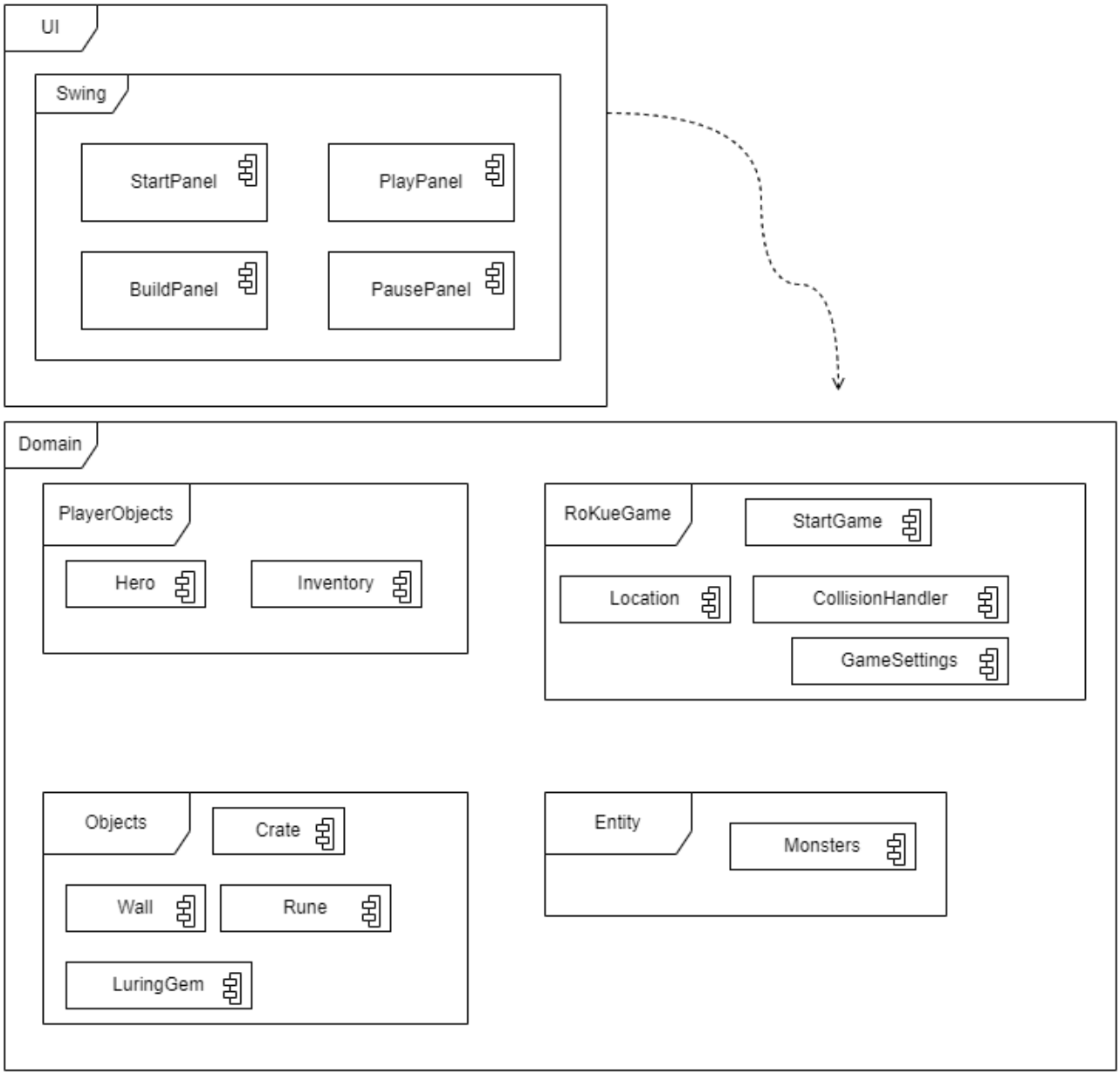
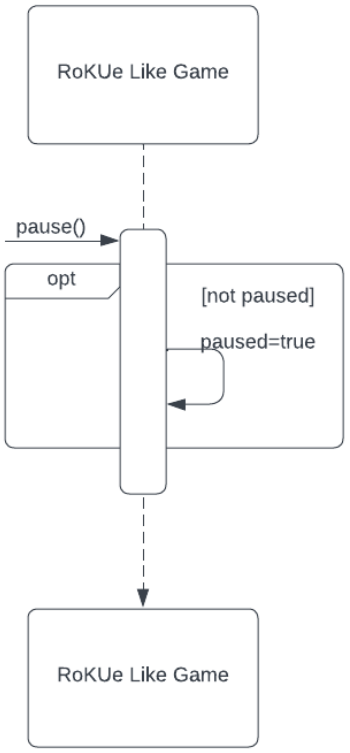
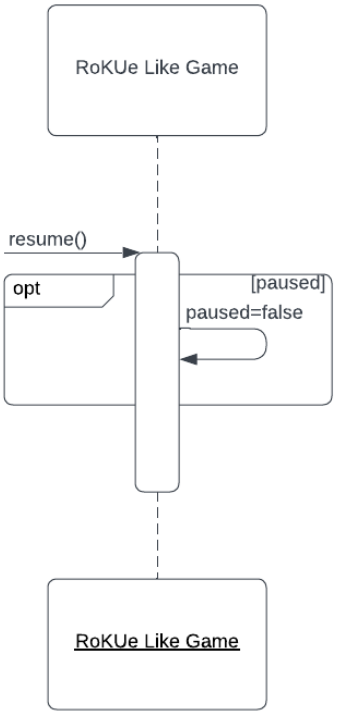
1. Logical Architecture
2. Interaction Diagrams
3. UML Class Diagram
4. Pattern Discussion

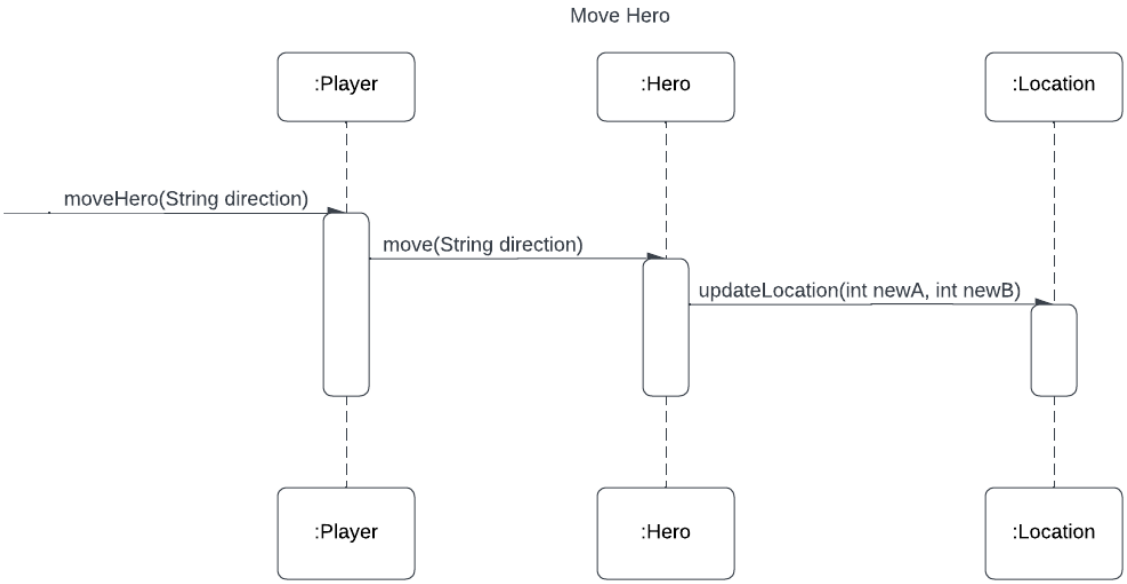
1.Logical Architecture

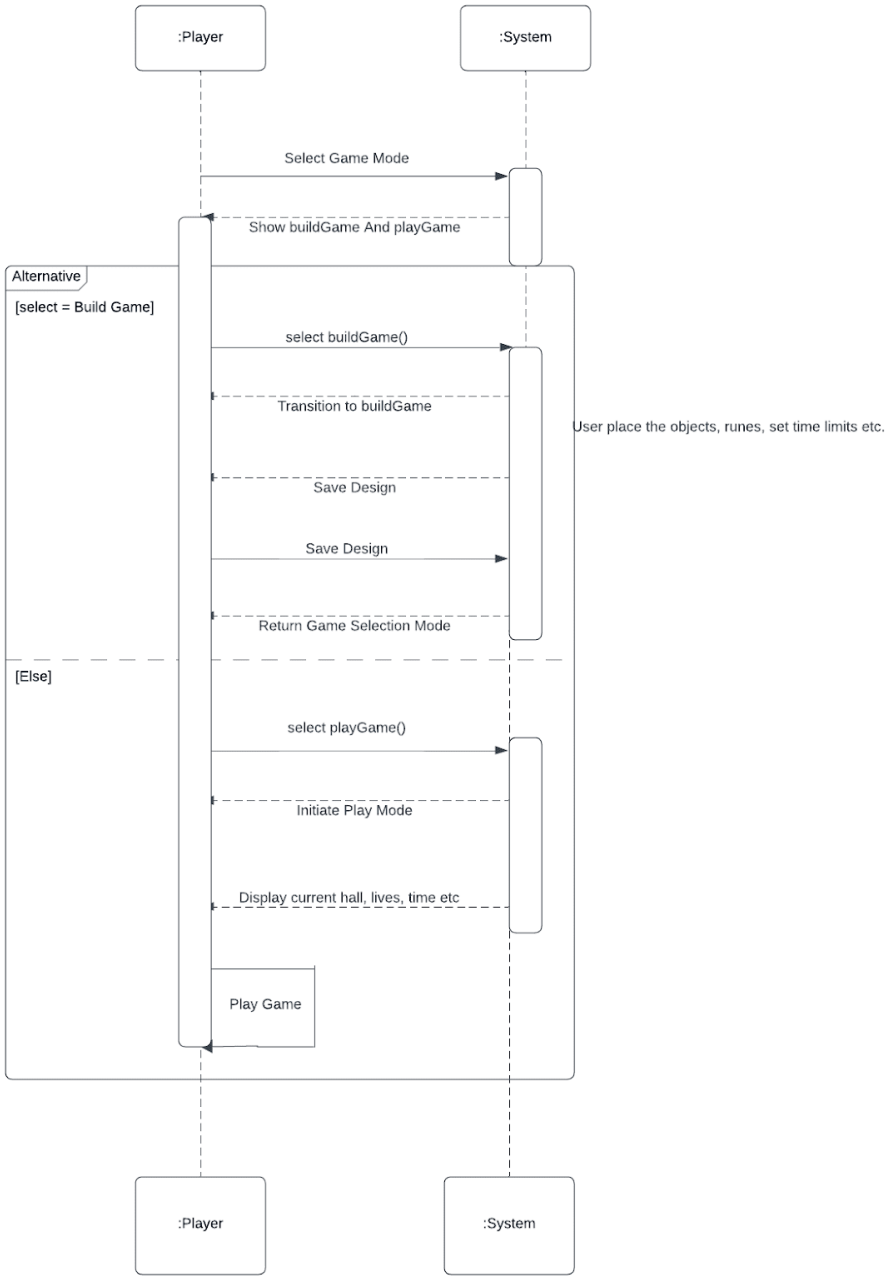


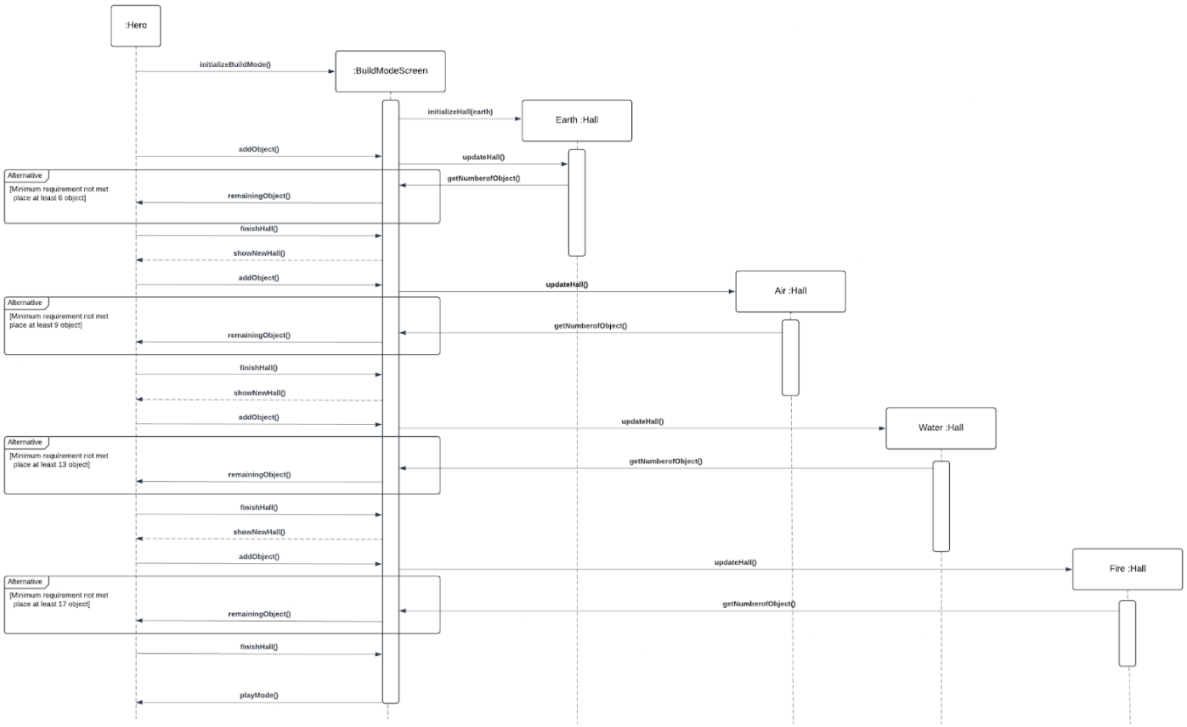
2.Interaction Diagrams

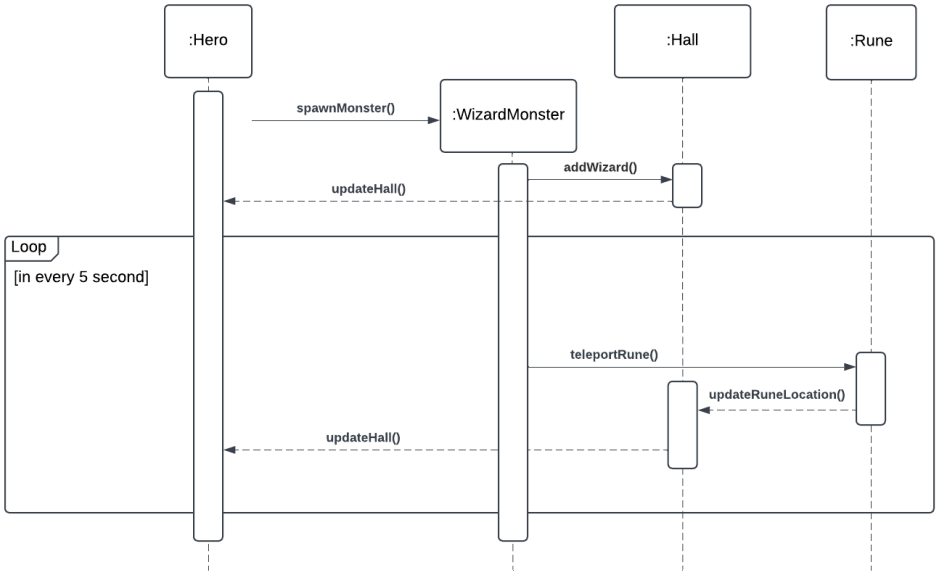


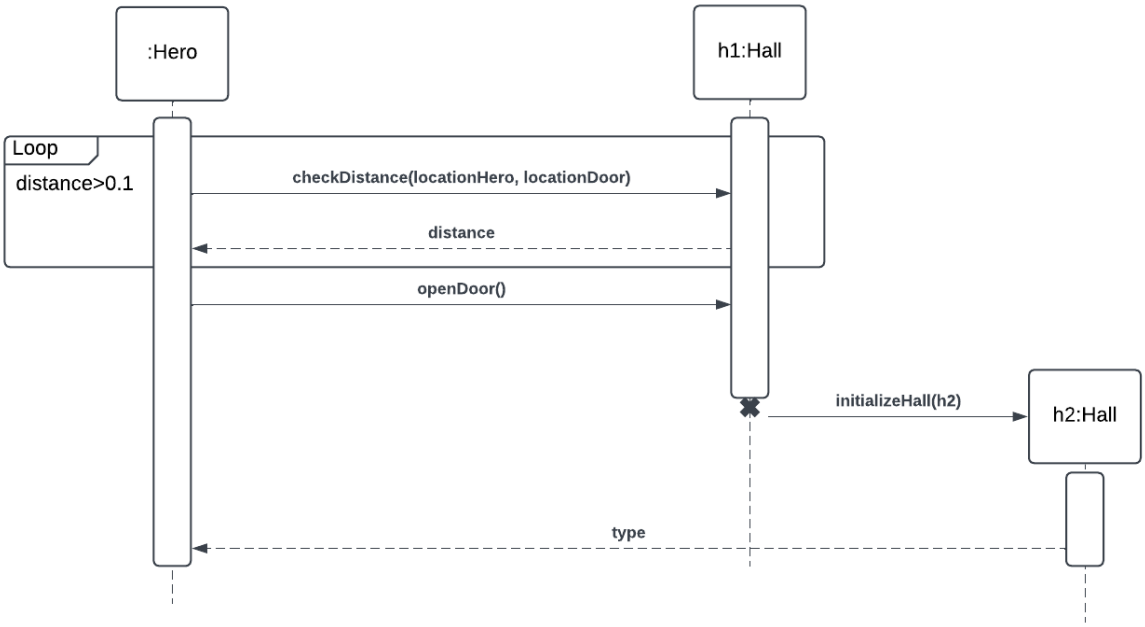


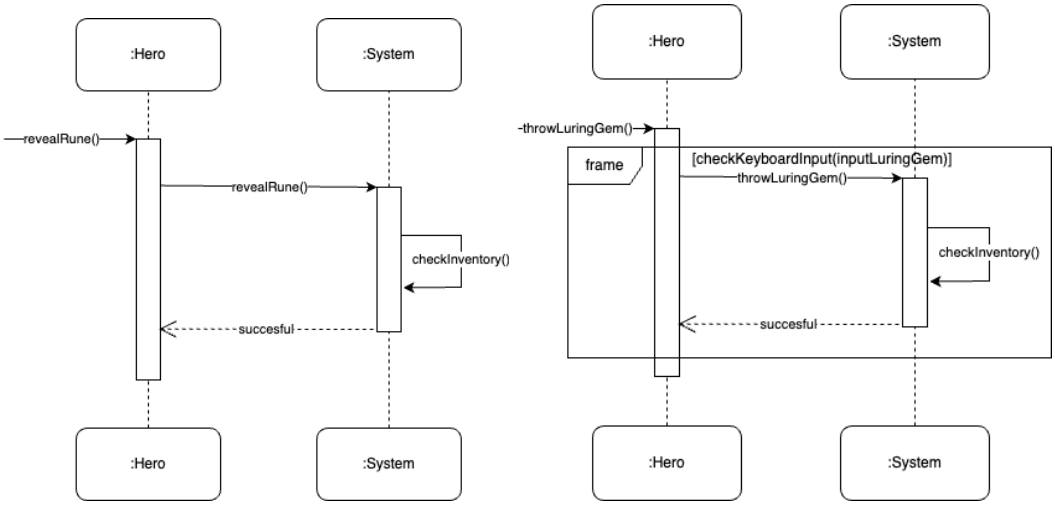


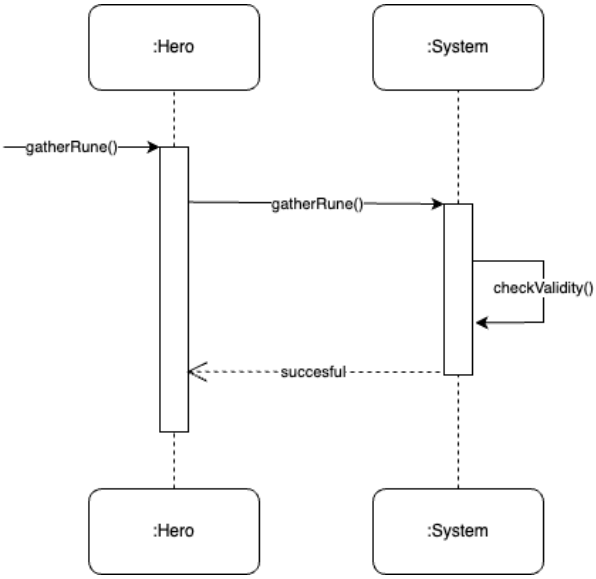

:Player
:System
Select Game Mode
Show buildGame And playGame
Alternative
[select = Build Game]
select buildGame()
Transition to buildGame
User place the objects, runes, set time limits etc.
Save Design
Save Design
Return Game Selection Mode
[Else]
select playGame()
Initiate Play Mode
Display current hall, lives, time etc
Play Game
:Player
:System

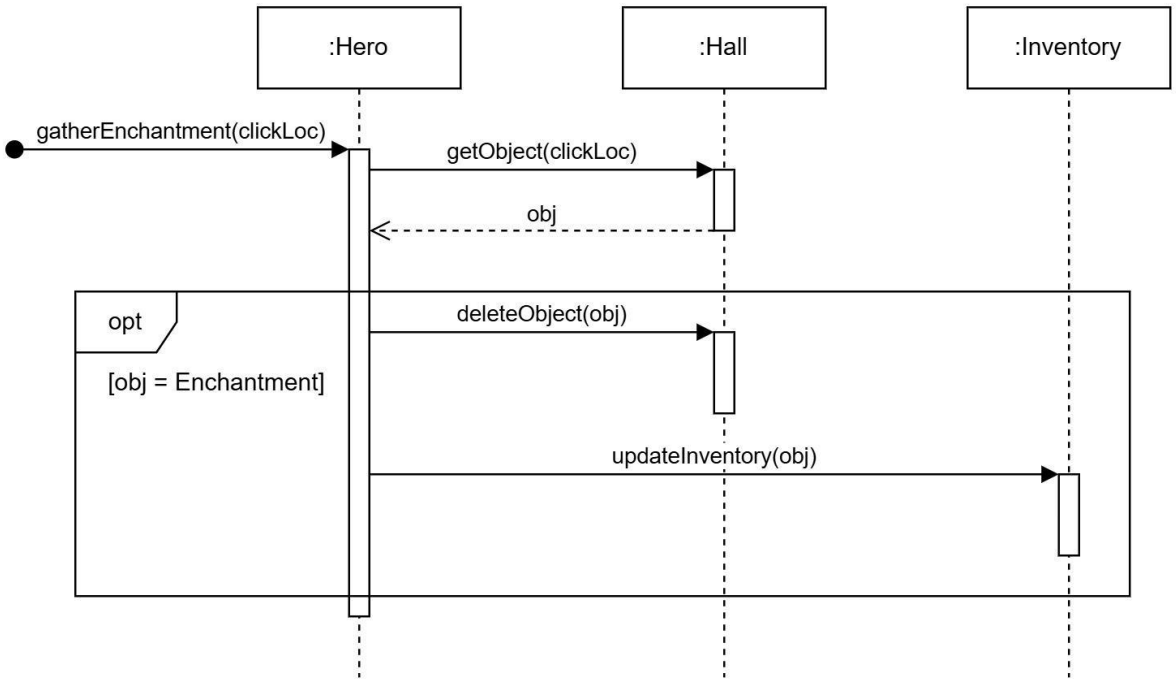

:Hero
initializeBuildMode()
:BuildModeScreen
initializeHall(earth)
Earth :Hall
addObject()
updateHall()
Alternative
[Minimum requirement not met
place at least 6 object]
getNumberofObject()
remainingObject()
finishHall()
showNewHall()
addObject()
updateHall()
Air :Hall
Alternative
[Minimum requirement not met
place at least 9 object]
getNumberofObject()
remainingObject()
finishHall()
showNewHall()
addObject()
updateHall()
Water :Hall
Alternative
[Minimum requirement not met
place at least 13 object]
getNumberofObject()
remainingObject()
finishHall()
showNewHall()
addObject()
updateHall()
Fire :Hall
Alternative
[Minimum requirement not met
place at least 17 object]
getNumberofObject()
remainingObject()
finishHall()
playMode()

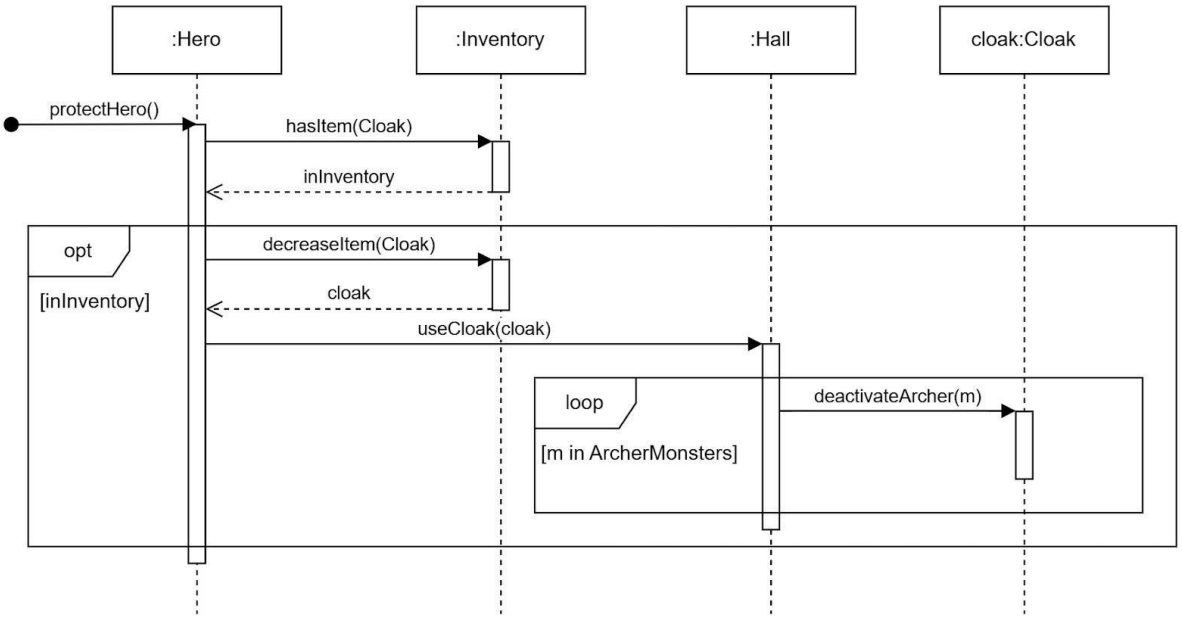

:Hero
:Hall
:Rune
spawnMonster()
:WizardMonster
addWizard()
updateHall()
Loop
[in every 5 second]
teleportRune()
updateRuneLocation()
updateHall()

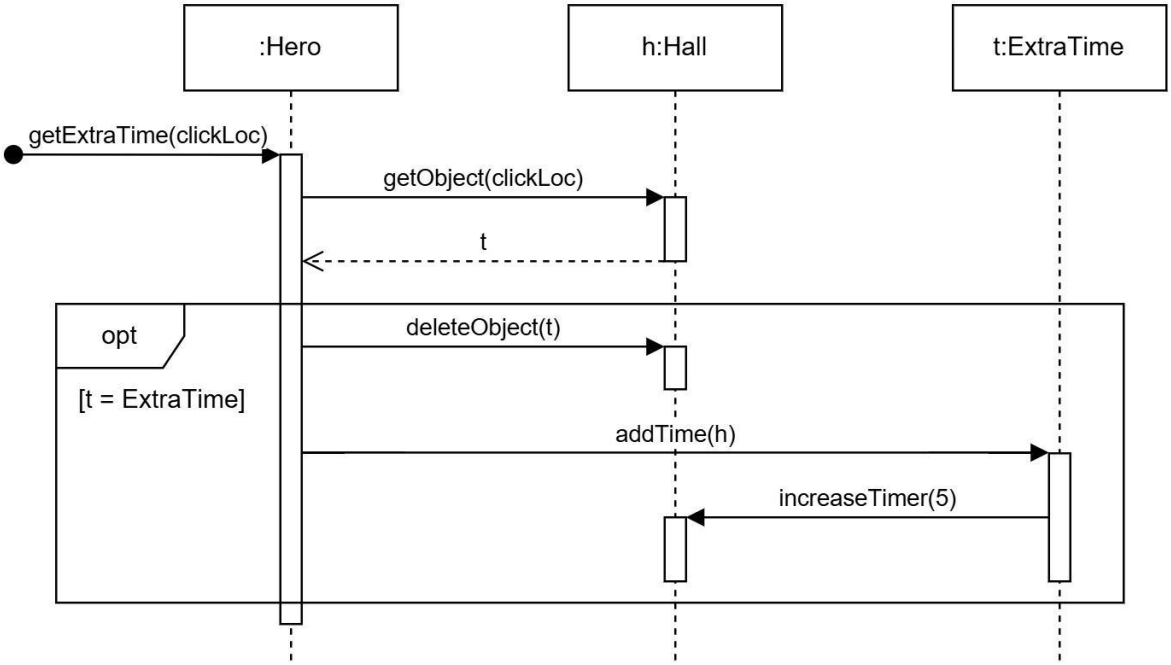

:Hero
h1:Hall
Loop
distance>0.1
checkDistance(locationHero, locationDoor)
distance
openDoor()
initializeHall(h2)
h2:Hall
type


:Hero
:System
revealRune()
revealRune()
checkInventory()
succesful
:Hero
:System
:Hero
:System
-throwLuringGem()
frame
[checkKeyboardInput(inputLuringGem)]
throwLuringGem()
checkInventory()
succesful
:Hero
:System

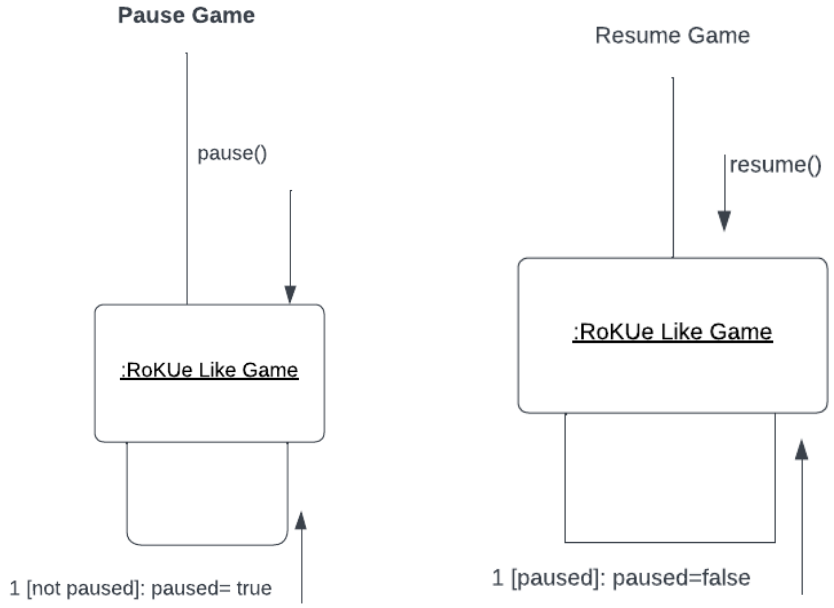

:Hero
:System
gatherRune()
gatherRune()
checkValidity()
succesful
:Hero
:System

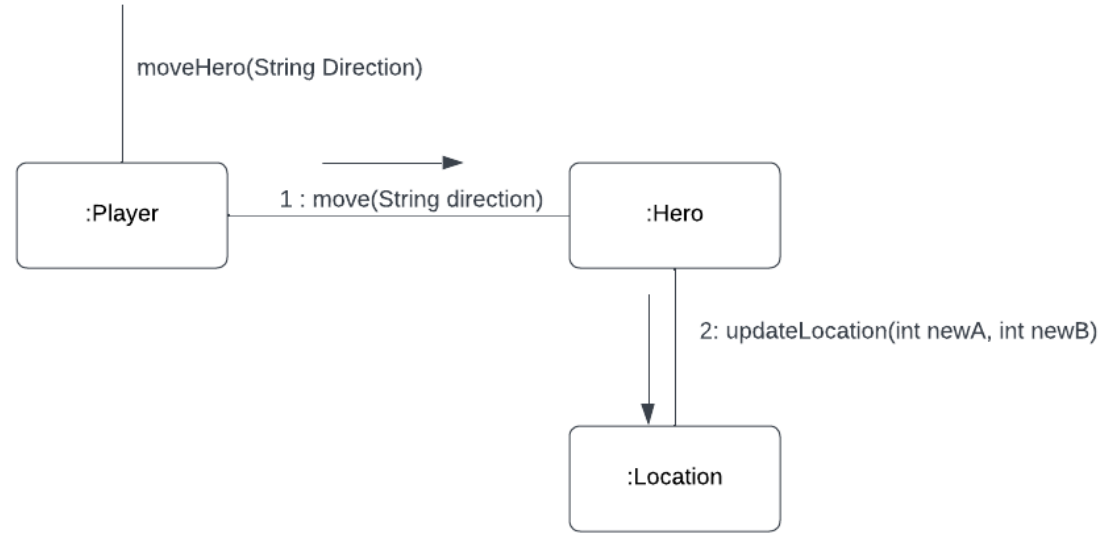

:Hero
:Hall
:Inventory
gatherEnchantment(clickLoc)
getObject(clickLoc)
obj
opt
[obj = Enchantment]
deleteObject(obj)
updateInventory(obj)

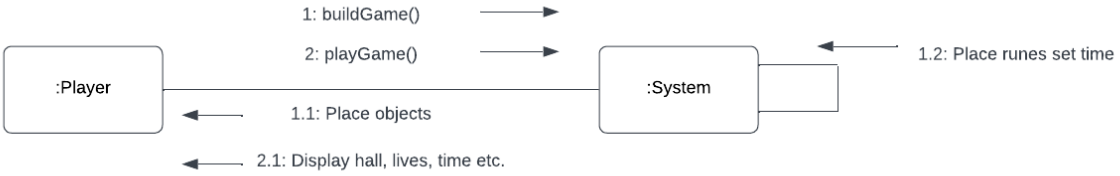

:Hero
:Inventory
:Hall
cloak:Cloak
protectHero()
hasItem(Cloak)
inInventory
opt
[inInventory]
decreaseItem(Cloak)
cloak
useCloak(cloak)
loop
[m in ArcherMonsters]
deactivateArcher(m)

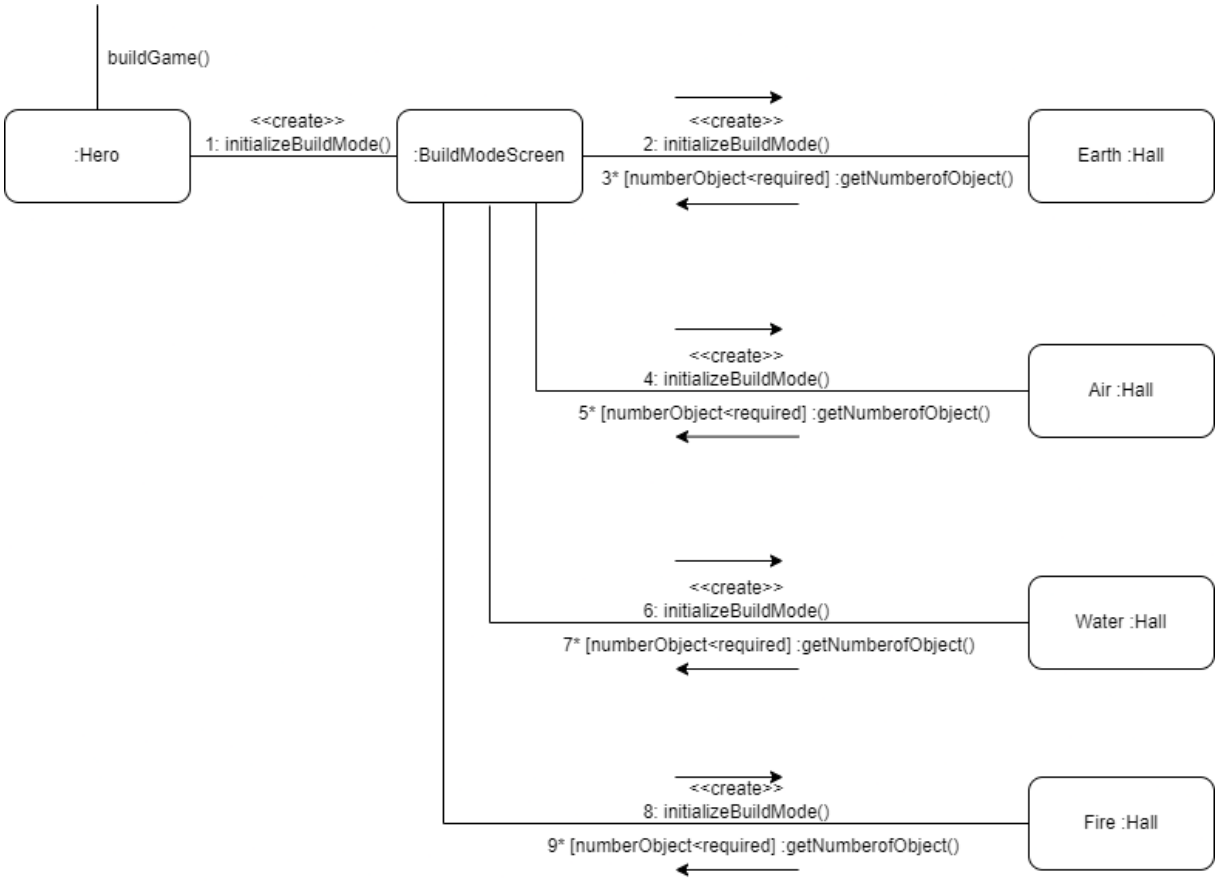


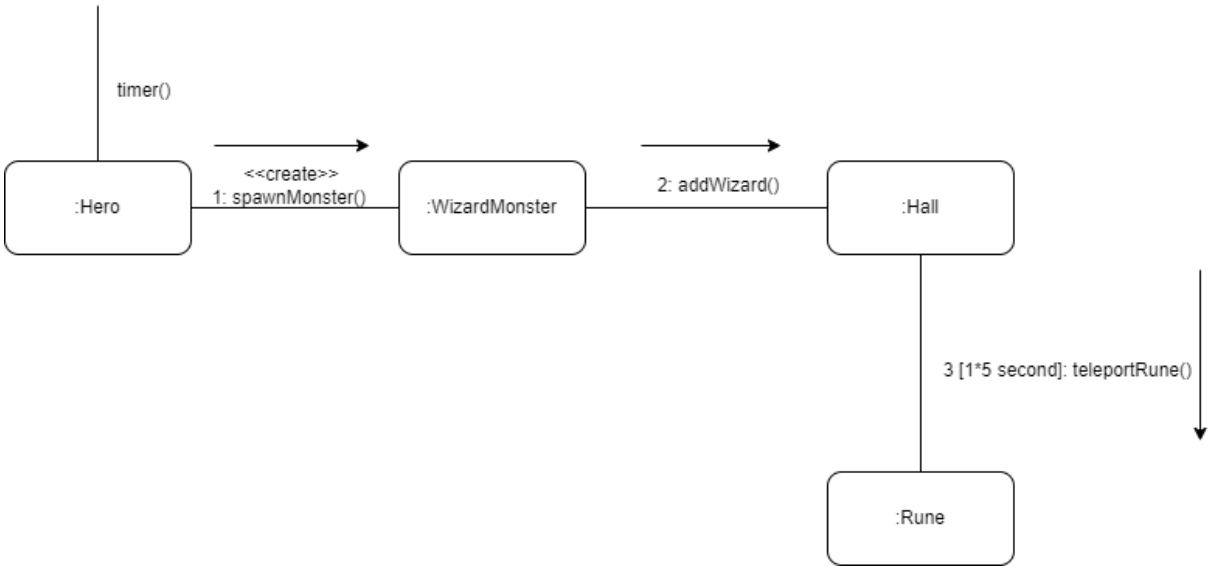
Communication Diagrams



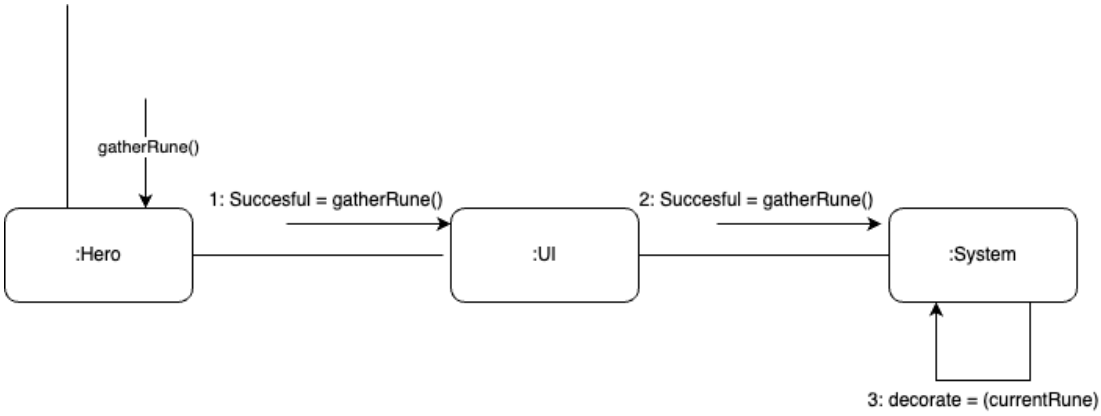

moveHero(String Direction)
:Player
1 : move(String direction)
:Hero
2: updateLocation(int newA, int newB)
:Location

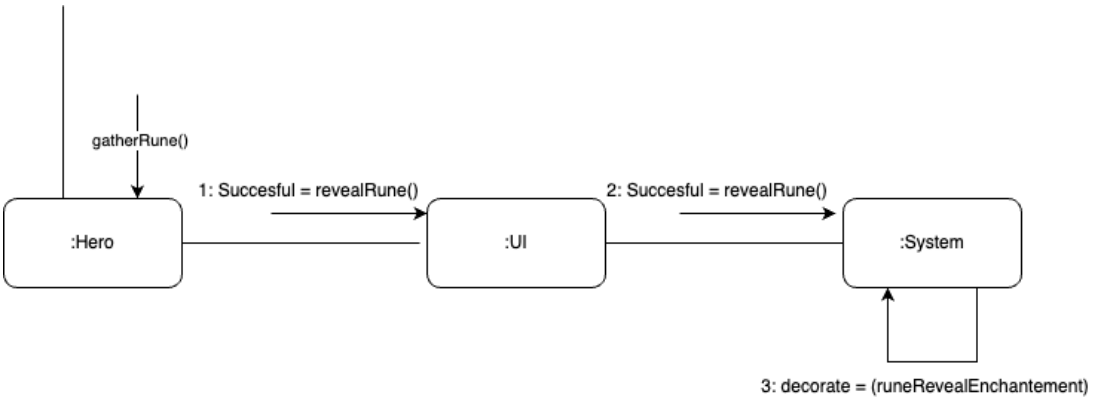

1: buildGame()
2: playGame()
:Player
:System
1.2: Place runes set time
1.1: Place objects
2.1: Display hall, lives, time etc.

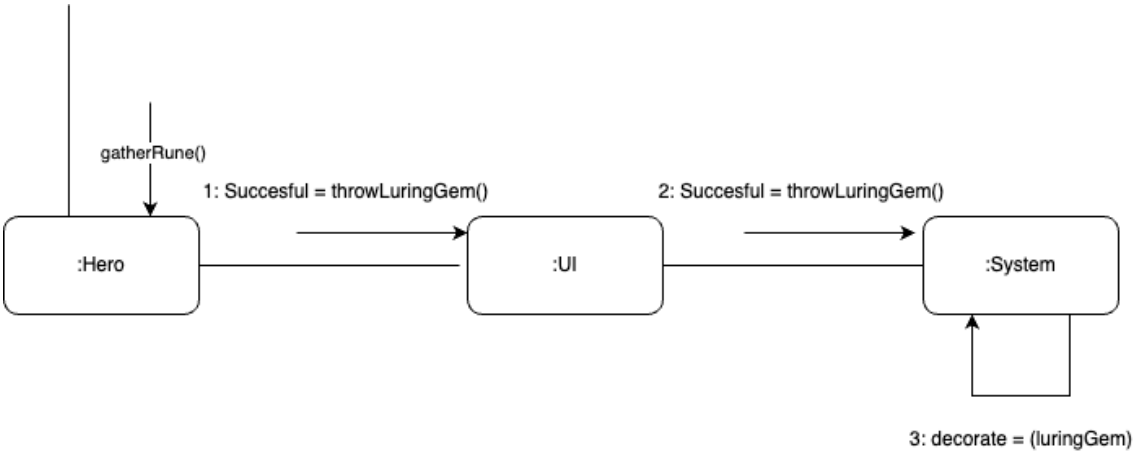

buildGame()
:Hero
<<create>>
1: initializeBuildMode()
:BuildModeScreen
<<create>>
2: initializeBuildMode()
3* [numberObject<required] :getNumberofObject()
Earth :Hall
<<create>>
4: initializeBuildMode()
5* [numberObject<required] :getNumberofObject()
Air :Hall
<<create>>
6: initializeBuildMode()
7* [numberObject<required] :getNumberofObject()
Water :Hall
<<create>>
8: initializeBuildMode()
9* [numberObject<required] :getNumberofObject()
Fire :Hall

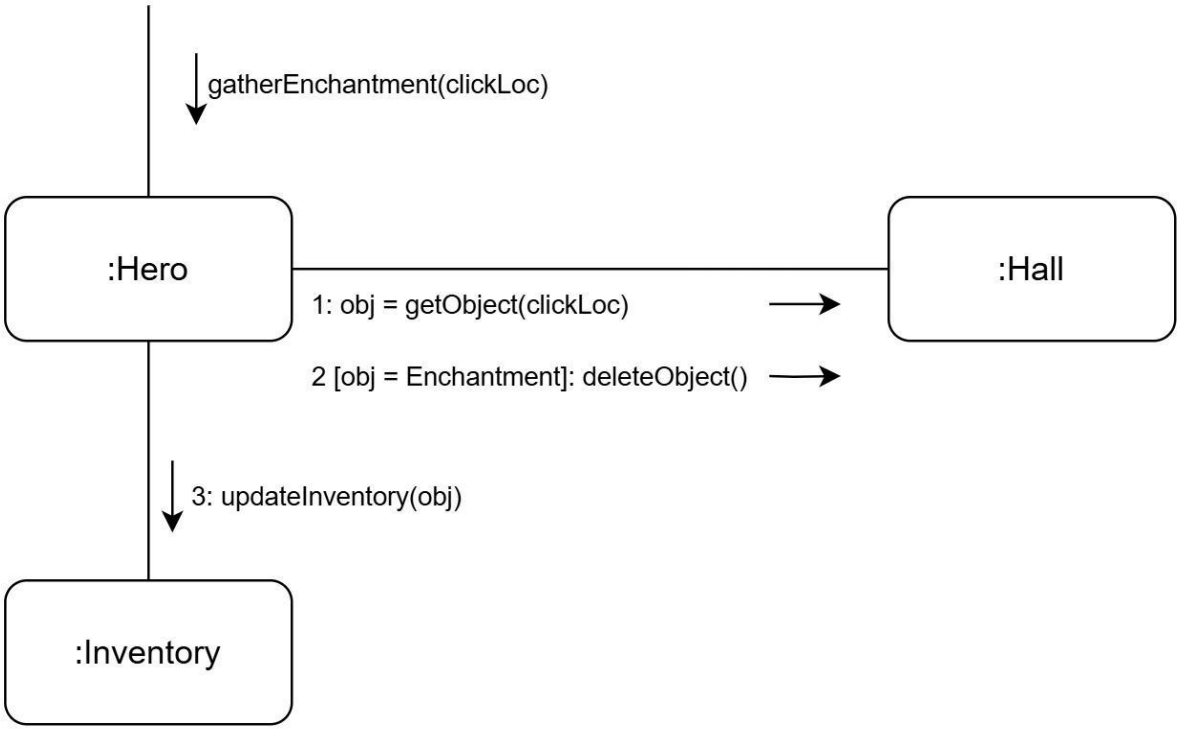

timer()
:Hero
<<create>>
1: spawnMonster()
:WizardMonster
2: addWizard()
:Hall
3 [1*5 second]: teleportRune()
:Rune

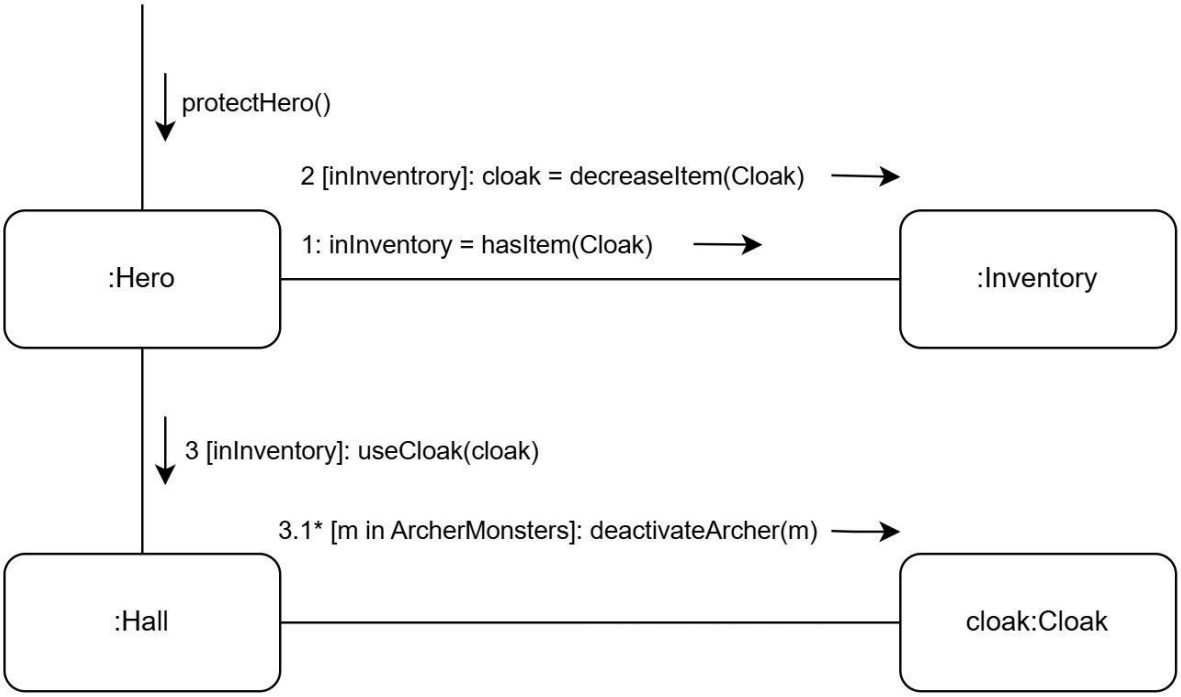

TeleportHero()
:Hero
1 [checkDistance(locationHero, locationDoor) < 1] : openDoor()
h1 :Hall
2: initializeHall(h2)
h2 :Hall

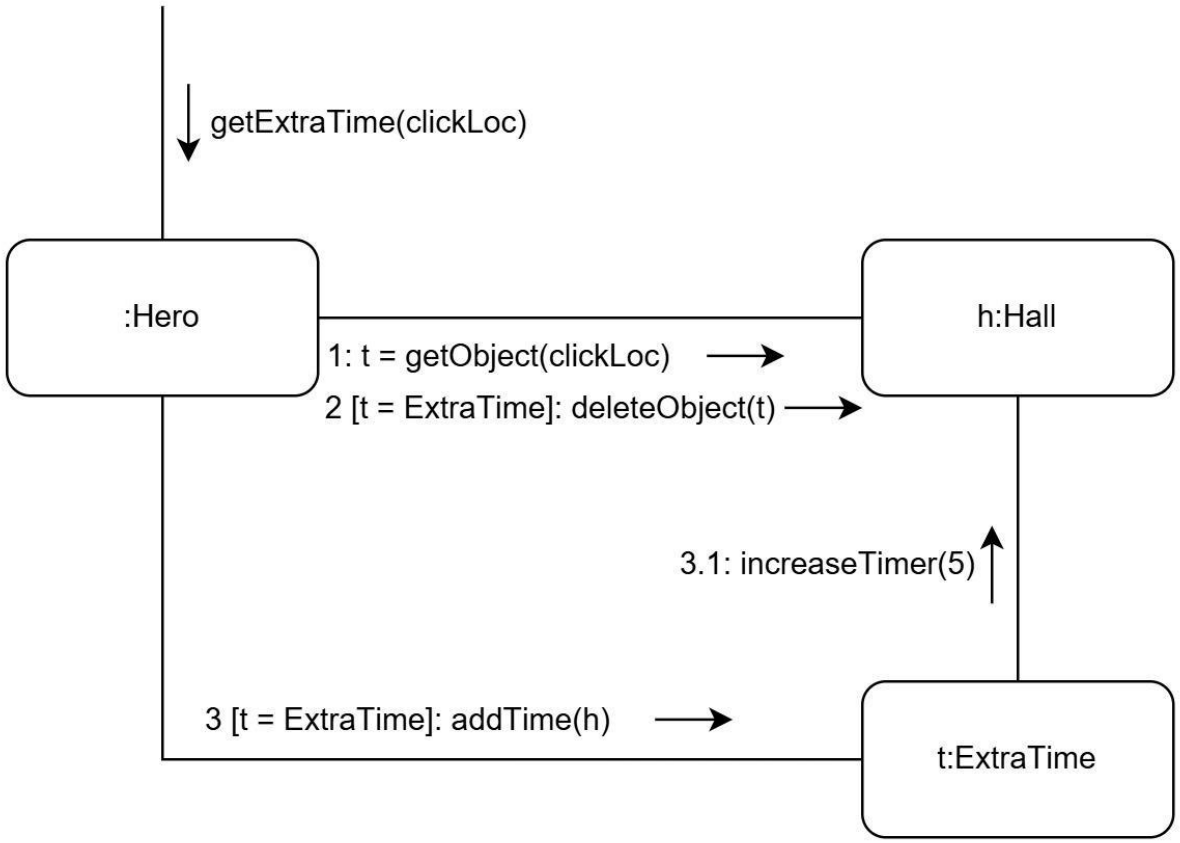

gatherRune()
:Hero
1: Succesful = gatherRune()
:UI
2: Succesful = gatherRune()
:System
3: decorate = (currentRune)


gatherRune()
1: Succesful = revealRune()
2: Succesful = revealRune()
:Hero
:UI
:System
3: decorate = (runeRevealEnchantement)

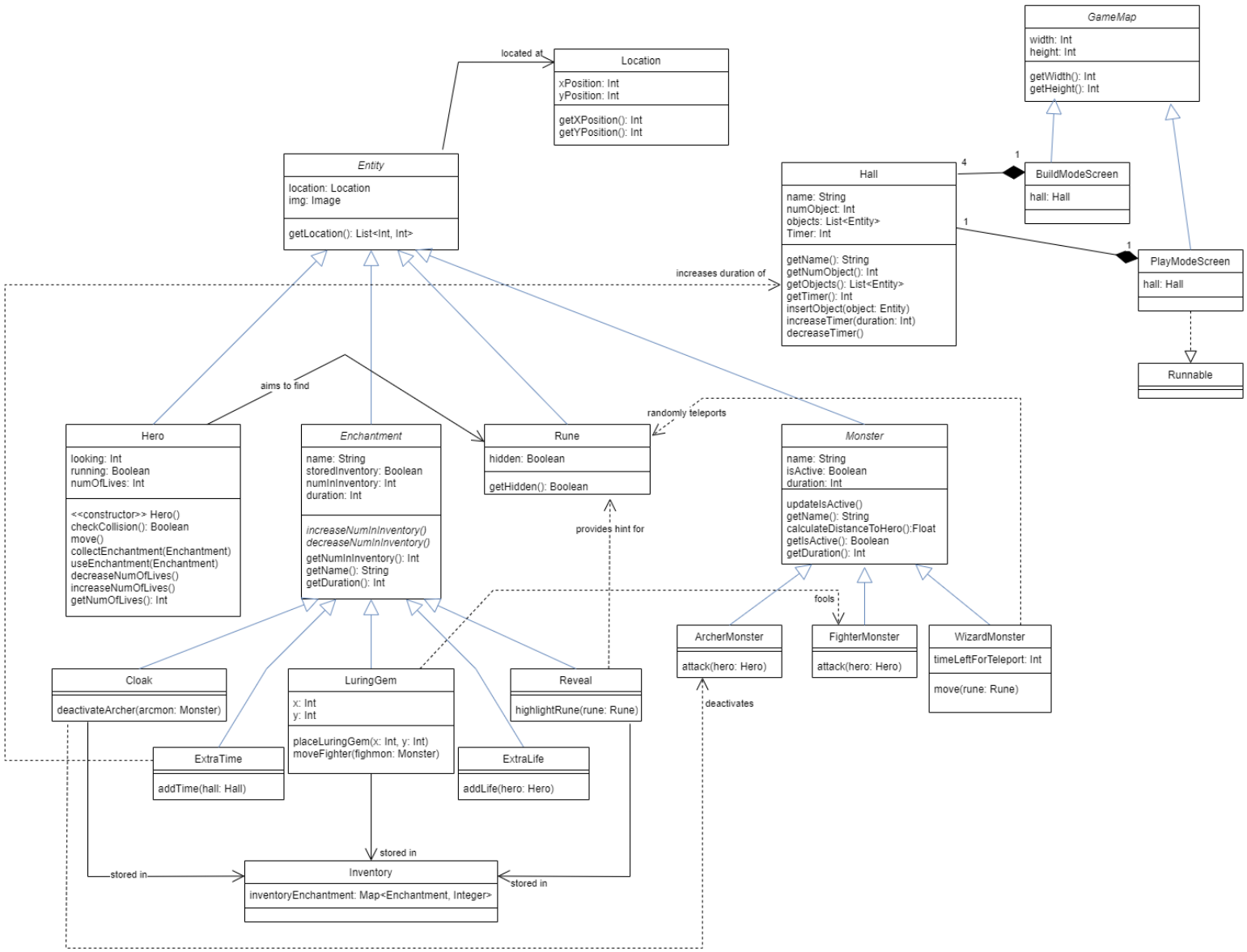

gatherRune()
1: Succesful = throwLuringGem()
2: Succesful = throwLuringGem()
:Hero
:UI
:System
3: decorate = (luringGem)


gatherEnchantment(clickLoc)
:Hero
:Hall
1: obj = getObject(clickLoc)
2 [obj = Enchantment]: deleteObject()
3: updateInventory(obj)
:Inventory


protectHero()
2 [inInventrory]: cloak = decreaseItem(Cloak)
1: inInventory = hasItem(Cloak)
:Hero
:Inventory
3 [inInventory]: useCloak(cloak)
3.1* [m in ArcherMonsters]: deactivateArcher(m)
:Hall
cloak:Cloak


getExtraTime(clickLoc)
:Hero
h:Hall
1: t = getObject(clickLoc)
2 [t = ExtraTime]: deleteObject(t)
3.1: increaseTimer(5)
3 [t = ExtraTime]: addTime(h)
t:ExtraTime

3.UML Class Diagram



4.Pattern Discussion

Design Alternatives for Rokue-Like Game

Design Alternative 1: Open-Ended Hall Selection (Complete Freedom)

Description:
The player can choose any hall in the dungeon to start with and proceed in any order. Each hall is accessible from a central hub (e.g., a starting room) where all available halls are displayed. Once a hall is completed, it is marked as such, and players return to the hub to select the next hall.

Pros:

1. ***Player Freedom:*** Provides maximum control, allowing players to explore the game in their preferred order, enhancing player agency.
2. ***Replayability:*** Different order choices could create varying levels of difficulty depending on how players prioritize challenges.
3. ***Level Testing****:* Players can easily revisit specific halls to test custom designs or strategies without replaying the entire game.

Cons:

1. ***Difficulty Balancing:*** Without a preset order, it can be challenging to balance the difficulty curve, as some players might encounter easier or harder halls first.
2. ***Narrative Impact:*** The absence of a linear progression might dilute the sense of story or accomplishment tied to a fixed sequence.

3. ***Implementation Complexity:*** Designing a central hub with accessible hall selection and tracking completion status requires additional programming and user interface work.

Design Alternative 2: Proximity-Based Melee Combat

Description:
The player can attack monsters by moving close to them (within 1 tile in the grid) and pressing the interact button (used for crates and other actions). Each interaction deals damage to the monster. Different monsters may have varying health, so some might require multiple hits to be defeated.

Pros:

1. ***Diverse Gameplay:***
   Players can make strategic choices between attacking monsters or avoiding them entirely.

2. ***Risk vs. Reward:***

Adds tension, as players need to get close to dangerous enemies, increasing the stakes and excitement.

0. ***Simplified Controls:***

Reuses the existing interact button, maintaining a clean and simple control scheme.

Cons:

Complex Implementation:

Requires significant changes to monster behavior to accommodate health states, damage effects, and death animations.

Difficulty Balancing:

Adding offensive capabilities to the player might make the game too easy if not balanced correctly.

Control Confusion:

Using the same button for attacking and interacting with objects might lead to accidental attacks or unintended interactions in high-pressure situations.

Design Alternative 3: Pet Companions in Rokue-Like

Description:
Introduce companion pets (e.g., dogs, cats, bats) that accompany the hero in the dungeon. Each pet has unique abilities and interacts with the environment or monsters to aid the player. Pets could be chosen at the start of the game or unlocked as rewards.

Pros

Enhanced Player Connection:

Pets add emotional depth and make players feel attached to their companions, increasing engagement.

Freedom of Expression:

Players can choose pets based on their preferred playstyle, enhancing gameplay variety.

Dynamic Gameplay:

Pets introduce new mechanics, such as scouting, distractions, and environmental interactions, adding strategic depth.

Cons

AI Complexity:

Implementing pathfinding, decision-making, and interaction behaviors for pets can be challenging.

Enemy Behavior Adjustments:

New interactions may need additional animations or logic (e.g., archer monsters targeting pets).

Balancing Challenges:

Pets must be helpful without making the game too easy.