered.
- **AbstractEffectArmoryItem** subclass to AbstractArmoryItem and super class to all armory items which causes some kind of physical effect. The execute() method in this super class takes a reference to a list of Meteor objects as parameter. This list should be the list of currently visible meteors, which the concrete subclasses should alter.
- **ScoreHandler** is responsible for the score calculation. It keeps track of the number of shots fired and meteors hit.

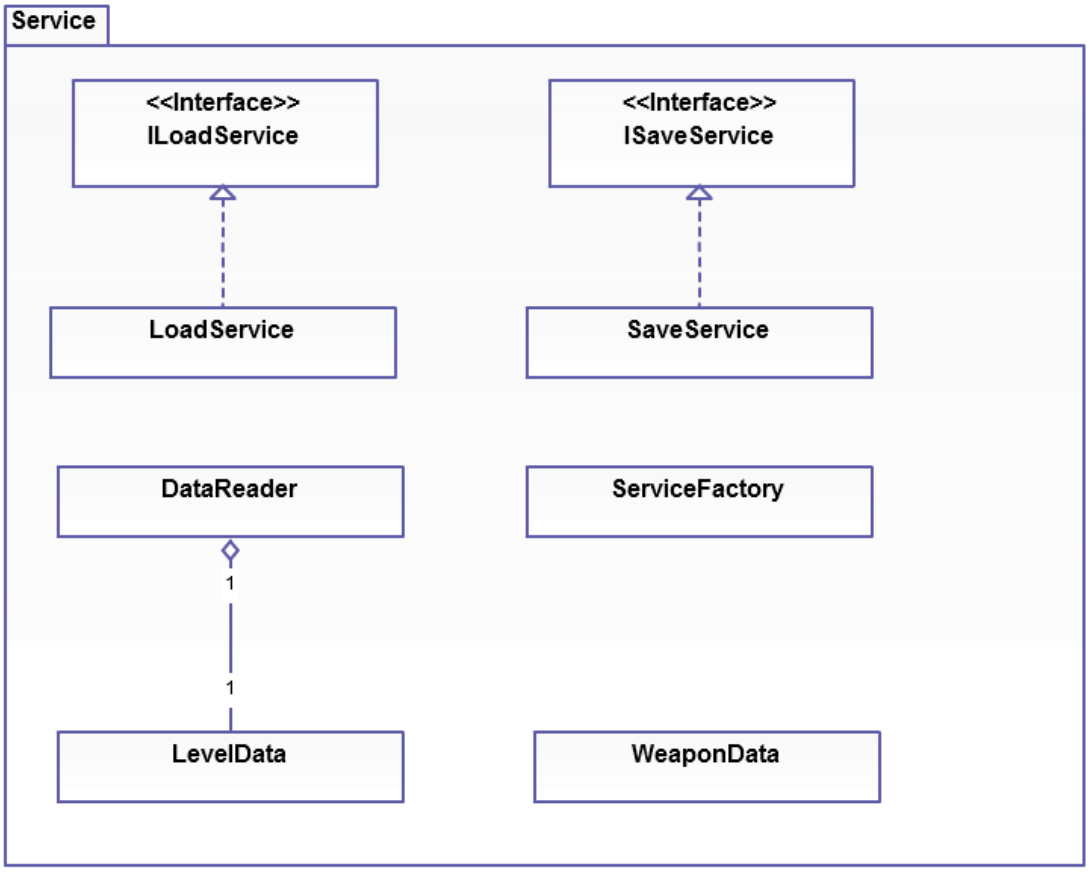

Figure 9, service package

- **ServiceFactory** provides methods for creating ILoadService and ISaveService.
- **LoadService** is the concrete implementation of ILoadService
- **SaveService** is the concrete implementation of ISaveService
- **DataReader** is a helper class for LoadService.
- **LevelData** is data holding class that DataReader uses to load level information like city and meteorshower.
- **WeaponData** is a data holding class that is used during loading and saving of the armory items states.

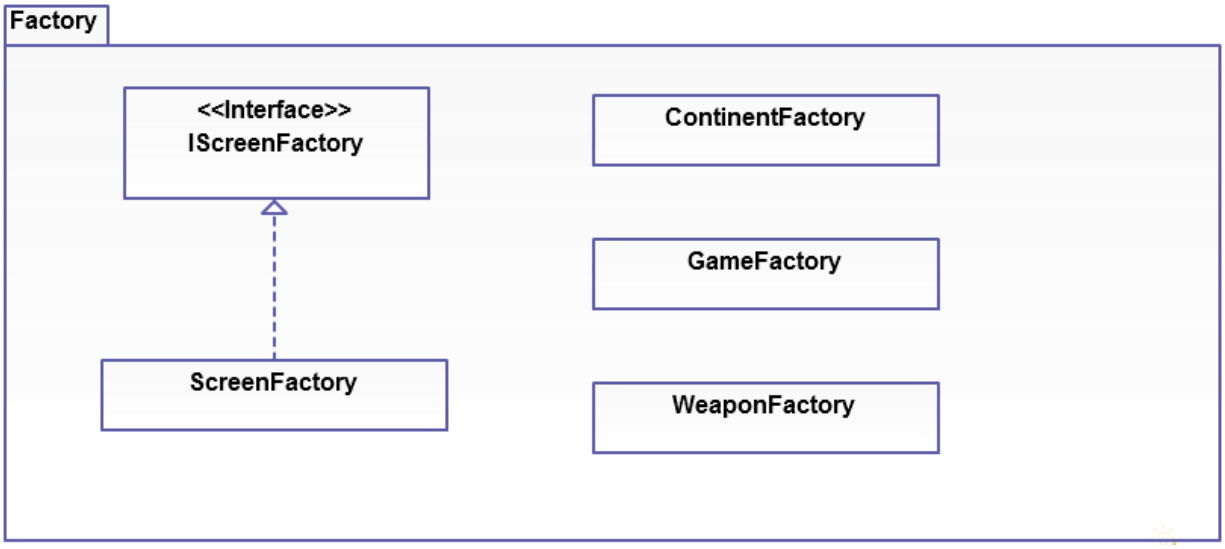

Figure 10, factory package

- **GameFactory** provides a method that creates an IScreenFactory.
- **ScreenFactory** is the concrete implementation of IScreenFactory.
- **ContinentFactory** is a helper class for ScreenFactory.
- **WeaponFactory** is a helper class for ScreenFactory.