SYSTEM DESIGN DOCUMENT FOR ‘TIS BUT A SCRATCH

TDA367

25 MAJ 2014

CHALMERS TEKNISKA HÖGSKOLA
Ivar Josefsson, Alma Ottedag, Anna Nylander, André Samuelsson

Contents

S

Version: 3

Date: 25-05-14

Revised by: Alma Ottedag & Anna Nylander

This version overrides all previous versions.

1 Introduction

1.1 Design goals

The goal is to have a loosely coupled, easily extended application. The model shouldn't be coupled to the graphical library, Slick2D, and the network library, KryoNet, at all, and the controllers should keep this coupling to a minimum. This will ensure the library is easily swapped.

1.2 Definitions, acronyms and abbreviations

The specific terms regarding the core "Tis but a scratch" game can be found in the references section.

- GUI - Graphical user interface
- Java - platform independent programming language.
- JRE - Java Runtime Environment, software needed to run Java applications
- Client - An instance of 'Tis but a scratch that receives updates through networking
- Frame - Rendering of one game state onto the screen, is done at approximately a rate of sixty per second.
- Update - One update of the internal game model
- MVC – *Model-View-Controller:* A design pattern to follow when developing applications involving a GUI, it's main goal is to separate logic from the view and make the model extractable and usable on other platforms.

2 System design

2.1 Overview

A Model-View-Controller model was used when implementing the application. All views have an association with its model counterpart, but the views do not know the controllers. The controllers in turn have one assigned model and view counterpart. Relevant information to the view is sent through method parameters. The model is not bound to the controller or view in any way.

2.2 Software decomposition

2.2.1 General

Package diagrams: For each package an UML class diagram can be found in the appendix.

The in-game application consists of mainly five packages, model, view, controller, construction and network. Packages such as the model, view and controller adapt the MVC pattern. The construction package contains all classes and interfaces for creating game maps. It will have a separate class for creating maps using the Slick2 library, such that it can be swapped with any library as needed. See the appendix for a full view of the packages.

The model contains all data and handles all fundamental game logic. The model strives not to be dependent upon libraries, so that any developer to latch on their own system to it without having to add extra dependencies. However, many classes in the package need to implement KryoSerializable to be able to send package data through the network. Even so, this should not be a major drawback when utilizing the model in other circumstances.

The controller package contains a unique controller for each entity that requires an update. This means that all controllers handles its own piece of model and its updating. This is connected to the networking. Initially a separate package was to be created for different modes of playing; as a host, locally or as a client to a host. In these packages there would be a separate hierarchy of controllers, so that a server does not have to be set up even when the player aims to play alone. As it is now, the latter is the case.

2.2.2 Decomposition into subsystems

Several subsystems will be used apart from the application's own. Creating maps will be done through a separate program called Tiled Map Editor. Graphics will be rendered partly using the graphics library Slick2d but to some degree also Swing. For networking the KryoNet library will be used. SimpleXML is used to load NPC and Weapon stats.

The system also utilizes more complex structures internally. The non-playable characters all have a movement pattern associated with them which is defined using the application's own plugin system. Reflections are used to load the compiled Java class files for these plugins. For the JUnit tests of the model, Guice is also used to create mock-objects.

It is vital to note that the application is not only a game application, but also a functional developing tool for creating other games. The application supports reading of any map without a transparent background. It is also flexible in terms of defining characters, which can be done directly by adding new XML-files to the given folder. In turn any weapon can be created analogically. The system further supports creating and connecting several plugins for movement patterns, which the developed can use to create more complex movement algorithms and behaviors. Focus has been put on making the application useful for developers, therefore more complex in-game actions have not been implemented.

2.2.3 Layering

See the UML-diagrams in the appendix.

2.2.4 Dependency analysis

Dependencies are as shown in *Figure 1*. There are no circular dependencies as of *the latest system iteration*.

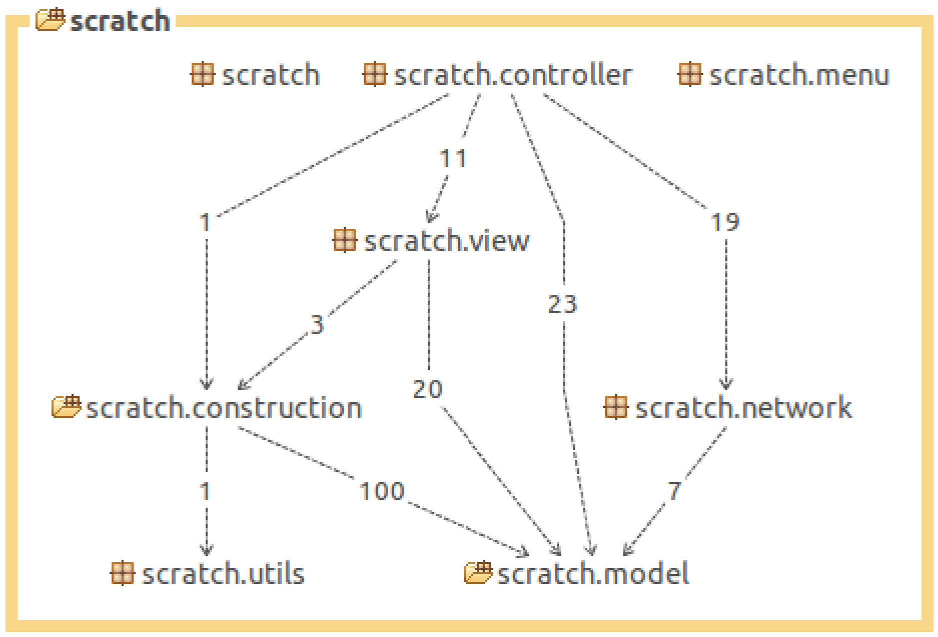

Figure 1 : A dependency diagram by STAN

2.3 Concurrency issues

The game will have to use multiple threads to accommodate networking, the server will be using three main threads. One to receive data, one to send data and one to do calculations with it. There is a strong potential for concurrency issues, this will need to be handled on a class level through use of the reserved keyword “synchronized”. The potential for issues are spread out among both client and server and we will need to be rigorous in our coding to ensure that concurrency conflicts will not occur.

2.4 Persistent data management

The application does not save any in-game data to be loaded later. The game will however load NPCs and weapons from xml files and AI logic from compiled .class files through reflection.

2.5 Access control and security

NA

2.6 Boundary conditions

NA - Game will be launched and terminated like a normal application

3 References

APPENDIX

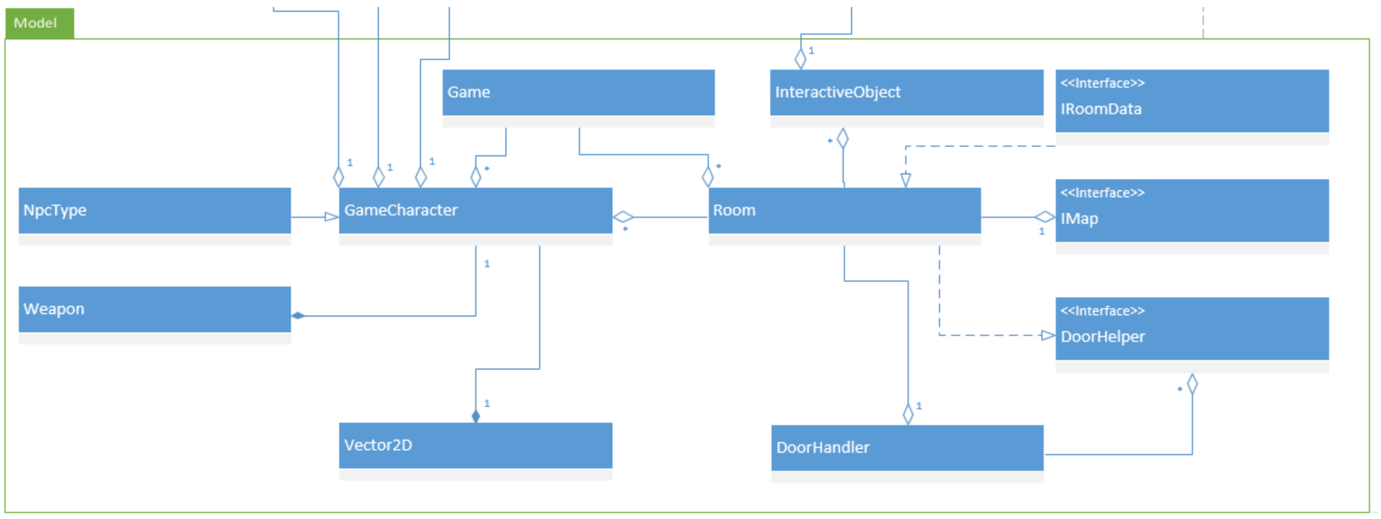

Figure 2 UML overview of the model package

Description of Model package:

- CharacterChangeListener, interface implemented by Room and contained by GameCharacter. Used to report actions that GameCharacters are able to do.
- DoorHandler is reported to when a door is interacted with. It quickly finds the doors connections and places the interacting character there.
- DoorHelper, interace implemented by room and used by DoorHandler to avoid circular dependency.
- Game contains all rooms and all players. It is also used to set the map which is to be used in the game.
- GameCharacter is simply a character which holds alot of information concerning the player.
- IMap implemented by a Slick2D class and used by room. In charge of collision with walls. Also used to get description such as width and height of the map currently in use.
- INPCMove, interface used by NpcType to get its input from. Implemented by plugins so that the behavior of Npc can be changed easily.
- InteractiveObject is a general description of an object which can be interacted with in some way with the player.

- IRoomData, interface implemented by room and used by the INpcMove so that plugins have a way of getting information from room to calculate how to act.
- MoveDirection, an Enumerator to describe the eight different move directions which can be interpreted by the arrow keys.
- NpcType extention of GameCharacter. It mainly changes how input is announced from the character.
- Room has collections of GameCharacters and InteractiveObjects. Room handles their cogency with each other and the map.
- Vector2D is used both as GameCharacters way of announcing their next move input, also used as a position vector.
- Weapon used by GameCharacter. Describes a weapon with stats such as rate of attack and damage.

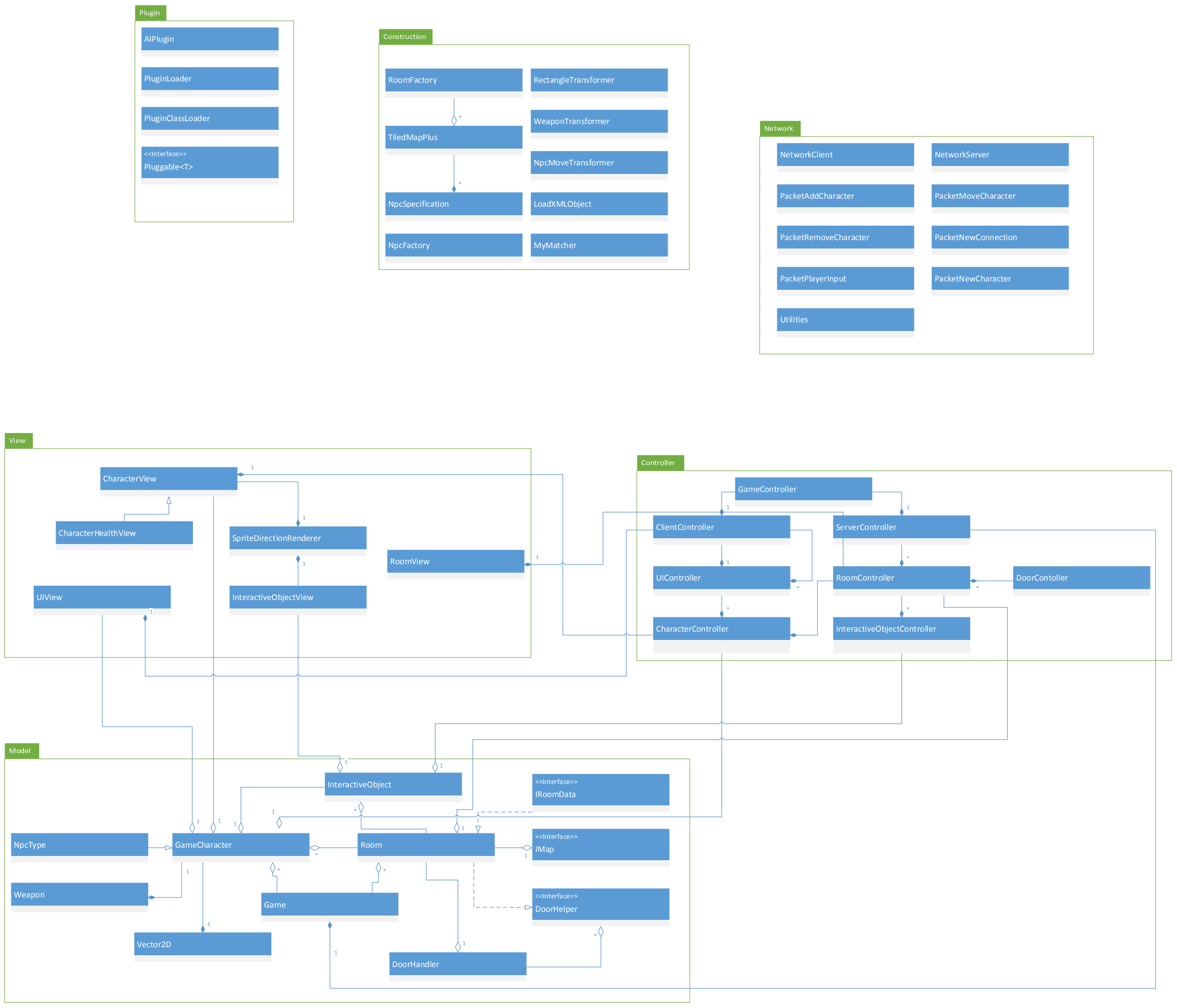

Figure 3: Domain model / Basic UML diagram

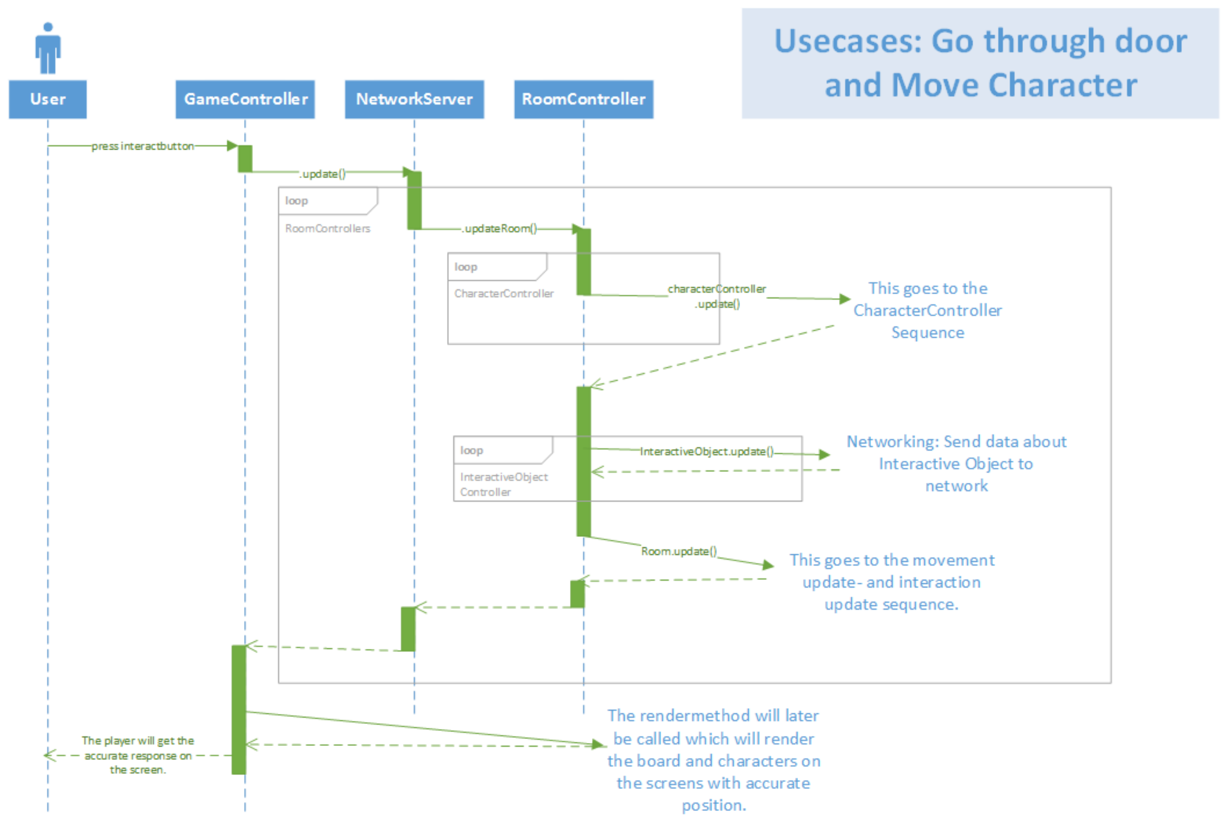
Usecases: Go through door and Move Character
User
GameController
NetworkServer
RoomController
press interactbutton
.update()
loop
RoomControllers
.updateRoom()
loop
CharacterController
characterController .update()
This goes to the CharacterController Sequence
loop
InteractiveObject Controller
InteractiveObject.update()
Networking: Send data about Interactive Object to network
Room.update()
This goes to the movement update- and interaction update sequence.
The rendermethod will later be called which will render the board and characters on the screens with accurate position.
The player will get the accurate response on the screen.

Figure 4 : Sequence diagram start for Use cases: GoThroughDoor and MoveCharacter

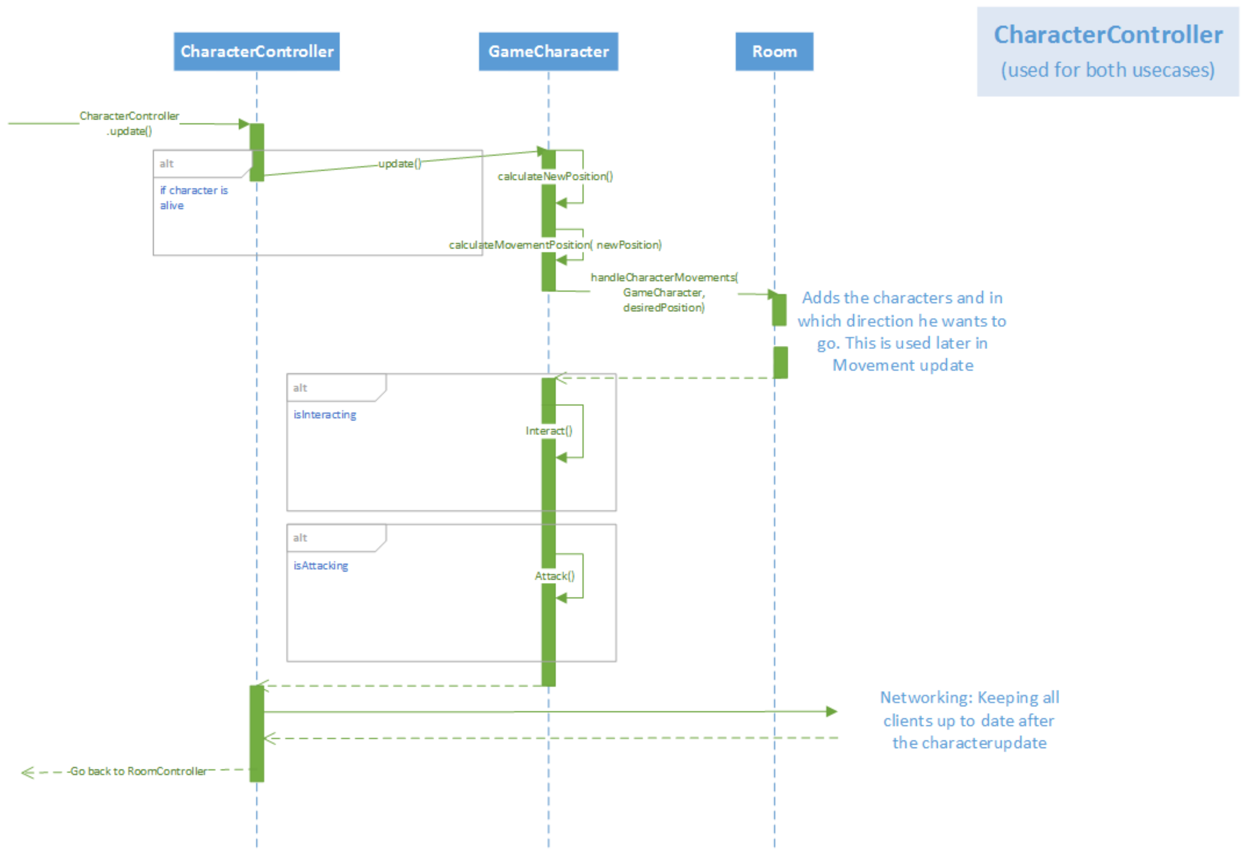
CharacterController
GameCharacter
Room
CharacterController
(used for both usecases)
CharacterController .update()
alt
if character is alive
update()
calculateNewPosition()
calculateMovementPosition(newPosition)
handleCharacterMovements(GameCharacter, desiredPosition)
Adds the characters and in which direction he wants to go. This is used later in Movement update
alt
isInteracting
Interact()
alt
isAttacking
Attack()
Networking: Keeping all clients up to date after the characterupdate
Go back to RoomController

Figure 5 : Sequence diagram part 2

Movement update

(usecase: Move Character)

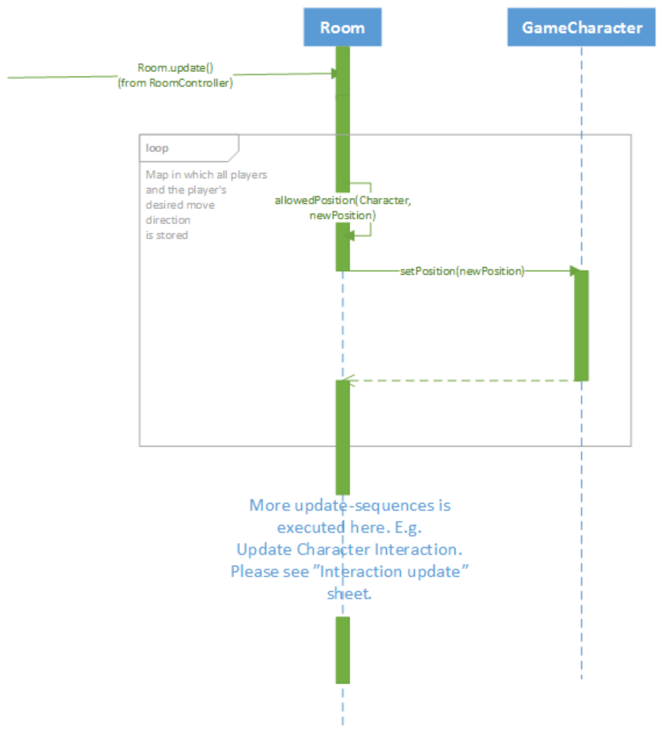
Room
GameCharacter
Room.update()
(from RoomController)
loop
Map in which all players and the player's desired move direction is stored
allowedPosition(Character, newPosition)
setPosition(newPosition)
More update-sequences is executed here. E.g. Update Character Interaction. Please see "Interaction update" sheet.

Figure 6 : Sequence diagram part 3

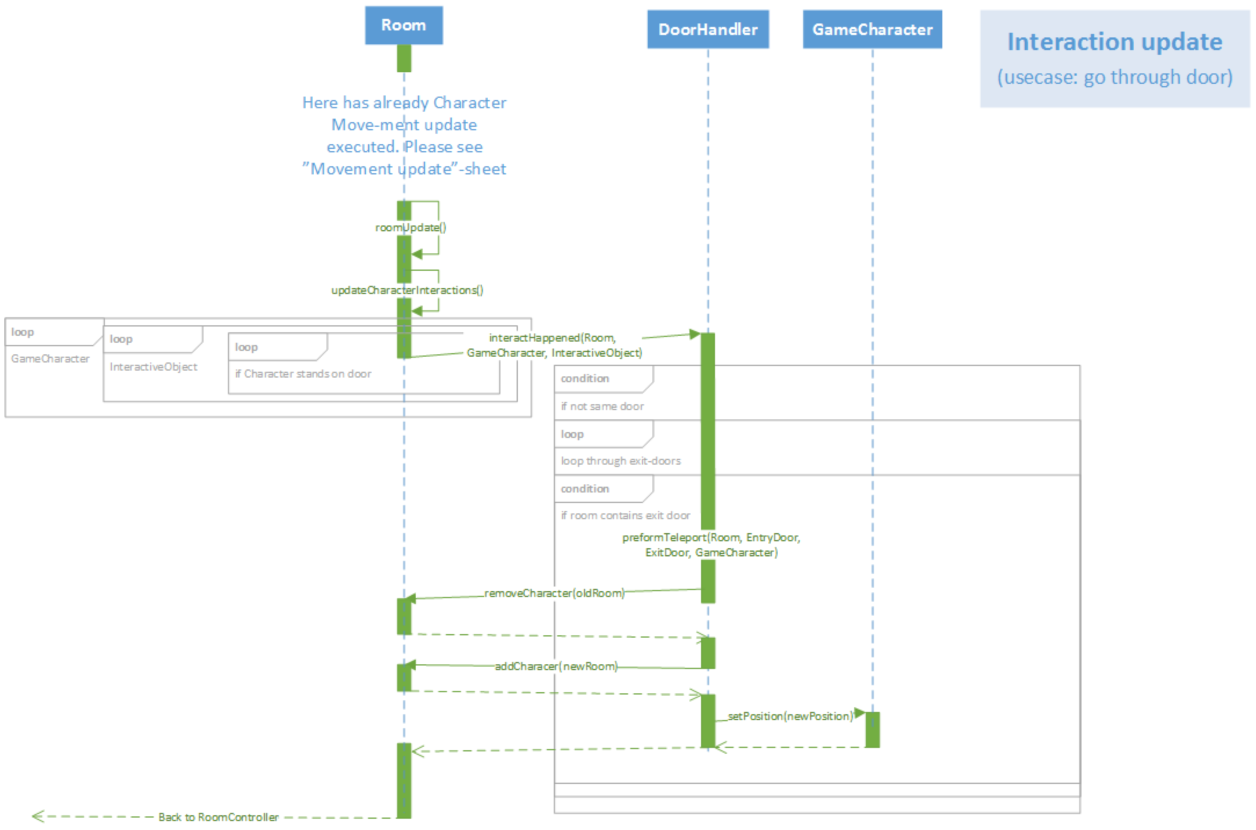
Room
DoorHandler
GameCharacter
Interaction update
(usecase: go through door)
Here has already Character Move-ment update executed. Please see "Movement update"-sheet
roomUpdate()
updateCharacterInteractions()
loop
GameCharacter
loop
InteractiveObject
loop
if Character stands on door
interactHappened(Room, GameCharacter, InteractiveObject)
condition
if not same door
loop
loop through exit-doors
condition
if room contains exit door
preformTeleport(Room, EntryDoor, ExitDoor, GameCharacter)
removeCharacter(oldRoom)
addCharacer(newRoom)
setPosition(newPosition)
Back to RoomController

Figure 7 : Sequence diagram part 4