TECHNICAL UNIVERSITY OF DENMARK

02291 SYSTEM INTEGRATION

MUD Game

GROUP 26

Authors:	*Student IDs:*
Cristina AILOAEI	s213804
Dinesh KUMAR KHADKA	s220073
Eleni GAITANIOU	s212361
Johannes VAHT STEINITZ	s205534
Nikolaos KARAVASILIS	s213685
Roza HASSO	s213558
Tamas PAULIK	s212569

Saturday 28th October, 2023

Contents

List of Figures

List of Tables

1 Introduction

The MUD Game is an online multi-user dungeon game with limited functions, inclusive of communication between the players, collection and trade of items.

In order to play the game, the user must own a mobile phone and create or have an account on the game server. Upon logging into the game, all players are redirected to room zero in the first level, where the game begins. There is one start room and one special room for each level. When a player finds the special room of its level then can level up (advance to the next room).

Each of the players can have an inventory with space up to five objects which can be either collected from the rooms or be traded between the players, there is also the option of discarding these objects if not needed.

The player who finds the special room of the last level room first wins the game.

This report analyzes the functional and non-functional requirements of the MUD Game as well as its domain analysis including diagrams, detailed use cases and acceptance fit tables.

2 Requirements

2.1 Domain Analysis

In order to understand the scope and avoid ambiguities of the domain we present the following definitions from the given text.

2.1.1 Glossary

- Multi user dungeon (MUD) game: A dungeon game with availability for multiple users
- Dungeon game: A game which takes place in a dungeon like environment
- Game Server: A server which hosts the game that player can connect to
- Mobile Phone: A device used by users to play and connect to the MUD game
- User: A real-world actor which uses the game through his/hers mobile device
- Player: A user instance inside the game
- Room: An interactive space inside the dungeon that mimics a room
- Level: An area containing of multiple interconnected rooms
- Special room: A room which advances to player to a successor level
- Start room: The first room of a level
- Advance to next level: Moving the user from one level to a successor level
- Meet player: Meet other players in a room
- Talk to players: Communicate to other players inside a room
- Objects: Items residing in rooms which can be picked up, dropped or traded with other players
- Inventory: A restricted space owned by a player that can hold up to five objects
- Pick up object: An action performed by a player which results in the user obtaining an object into its inventory
- Lay down object: An action performed by a player which results in the user removing an object from its inventory
- Trade object: An action performed by a player, which results in a trade of object(s) with other players

2.1.2 Class diagram

The players of the MUD game have to go through levels, each of which consisting of a start room, a special room and possibly other rooms. The relationship between rooms is represented as inheritance, having Start Room and Special Room inherit from the simple Room. Rooms may contain objects and the players may store up to five objects in their inventory. They can pick them up, lay them down and trade them with other players. Therefore, the reflexive association of the Player class. Besides trading, a player can also talk to the other players he meets, while he moves from a room to another. Moving to the special room of a level means finding it and automatically advancing to the next level. Finally, for all of these to be possible, the phone requests to join the game server. [2]

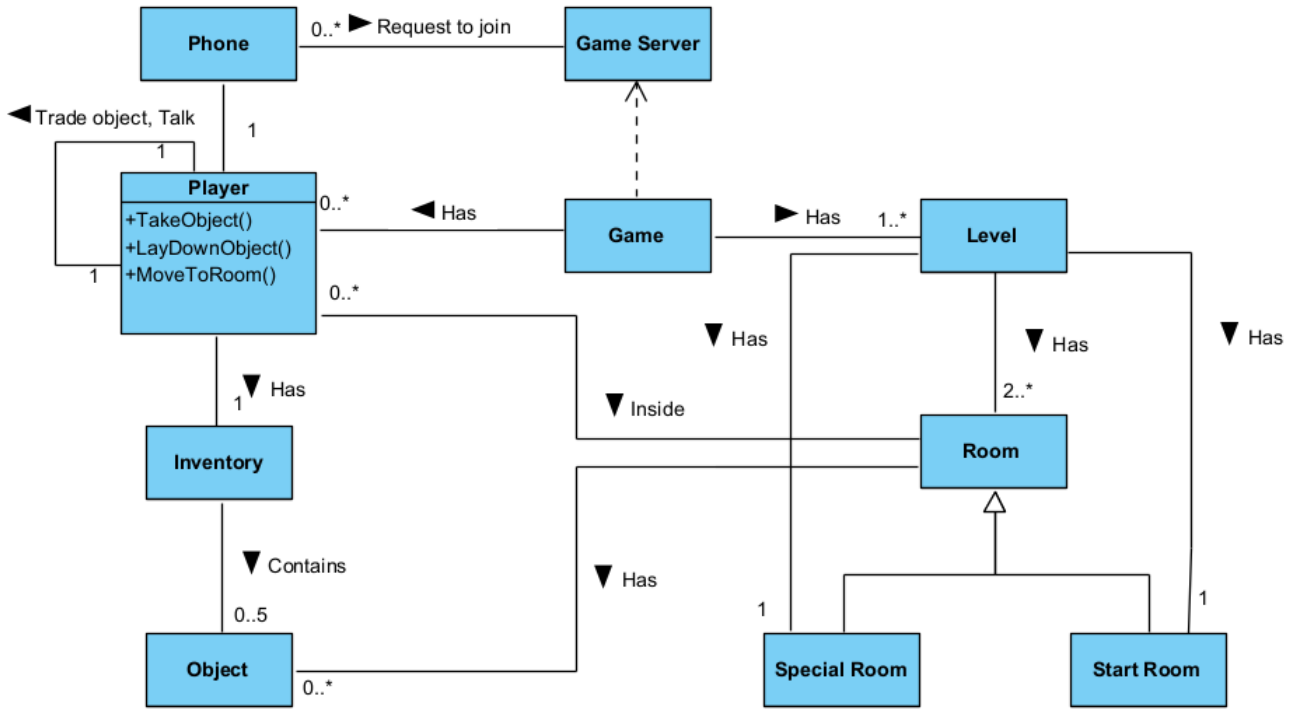


Figure 1: MUD Game Class Diagram

2.2 Functional Requirements

- The MUD Game Server must be able to receive and handle join requests from mobile phones.
- The player must be able to move between the rooms of the game in each level.
- The player must be automatically advanced to the next level after he has found the special room.
- The player must be able to take objects from rooms.

- The player must be able to lay down objects in rooms.
- The player must be able to trade objects with other players.
- The player must be able to talk to other players.

2.2.1 Use Case diagrams

Two actors have been identified to interact with the MUD Game System, namely Player and Phone.

Phone is an actor perceived as an outside system which sends a join request to the Game Server. Once the server accepts the incoming request, the Player actor is allowed to start playing the game. Figure 2 shows in greater detail the interactions of these two actors with the system.

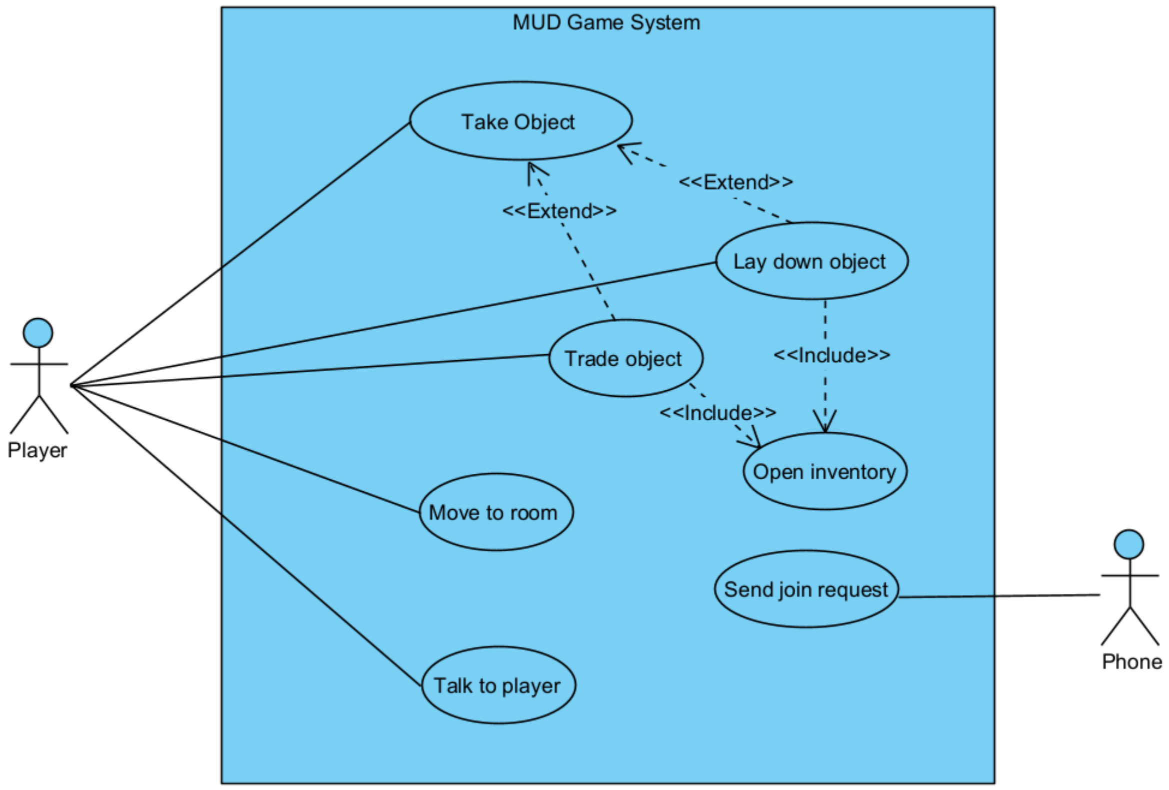


Figure 2: MUD Game Use Case Diagram

The Player actor can execute the following actions: *Move to a room* inside a level and *Take objects* found in these rooms. While visiting rooms, the Player may meet other players, *Talk to them* and *Trade objects* with them. Additionally, the Player has the possibility to *Lay down objects* in rooms. Both the "Trade object" and "Lay down object" use cases extend the "Take object" use case and include the "Open inventory" use case, since these objects come from the inventory and have to be first taken and stored before being laid down or further traded.

2.2.2 Detailed Use Cases

Table 1 represents a detailed use case for the *Move to room* use case.

Use case name	Move to room
Description	The player moves between rooms in each level
Actors	Player
Preconditions	The player is playing the MUD game and is currently in a room
Main Scenarios	1. The player chooses a room to go to 2. The player moves to the chosen room
Postconditions	The player is moved to the chosen connected room in the same level
Assumptions	Once the player moves out from the start room in each level, it is not possible to get back to it i.e. the arrow from the start room in each level is unidirectional

Table 1: *Move to room* detailed use case

Table 2 is the detailed use case for the *Lay down object* use case.

Use case name	Lay down object
Description	The player drops an object in the current room
Actors	Player
Preconditions	The player has been accepted by the game server to join MUD game
Main Scenarios	1. The player opens its inventory. 2. The player repeats: 2.1 Choose an object from the inventory. 2.2 Drop the object.
Alternative paths	1.a The player's inventory is empty. No object to choose and drop.
Postconditions	The object is dropped into the room.
Assumptions	All objects from the inventory can be dropped.

Table 2: *Lay down object* detailed use case

2.3 Non-Functional Requirements

- The game must be a multi user based system.
- The game must be played on a mobile phone environment.
- The user must create an account in order to join the game.
- The player's mobile phone must have enough memory space for the game to be downloaded.
- The player's mobile phone must be connected to the Internet while playing the game.
- The game server must be available at all times.

2.4 Acceptance test Fit tables

The fit acceptance test in table 3 is based on the use case *Move to room*, while the one in table 4 is based on use case *Lay down object*. The first test uses the example tree-like map from the hand-out.[1]

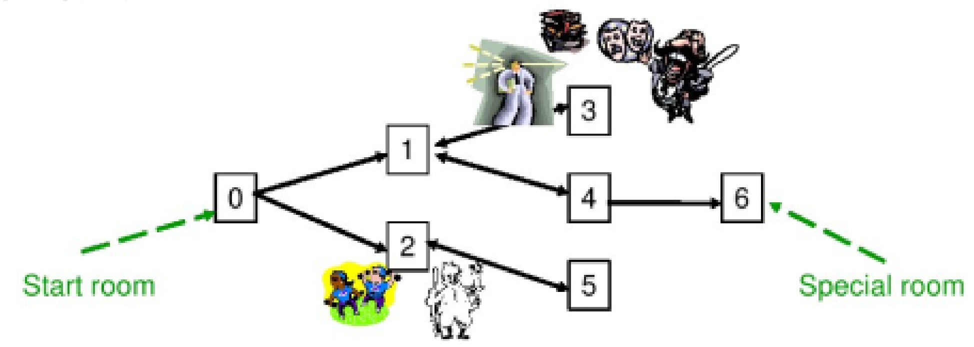


Figure 3: Example MUD tree

ActionFixture		
start	Start room	0
check	possible destination rooms	1,2
click	move to room	1
check	possible destination rooms	3,4
check	current room	1
click	move to room	4
check	possible destination rooms	6
check	current room	4
click	move to room	6
check	possible destination rooms	0
check	current room	6

Table 3: Fit Acceptance Test for Use Case *Move to room*

ActionFixture		
start	open inventory	
check	inventory is opened	True
check	inventory is not empty	True
click	select item1	
check	item1 is selected	True
click	lay down	
check	item1 is part of the room	True
check	item1 is NOT part of the inventory	True

Table 4: Fit Acceptance Test for Use Case *Lay down object*

3 Design

The design of the MUD Game builds up on the specified requirements and has as its sole focus the two detailed use cases mentioned in section 2.2.2, namely *Move to room* and *Lay down object*.

3.1 Component Design

The MUD Game System comprises two components: the *Phone* component and the *GameServer* component, represented in figure 4. These two reusable software units are pluggable through their ports by the means of a connector. It is also worth mentioning that the multiplicity of the relationship between the two components is one-to-many, as we expect multiple instances of the *Phone* component.

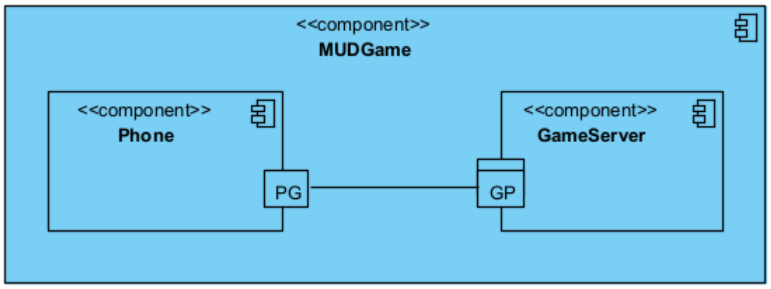
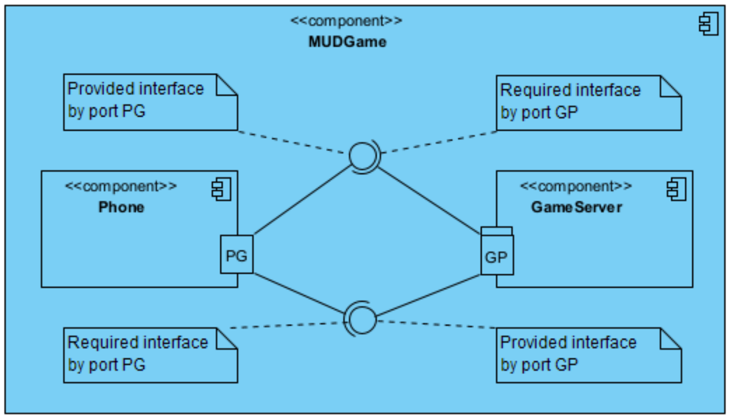


Figure 4: Component Diagram

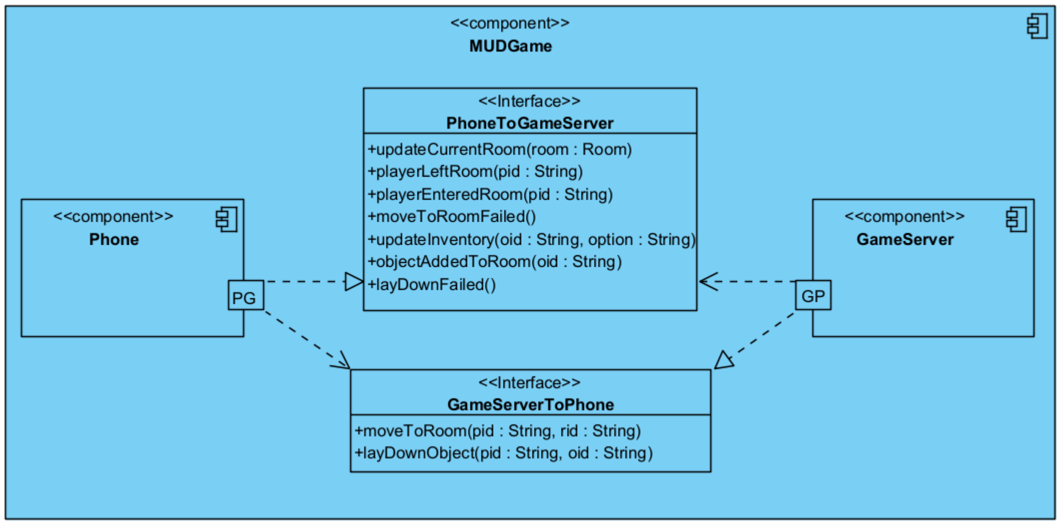
Next, we present a first look into the provided and required interfaces owned by the ports *PG* and *GP*. Since the communication between the two components must be back-and-forth, both the *Phone* and the *GameServer* components provide and require a set of functions. Therefore, the following provided and required interfaces exist.



<<component>>
MUDGame
Provided interface by port PG
Required interface by port GP
<<component>>
Phone
PG
<<component>>
GameServer
GP
Required interface by port PG
Provided interface by port GP

Figure 5: Component Diagram - showing interfaces

More specifically, they are defined as following:



<<component>>
MUDGame
<<Interface>>
PhoneToGameServer
+updateCurrentRoom(room : Room)
+playerLeftRoom(pid : String)
+playerEnteredRoom(pid : String)
+moveToRoomFailed()
+updateInventory(oid : String, option : String)
+objectAddedToRoom(oid : String)
+layDownFailed()
<<component>>
Phone
PG
<<component>>
GameServer
GP
<<Interface>>
GameServerToPhone
+moveToRoom(pid : String, rid : String)
+layDownObject(pid : String, oid : String)

Figure 6: Component Diagram - showing interfaces with methods

The *GameServer* component provides the *GameServerToPhone* interface, with the functions "move-ToRoom" and "layDownObject". At the same time, it requires the *PhoneToGameServer* interface, whose functions it uses to update the phone.

3.2 Detailed Class Design incl. OCL constraints

A holistic view of the detailed class design is presented in figure 7, consisting of three main packages: *Phone* (top, left), *GameServer* (top, right) and, lastly, *Library* (bottom, center). The first two packages will be explained in greater detail in the following sections. However, it is important to highlight the relationships between these packages, specifically the relationship Phone-Library and GameServer-Library. This is an "import" relationship, as both *Phone* and *GameServer* make use of the same group of classes. Thus, a shared class library has been created, namely the package *Library*. This package can be inspected in figure 8.

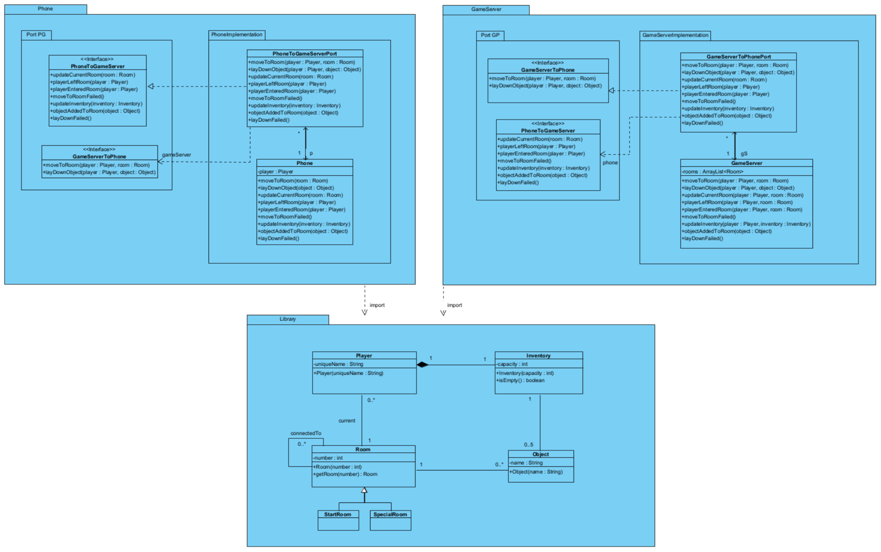


Figure 7: Detailed class diagram - holistic view

The *Library* package contains only classes related to the two detailed use cases, specifically *MoveToRoom* and *LayDownObject*. A *Player* can be identified by its id and can hold in its *Inventory* up to 5 objects. An unlimited number of objects can be found in *Rooms*, let them be ordinary, *Start rooms* or *Special rooms*. Each room is connected to possibly many other rooms. The relationship between *Room* and *Player* is bidirectional, since a room contains many players and a player has a current room. It is possible to remove objects from the inventory, as well as add objects to rooms.

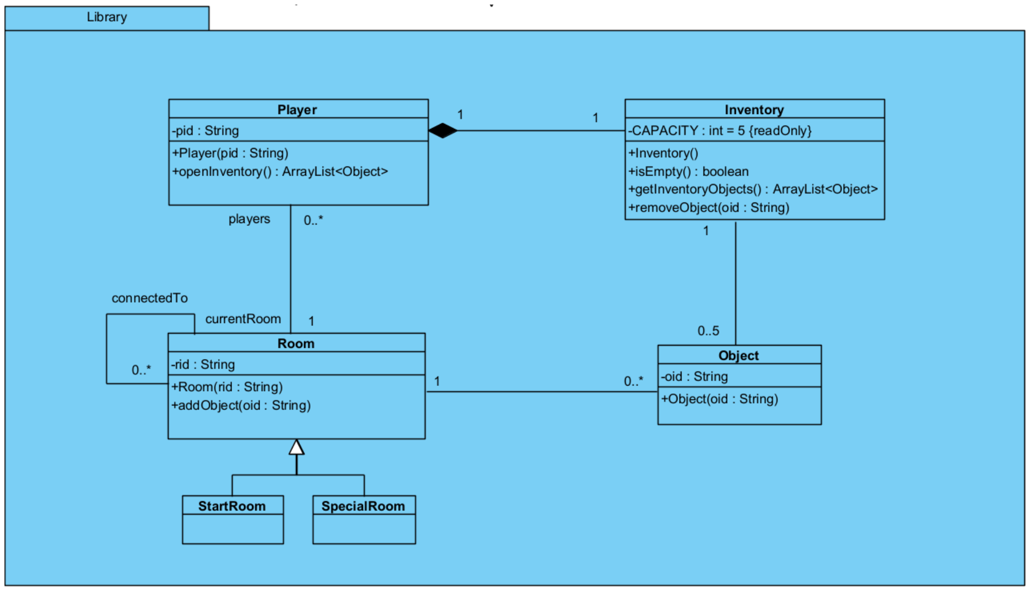
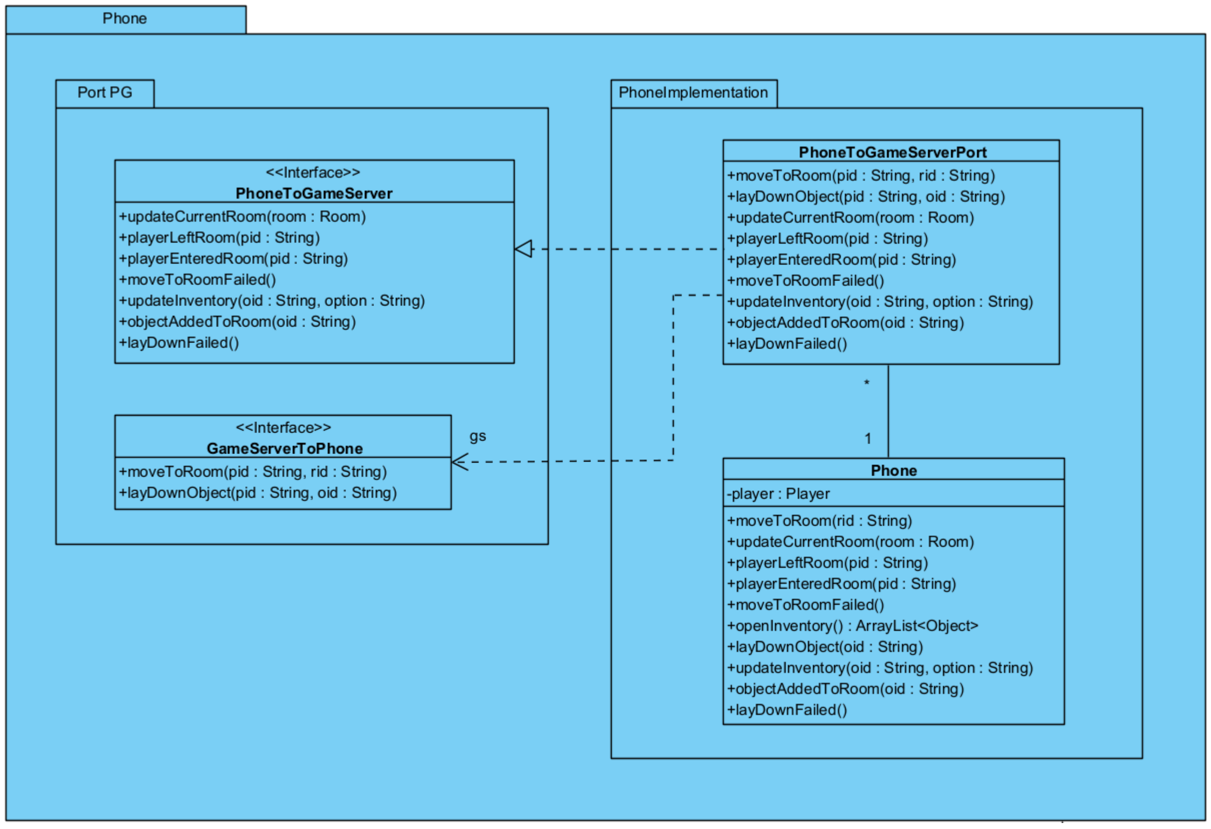


Figure 8: The *Library* Package

3.2.1 Game Server

The *GameServer* package comprises 2 other packages, namely *Port GP* and *GameServerImplemenation*. The latter contains the GameServer class, which holds a list of the rooms and players in the game. That is, the GameServer is responsible for maintaining a whole view of the system, since a room knows about its connected rooms, has objects, as well as players, who ultimately keep objects in their inventory. Therefore, we can conclude that the GameServer holds the entire state of the game.

3.2.2 Phone



Phone
Port PG
<<Interface>>
PhoneToGameServer
+updateCurrentRoom(room : Room)
+playerLeftRoom(pid : String)
+playerEnteredRoom(pid : String)
+moveToRoomFailed()
+updateInventory(oid : String, option : String)
+objectAddedToRoom(oid : String)
+layDownFailed()
<<Interface>>
GameServerToPhone
+moveToRoom(pid : String, rid : String)
+layDownObject(pid : String, oid : String)
gs
PhoneImplementation
PhoneToGameServerPort
+moveToRoom(pid : String, rid : String)
+layDownObject(pid : String, oid : String)
+updateCurrentRoom(room : Room)
+playerLeftRoom(pid : String)
+playerEnteredRoom(pid : String)
+moveToRoomFailed()
+updateInventory(oid : String, option : String)
+objectAddedToRoom(oid : String)
+layDownFailed()
*
1
Phone
-player : Player
+moveToRoom(rid : String)
+updateCurrentRoom(room : Room)
+playerLeftRoom(pid : String)
+playerEnteredRoom(pid : String)
+moveToRoomFailed()
+openInventory() : ArrayList<Object>
+layDownObject(oid : String)
+updateInventory(oid : String, option : String)
+objectAddedToRoom(oid : String)
+layDownFailed()

Figure 9: The *Phone* Package

The *Phone* package also consists of 2 packages, these being *Port PG* and *PhoneImplementation*. The second named package contains the Phone class, which holds a single object, namely a Player. From this perspective, it means that the *Phone* component doesn't have a holistic view on the system, as the *GameServer* component does, but it only knows about a single player, its own. Correlating with the *Library* Package in figure 8, the Phone can only know about its single player, its inventory and objects, but also the player's current room, which ultimately holds a list of all the players in the room, objects found on the floor, as well as other connected rooms. To summarize, the Phone can only know about one room at a time.

3.2.3 OCL Constraints for the *Lay Down Object* Action

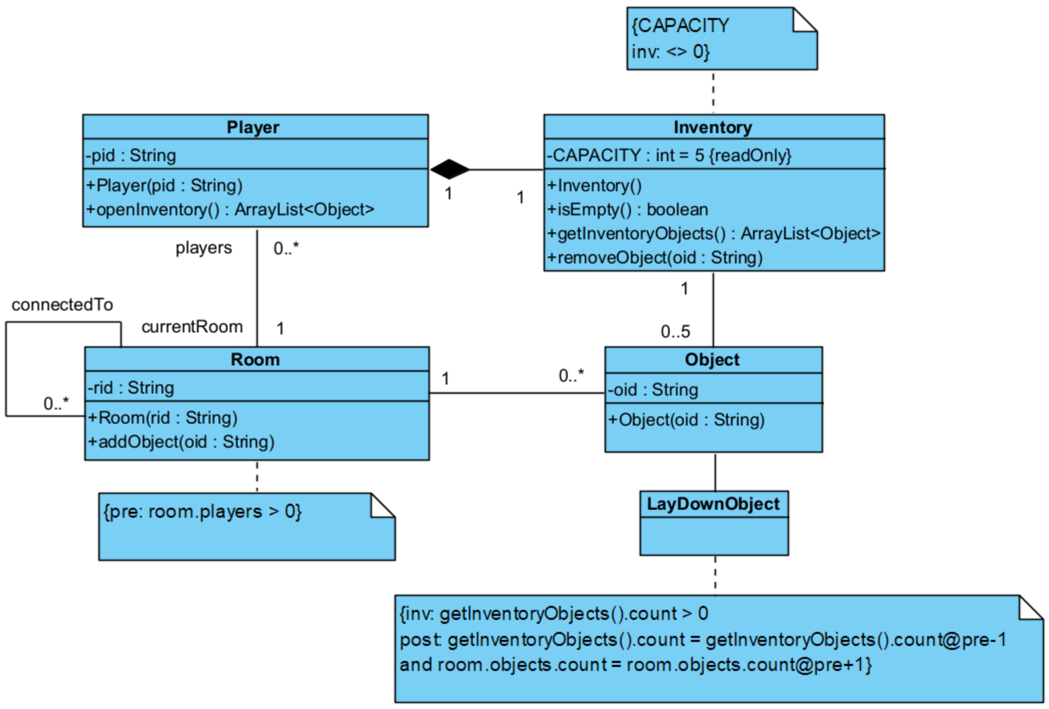

{CAPACITY
inv: <> 0}
Player
-pid : String
+Player(pid : String)
+openInventory() : ArrayList<Object>
Inventory
-CAPACITY : int = 5 {readOnly}
+Inventory()
+isEmpty() : boolean
+getInventoryObjects() : ArrayList<Object>
+removeObject(oid : String)
1
1
players
0..*
connectedTo
currentRoom
1
1
0..5
Room
-rid : String
+Room(rid : String)
+addObject(oid : String)
0..*
1
0..*
Object
-oid : String
+Object(oid : String)
{pre: room.players > 0}
LayDownObject
{inv: getInventoryObjects().count > 0
post: getInventoryObjects().count = getInventoryObjects().count@pre-1
and room.objects.count = room.objects.count@pre+1}

Figure 10: OCL Object

A player must meet certain conditions before laying down an object. Firstly, the player has successfully moved to an available room, so there is at least one player in the specific room where the object will be laid down. Secondly, the player's inventory cannot be empty; at least one object must be in it. Lastly, the post-conditions after the player successfully laid down the object will be that the objects in the inventory will be decremented by one and the ones in the room will be incremented by one.

3.3 Behaviour Design

In order to lay down an object, the user starts by opening its inventory and then proceeds by choosing one object in the inventory. This is captured in the method *openInventory()*, which fetches all the player's objects. Then, the method *layDownObject(oid)* with the id of the chosen object as an argument is sent to the Phone, which in turn calls the GameServer to execute the action. The GameServer will remove the object from the inventory of the player and add it to its current room on the server, then call the method *updateInventory(oid, "remove")* on the Phone, which will update its player's inventory by removing the object corresponding to oid. Furthermore, the GameServer notifies all players in the current room of the player (him included) who initiated the lay down that a new object is now in the room, which consequently leads to an update of the current room of each of these players by adding the object.

3.3.1 Phone Component

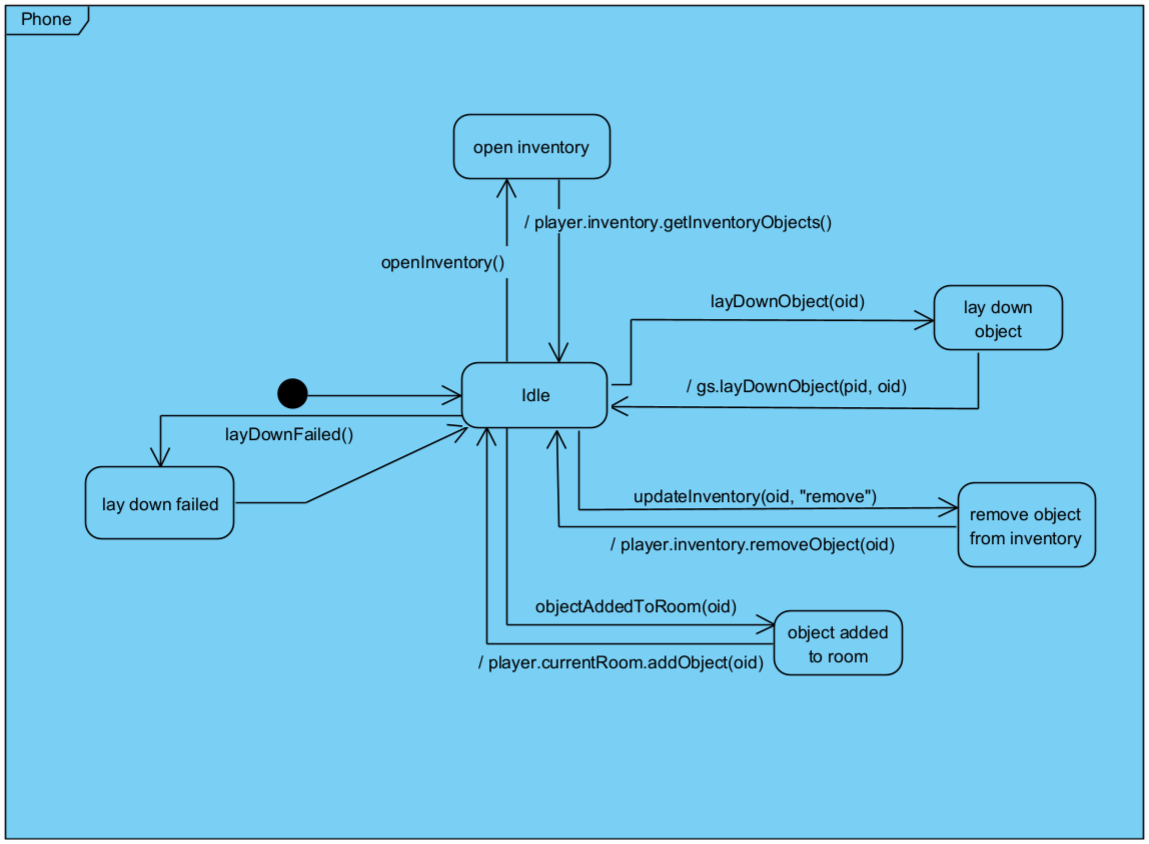

Phone
open inventory
/ player.inventory.getInventoryObjects()
openInventory()
layDownObject(oid)
lay down object
/ gs.layDownObject(pid, oid)
Idle
layDownFailed()
lay down failed
updateInventory(oid, "remove")
remove object from inventory
/ player.inventory.removeObject(oid)
objectAddedToRoom(oid)
object added to room
/ player.currentRoom.addObject(oid)

Figure 11: LSM class Phone

The behaviour of class Phone presented in figure 11 can also be expressed through the following OCL constraints:

context: Phone::openInventory() : ArrayList<Object>
body: player.inventory.objects->select()

context: Phone:: layDownObject(oid : String)
pre: oid <> null
post: gs^layDownObject(self, oid)

context: Phone:: updateInventory(oid : String, option : String)
pre: oid <> null and option == "remove"
post: player.inventory.objects = player.inventory.objects@pre->excluding(oid)

context: Phone:: objectAddedToRoom(oid : String)
pre: oid <> null
post: player.currentRoom.objects = player.currentRoom.objects@pre->including(oid)

context: Phone:: layDownFailed()
post: true

3.3.2 GameServer Component

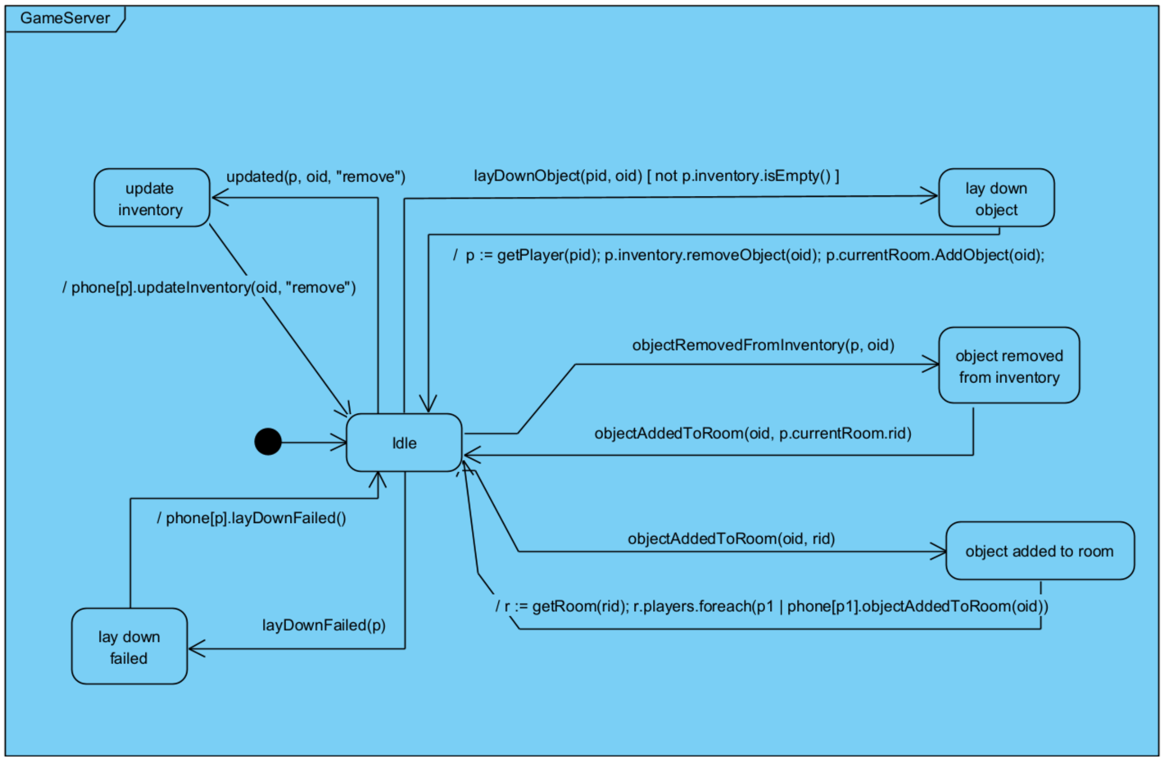

GameServer
update inventory
updated(p, oid, "remove")
layDownObject(pid, oid) [not p.inventory.isEmpty()]
lay down object
/ p := getPlayer(pid); p.inventory.removeObject(oid); p.currentRoom.AddObject(oid);
/ phone[p].updateInventory(oid, "remove")
objectRemovedFromInventory(p, oid)
object removed from inventory
Idle
objectAddedToRoom(oid, p.currentRoom.rid)
/ phone[p].layDownFailed()
objectAddedToRoom(oid, rid)
object added to room
/ r := getRoom(rid); r.players.foreach(p1 | phone[p1].objectAddedToRoom(oid))
layDownFailed(p)
lay down failed

Figure 12: LSM class GameServer

The behaviour of class GameServer presented in figure 12 can also be expressed through the following OCL constraints:

context GameServer::layDownObject(pid: String, oid: String)
pre: pid <> null and oid <> null and p.inventory.objects.count > 0

post: p = getPlayer(pid); p.inventory.objects = p.inventory.objects@pre->excluding(oid); p.currentRoom.objects = p.currentRoom.objects@pre->including(oid);

context GameServer:: updated(p : Player, oid : String, option : String)
pre: p <> null and oid <> null and option == "remove"
post: phone[p]^updateInventory(oid, option)

context GameServer:: objectRemovedFromInventory(p : Player, oid : String)
pre: p <> null and oid <> null
post: -> self^objectAddedToRoom(oid, p.currentRoom.rid)

context GameServer:: objectAddedToRoom(oid : String, rid: String)
pre: oid <> null and rid <> null
post: -> r = getRoom(rid); r.players->forAll(p1| phone[p1]^objectAddedToRoom(oid))

context GameServer:: layDownFailed(p: Player)
pre: p <> null
post: phone[p]^moveFailed()

4 Validation

4.1 Use Case Realisation

4.1.1 Sequence diagram for successful *Lay Down Object*

The successful *Lay Down Object* sequence diagram, which depicts the main flow of the use case, comprises of an actor an multiple objects. The actor is denoted *Player* and represents the external user who desires to lay down an object from his inventory, marked as *inv: Inventory*, by initiating the action "openInventory()" on his phone *phone1: Phone*. Note that both calls "openInventory()" and "getInventoryObjects()" are synchronous, as the list of objects needs to be returned before any further action can take place.

Once the list is returned, the *Player* can initiate as many "layDownObject(oid)" calls as wanted, while the count of the objects in the inventory is still positive. Each of these calls is asynchronous and reaches the *gs : GameServer* lifeline, which synchronously finds the player requesting a lay down, specified by *p1 : Player*, and then asynchronously updates the state of the inventory inv1 : Inventory and current room *currentRoom : Room* of this player.

When the update is finalized on the server side, the *gs : GameServer* asks the *phone1: Phone* to update its inventory on the client side, by using the message "updateInventory(oid, "remove"). Finally, the *gs : GameServer* fetches synchronously all players from the current room of the player who initiated the lay down, and asks them to update their current room by adding the new object using the call "objectAddedToRoom(oid)". Since there might be many players inside the room, we represent them only by using one lifeline, namely *phone2 : Phone*.

See figure 13.

4.1.2 Sequence diagram for failed *lay down object*

The failed *Lay Down Object* sequence diagram represents an exceptional flow of the use case, more specifically when removing the object from the player's inventory fails on the server's side. In this situation, no actions for updating the objects in the *currentRoom : Room* are executed by the *gs : GameServer*, but instead a "layDownObjectFailed()" message is sent back to the *Player* through *phone1 : Phone*.

See figure 14.

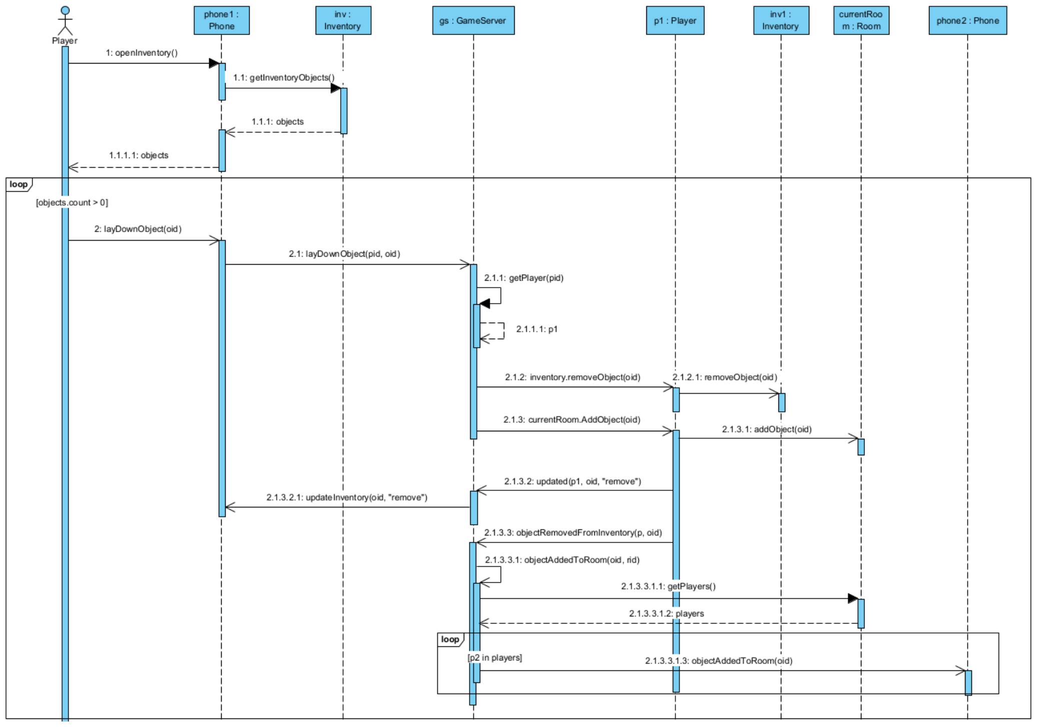

Player
phone1 : Phone
inv : Inventory
gs : GameServer
p1 : Player
inv1 : Inventory
currentRoom : Room
phone2 : Phone
1: openInventory()
1.1: getInventoryObjects()
1.1.1: objects
1.1.1.1: objects
loop
[objects.count > 0]
2: layDownObject(oid)
2.1: layDownObject(pid, oid)
2.1.1: getPlayer(pid)
2.1.1.1: p1
2.1.2: inventory.removeObject(oid)
2.1.2.1: removeObject(oid)
2.1.3: currentRoom.AddObject(oid)
2.1.3.1: addObject(oid)
2.1.3.2: updated(p1, oid, "remove")
2.1.3.2.1: updateInventory(oid, "remove")
2.1.3.3: objectRemovedFromInventory(p, oid)
2.1.3.3.1: objectAddedToRoom(oid, rid)
2.1.3.3.1.1: getPlayers()
2.1.3.3.1.2: players
loop
[p2 in players]
2.1.3.3.1.3: objectAddedToRoom(oid)

Figure 13: Sequence diagram: successful Lay Down Object

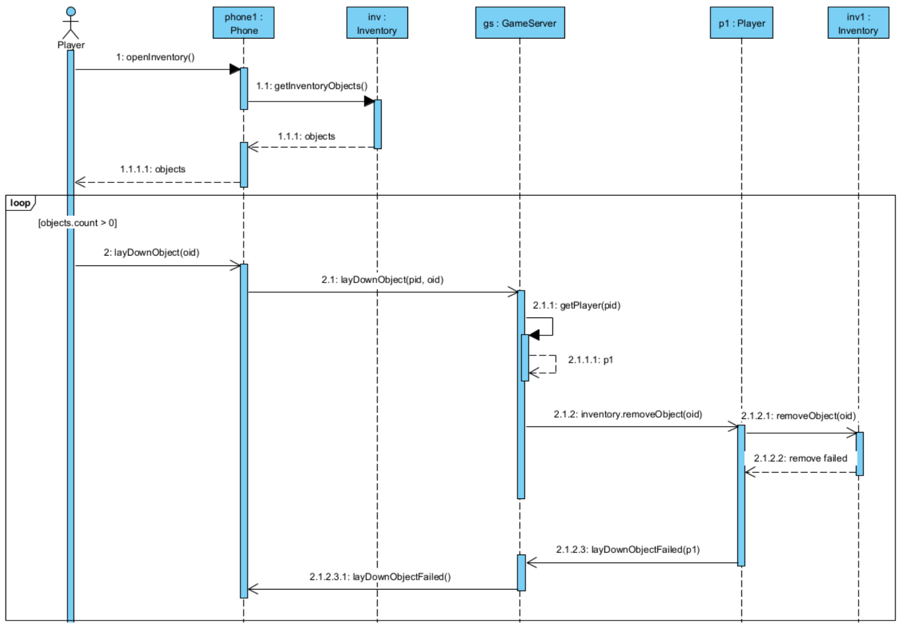


Figure 14: Sequence diagram: failed Lay Down Object

References

[1] Assoc. Prof. H. Baumeister. System Integration. mud game, February 8 2022.

[2] visual paradigm.com. Class diagram relationships. Accessed: 09/03/2022.