Lumiere Final Documentation

CS428

Table of contents:

1. A Brief Description of Lumiere

Lumiere is a 2D medieval based game where a player has to escape the specific dungeon in each level to move on to the next. Each dungeon is randomly generated, and they progressively get larger/harder to escape as the levels go on. Various different tiles exist on the map, and some of them are walkable, while others are not. The player must find the exit door in each level in order to escape, and along the way has to confront multiple different monsters and avoid getting killed. The player also maintains an inventory of items that he has picked up which he can use for various purposes. Two of the most important ones are either attacking monsters, or healing himself. If attacked enough, the monsters will die, and the potions can be used to heal the player to some extent. As the levels go on, more monsters will spawn in addition to the more challenging maps, making it harder for the player to survive. Once he has completed all of the levels, he will win the game.

2. Details Of Our Process

Process Chosen: XP (Extreme Programming)

Platform/Framework Chosen: Unity Game Development Studio

Language Chosen: C# (Mandatory for Unity)

Iterative Development Process:

- The workload was split into several iterations.
    - 6 Total iterations.
    - First iteration was preparatory.
- Each iteration contained between 2 – 4 User Stories.
    - Each user story was given to pairs of two students each.
    - The idea of each pair was to allow both for pair programming, and allow for the main project to be modularized more efficiently.
    - Each user story was to have an estimated amount of time for completion.
    - When completed each story had its actual duration documented on the wiki. (If it took longer or shorter, this was noted on the wiki).
- Each iteration had a duration of around 1 – 2 weeks.
    - Each week the group was to meet once to discuss progress and tentative schedules for their iterations.
    - At the end of each period the group was to prepare a set of slides and a demo for the TA to look over before moving onto the next iteration.
- Each iteration was designed to build off of the others.
- Each iteration was intended to supply a new or improved version of a feature.

Refactoring & Collaborative Work Process:

- The group used GitLab for their code review and version control system.
- Each user story would have its own branch created from the master branch. When a user story was completed, a merge request would be placed.
    - During this merge request, each group member would review each other's code to make sure that it was possible to merge, adhered to our coding standards, and didn't have any glaring issues that needed correction before merging.
    - We usually set a deadline such that all merge requests be in 2 days before the demo time to avoid being stuck doing it last minute.
- In general each subgroup would usually improve the code base they were working on before there were any explicit requirements for refactoring.
- Each iteration starting from the 4th required each individual member of the team to work on their own unique refactoring of previous code.
    - Refactorings were usually related to the user story that was being performed by the individual.
    - In cases where this was not the case, a separate branch would be created and merged while the remainder of user stories were being worked on.

Testing Process:

- Unity has its own testing environment for running various tests.
    - Starting from one of the later iterations, a mandatory minimum of 5 tests as well as adding parameterized tests became part of our standard.
    - In general the individual who wrote a section of code would not be the person writing tests for that code, in cases of pair programming this meant the person not actually doing the typing for the code base would be doing the typing for the tests.
- Each user story subgroup was required to write tests for each of their major functions.
- Our team created our own GitLab CD/CI system that allowed for automatic testing as soon as a commit to the repository was made.
    - The CD/CI was improved each iteration to return more valuable information, in addition to being quicker and more efficient.
    - A CD/CI summarizer script was written in Haskell to provide a clean and brief parse of the Unity test results.

3. Requirements and Specifications of Lumiere

Here we present a list of the implemented user stories and casual use cases related to the user stories. There are fully dressed use cases available at our homework 2 page (https://goo.gl/h4ScAU), but we use casual use cases here to reduce verbosity.

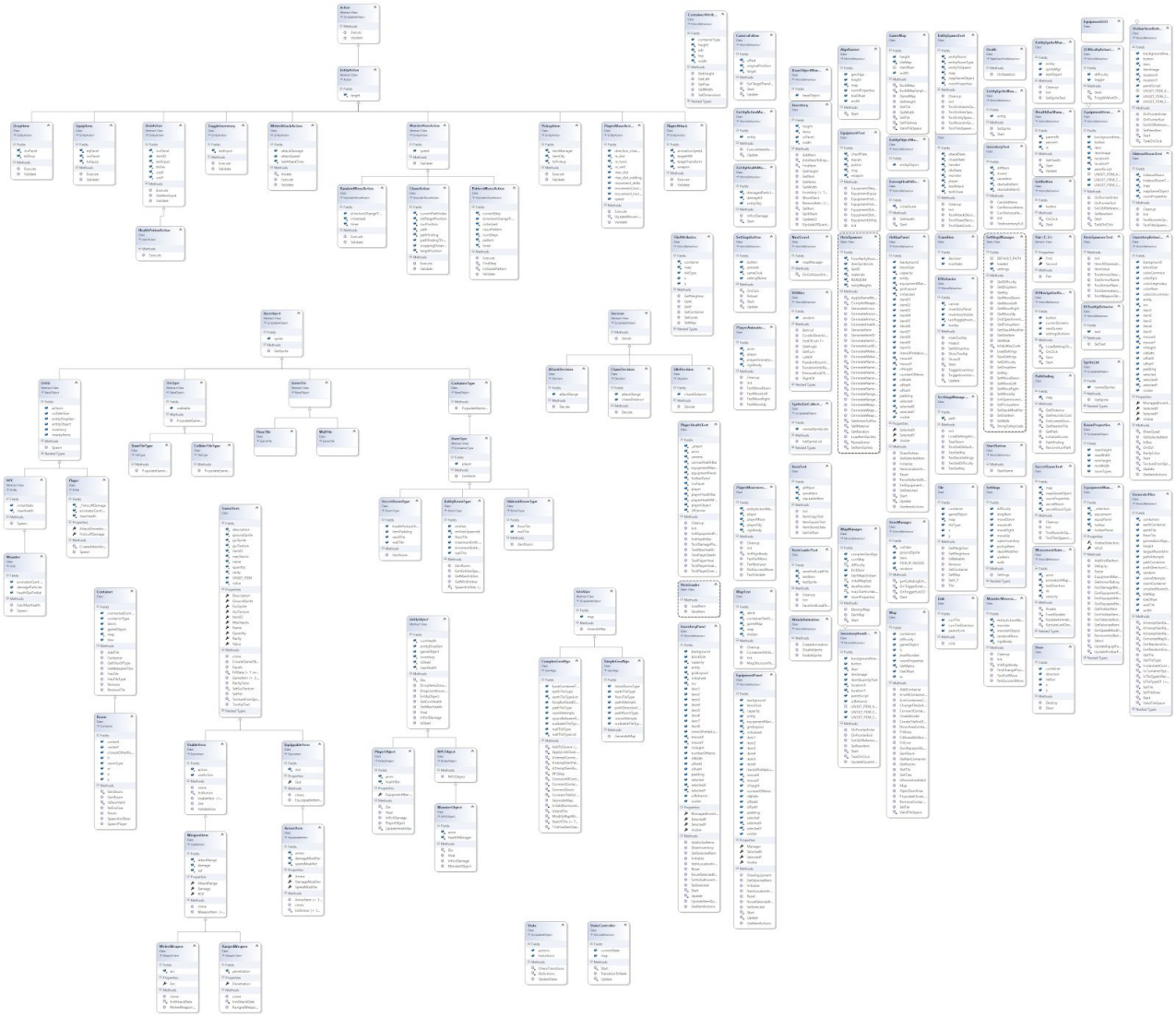
User stories:

1. A basic world is viewable by the user. (j, k)
2. Player can move and explore the world and cannot walk through obstacles/walls and other entities such as monsters. The camera is centered around the player. (g)
3. Basic monsters can move and explore the world and cannot walk through obstacles/walls and other entities. (h)
4. Entities have an inventory that stores items. (a, b, c)
5. The world contains at least 3 types of connected rooms. Each type of room is visually and functionally different. (j, k)
6. Entities are spawned in the world in separate rooms. (j)
7. Players can select items and read item descriptions and effects. e.g. the player's inventory is viewable. (a, b, c)
8. Players can pick up and use items. Items have at least 3 different types of effects on the world or entities. (b, c, d)
9. Entities have health bars that are viewable in-game. Health ranges from 0 to a maximum value. (l)
10. Monsters can navigate to and attack other entities. If an entity's health drops below 0, then this entity dies. (e, h, i)
11. Players can attack and kill other entities using equipped weapons. (f, l, n)
12. Players can equip various weapons and armor and view equipment combat stats. (n)
13. Maps appear more natural and dungeon-like along with having exit doors and the ability to change the difficulty of the level through a single value. (j, k)
14. The player can view a title screen for game settings and information and a game over screen. (m)
15. The user will be able to view an animation for every action an entity can perform. (o)
16. The user can experience a full game loop, starting from the title screen, to playing the game and moving through levels via finding the level's exit door, to losing and seeing the game over screen and then back to the title screen. (m)
17. The further the user progresses in the game, all aspects of the game become more difficult - this includes monsters becoming tougher, dungeon layouts become larger and more complex, items become better to combat the tougher enemies. (j)
18. The user can find loot in treasure rooms or upon defeating monsters. (j)

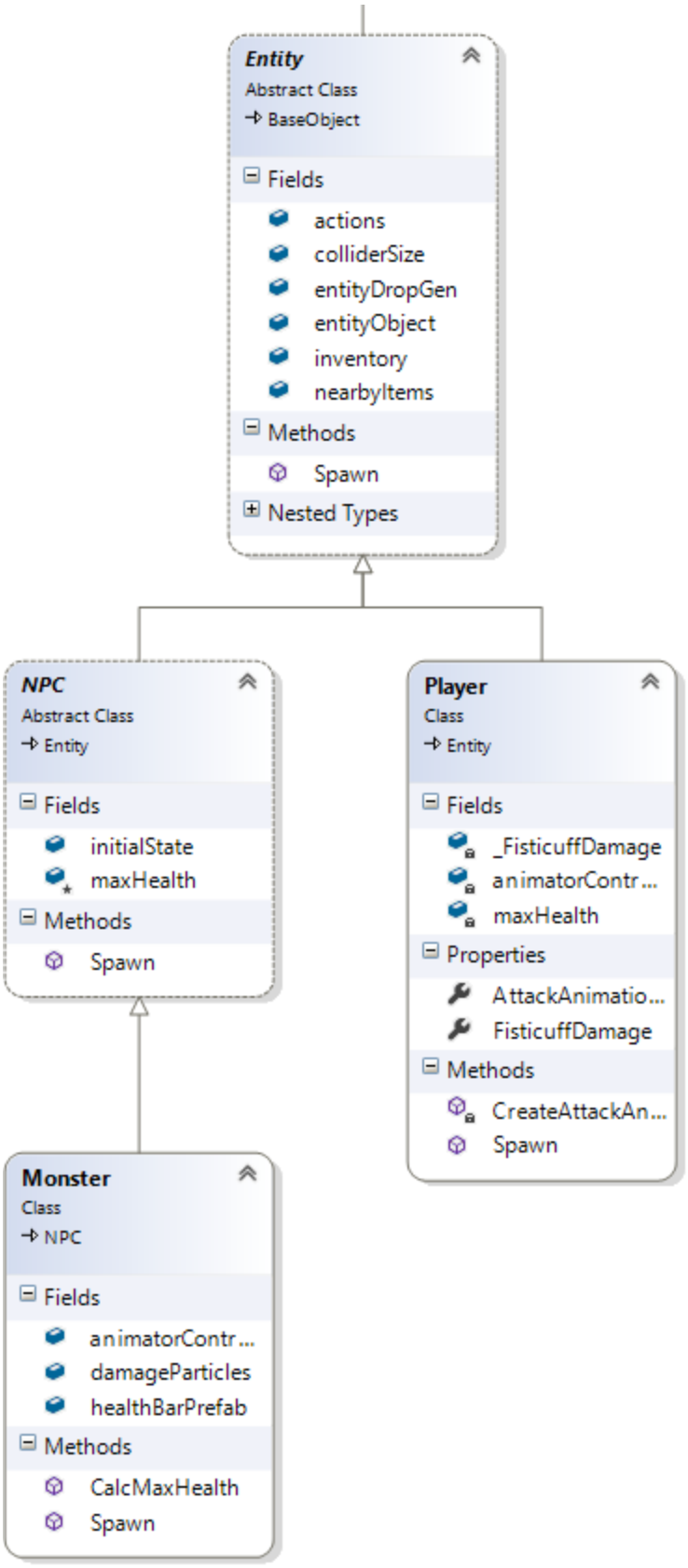
Use cases:

(a) The player can press a hotkey and view his own inventory, which displays the items he currently owns as well as their descriptions, effects, and quantity.
(b) The player can interact with his inventory and drop an item.
(c) The player can interact with his inventory and use an item.
(d) The player can pick up free standing items and stacks from the world and place them in his inventory if there is space.
(e) Entities can die if they take too much damage and run out of health.
(f) The player can attack monsters and cause them to die.
(g) The player can move around the world and interact with world entities by not walking through walls.
(h) Monsters will be able to path intelligently around the world, pathing somewhat efficiently towards the player if conditions are met.
(i) Monsters will be able to attack the player when they get close enough.
(j) The world can generate random levels for players and monsters to interact in and these levels increase in difficulty as the game progresses.
(k) The world will be able to change its appearance by switching out a tilesets for levels with a different look and feel.
(l) The UI is able to display the player's current health status as well as his currently equipped armor and weapons.
(m) The UI will have an initial title screen and a terminal game over screen to provide a polished look and feel to the game.
(n) The player can equip armor and weapons from his inventory.
(o) When actions are performed, a corresponding animation is performed.

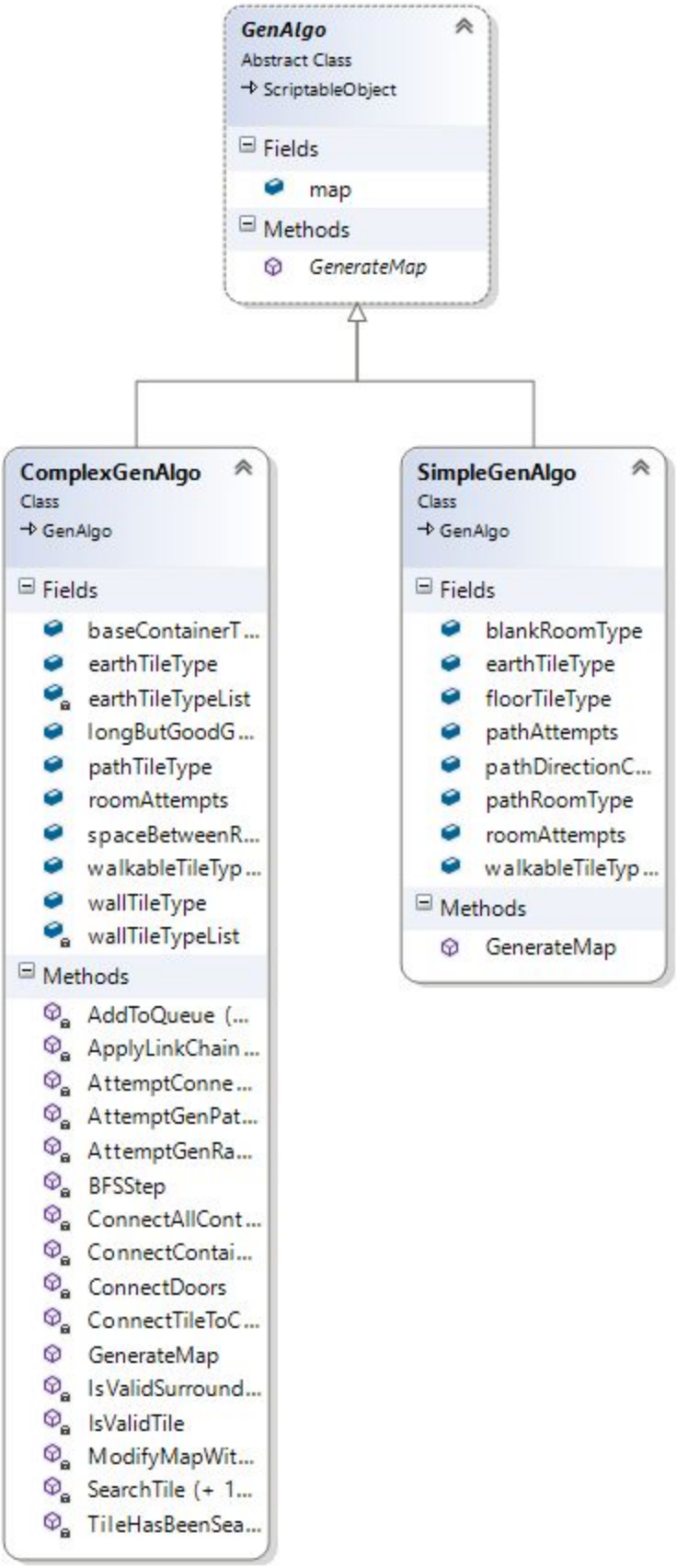
4. Architecture & Design



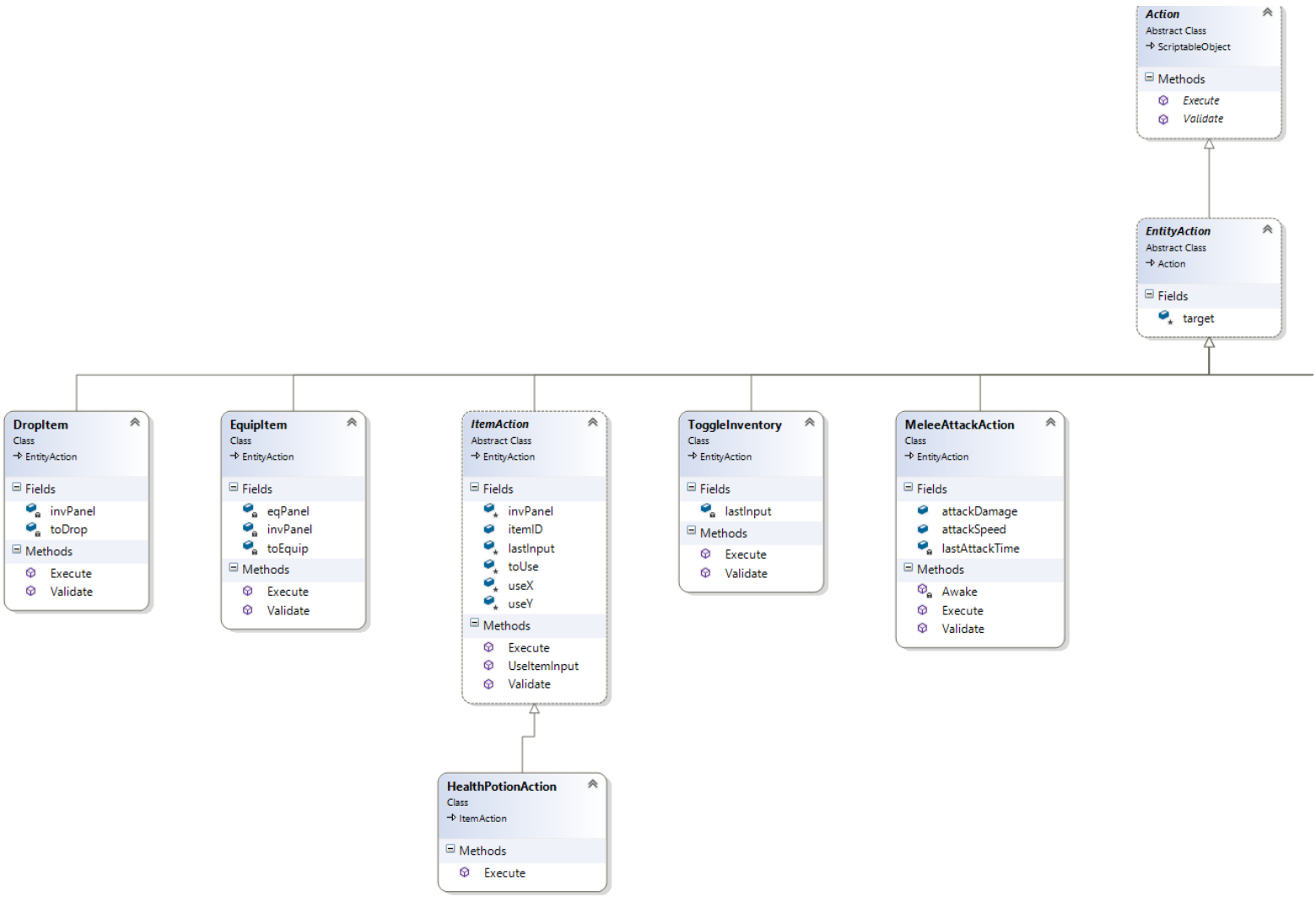
Entity


Entity
Abstract Class
BaseObject
Fields
actions
colliderSize
entityDropGen
entityObject
inventory
nearbyItems
Methods
Spawn
Nested Types
NPC
Abstract Class
Entity
Fields
initialState
maxHealth
Methods
Spawn
Player
Class
Entity
Fields
_FisticuffDamage
animatorContr...
maxHealth
Properties
AttackAnimatio...
FisticuffDamage
Methods
CreateAttackAn...
Spawn
Monster
Class
NPC
Fields
animatorContr...
damageParticles
healthBarPrefab
Methods
CalcMaxHealth
Spawn

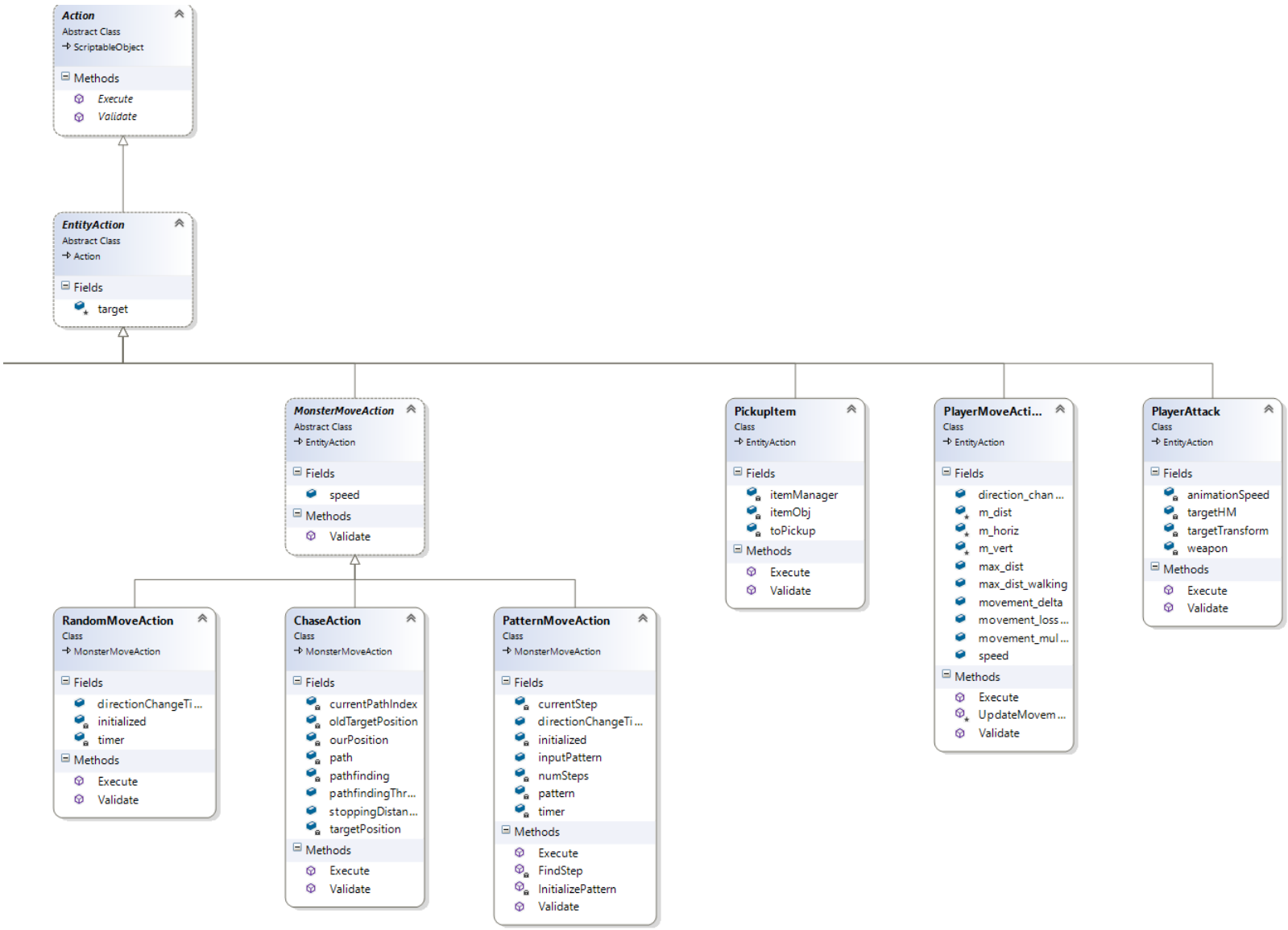
Generation Algorithm



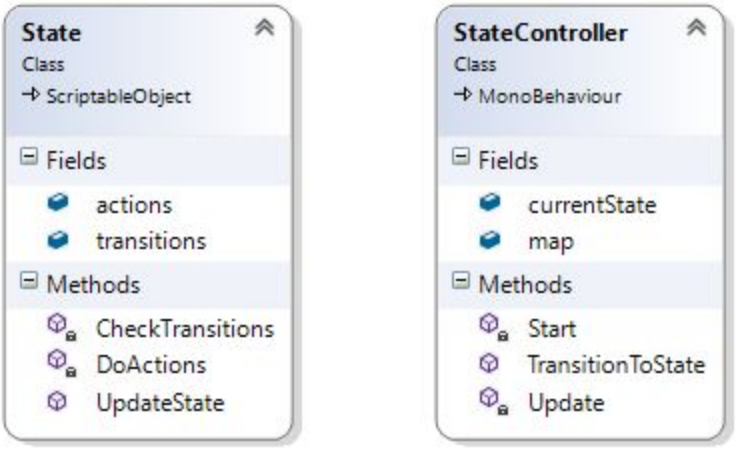
Actions


Action
Abstract Class
ScriptableObject
Methods
Execute
Validate
EntityAction
Abstract Class
Action
Fields
target
DropItem
Class
EntityAction
Fields
invPanel
toDrop
Methods
Execute
Validate
EquipItem
Class
EntityAction
Fields
eqPanel
invPanel
toEquip
Methods
Execute
Validate
ItemAction
Abstract Class
EntityAction
Fields
invPanel
itemID
lastInput
toUse
useX
useY
Methods
Execute
UseItemInput
Validate
ToggleInventory
Class
EntityAction
Fields
lastInput
Methods
Execute
Validate
MeleeAttackAction
Class
EntityAction
Fields
attackDamage
attackSpeed
lastAttackTime
Methods
Awake
Execute
Validate
HealthPotionAction
Class
ItemAction
Methods
Execute

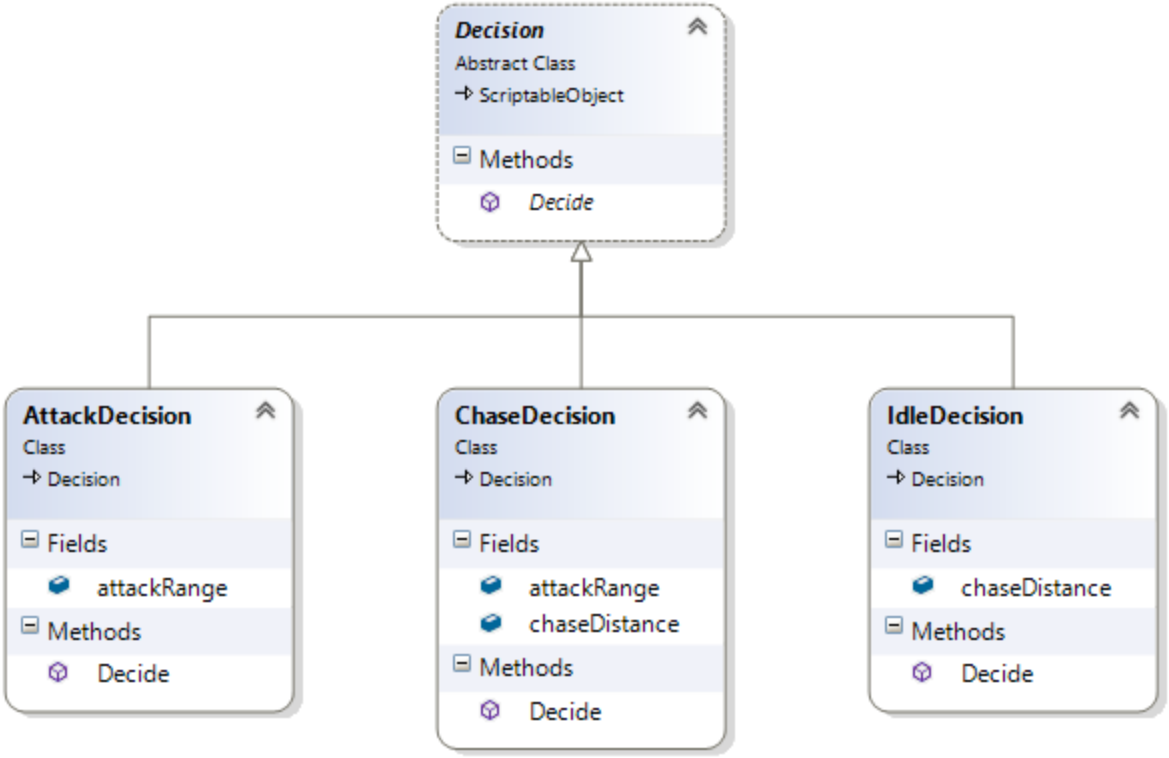
Actions cont.



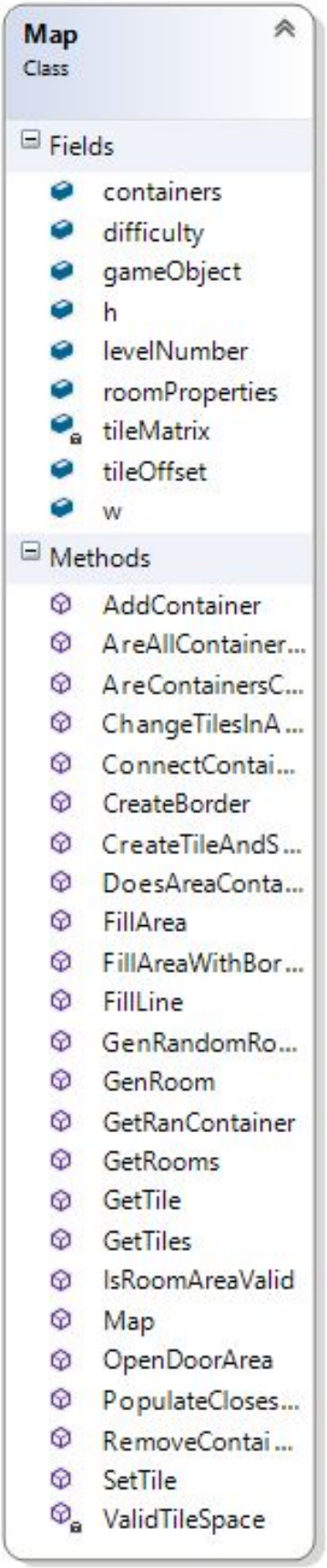
State



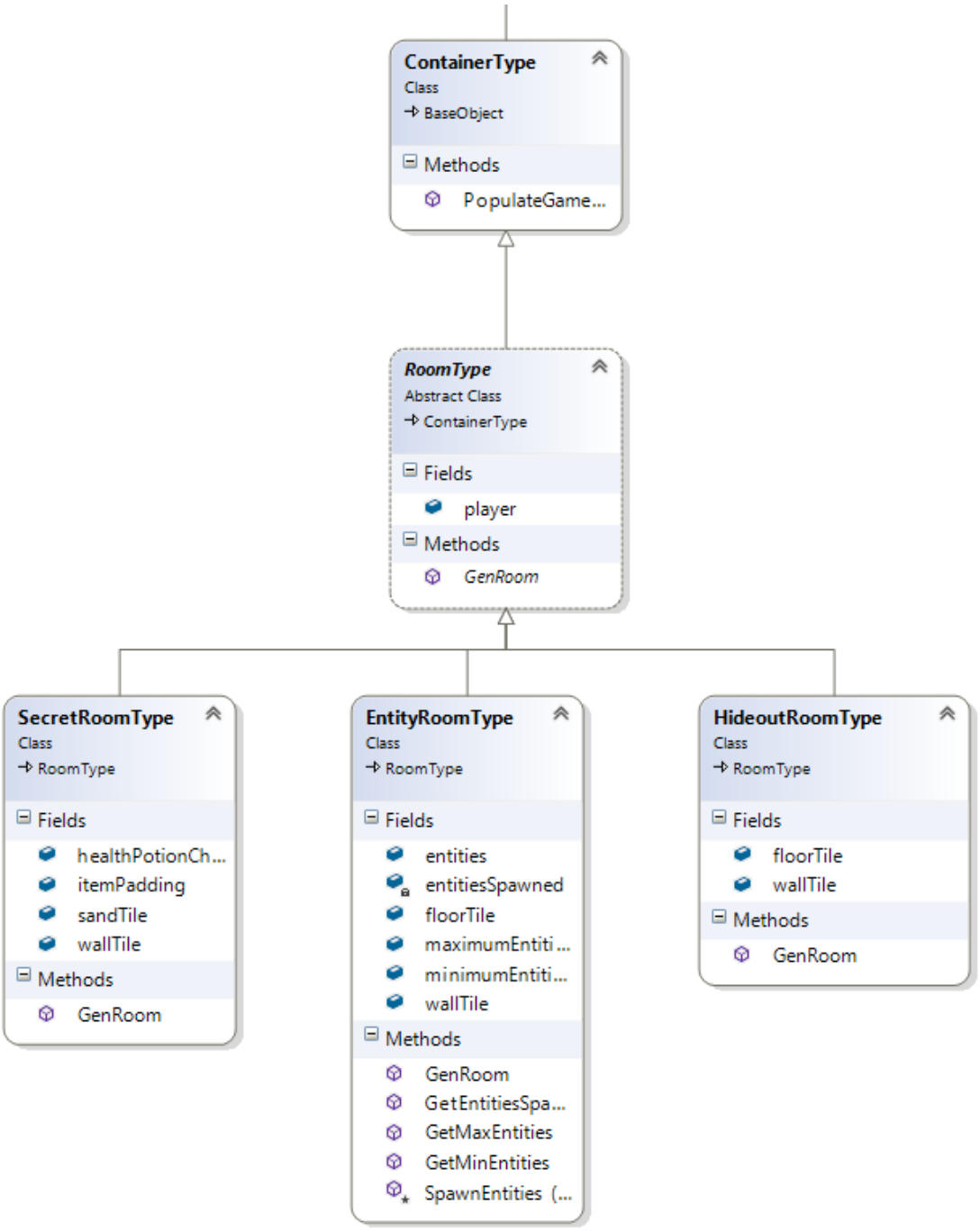
Decisions



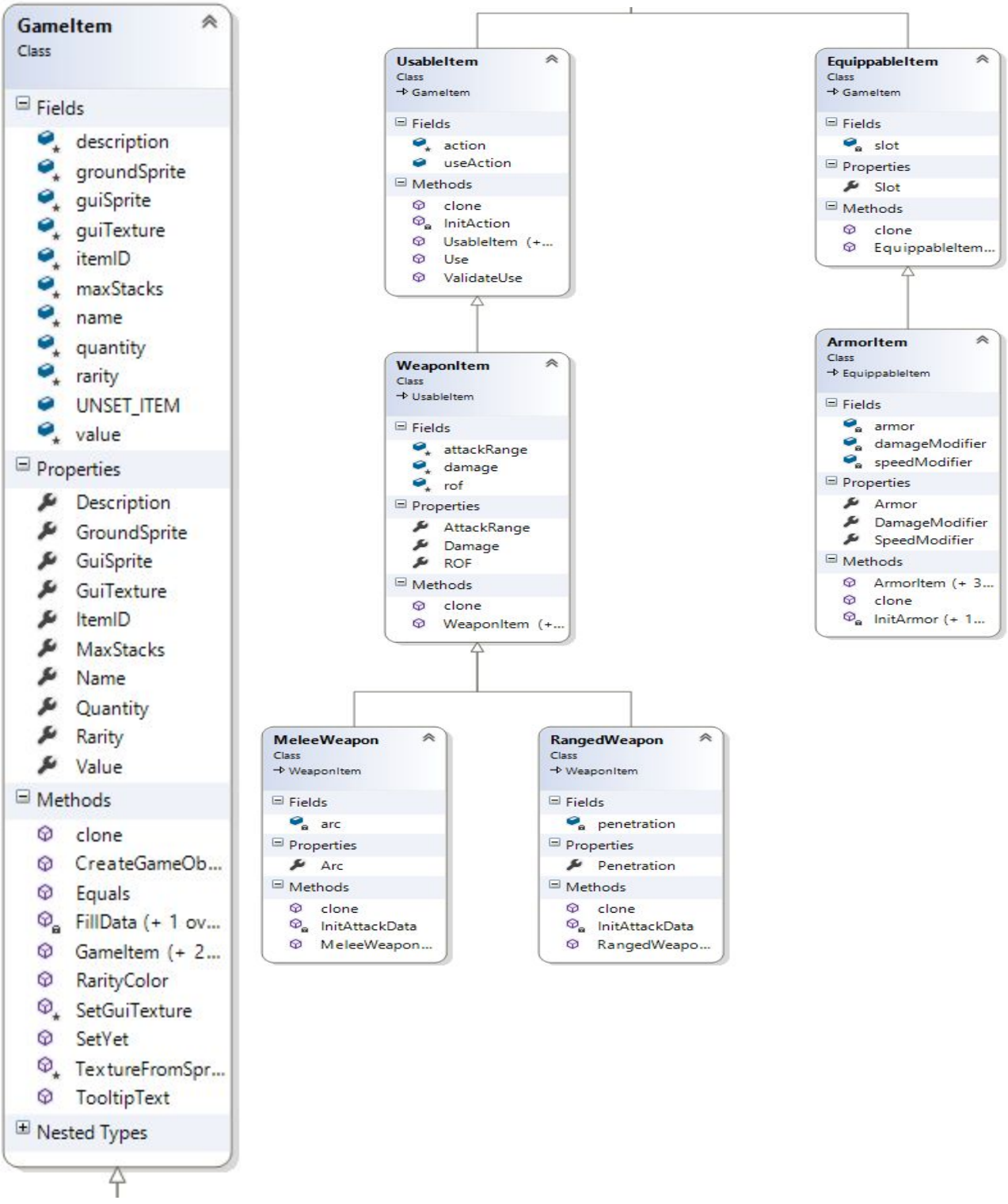
Map


Map
Class
Fields
containers
difficulty
gameObject
h
levelNumber
roomProperties
tileMatrix
tileOffset
w
Methods
AddContainer
AreAllContainer...
AreContainersC...
ChangeTilesInA...
ConnectContai...
CreateBorder
CreateTileAndS...
DoesAreaConta...
FillArea
FillAreaWithBor...
FillLine
GenRandomRo...
GenRoom
GetRanContainer
GetRooms
GetTile
GetTiles
IsRoomAreaValid
Map
OpenDoorArea
PopulateCloses...
RemoveContai...
SetTile
ValidTileSpace

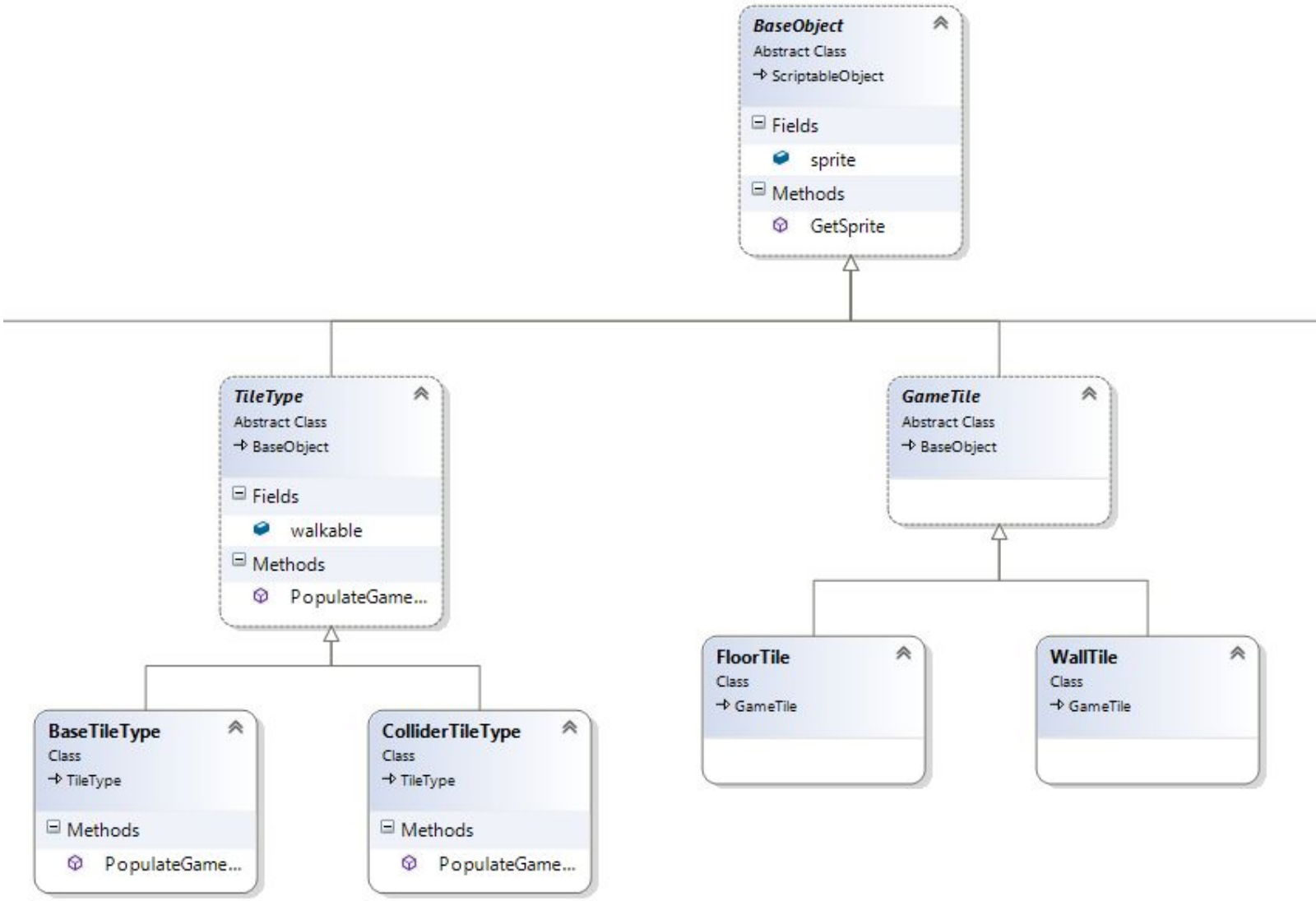
Container Type



Game Item


GameItem
Class
Fields
description
groundSprite
guiSprite
guiTexture
itemID
maxStacks
name
quantity
rarity
UNSET_ITEM
value
Properties
Description
GroundSprite
GuiSprite
GuiTexture
ItemID
MaxStacks
Name
Quantity
Rarity
Value
Methods
clone
CreateGameOb...
Equals
FillData (+ 1 ov...
GameItem (+ 2...
RarityColor
SetGuiTexture
SetYet
TextureFromSpr...
TooltipText
Nested Types
UsableItem
Class
GameItem
Fields
action
useAction
Methods
clone
InitAction
UsableItem (+...
Use
ValidateUse
EquippableItem
Class
GameItem
Fields
slot
Properties
Slot
Methods
clone
EquippableItem...
WeaponItem
Class
UsableItem
Fields
attackRange
damage
rof
Properties
AttackRange
Damage
ROF
Methods
clone
WeaponItem (+...
ArmorItem
Class
EquippableItem
Fields
armor
damageModifier
speedModifier
Properties
Armor
DamageModifier
SpeedModifier
Methods
ArmorItem (+ 3...
clone
InitArmor (+ 1...
MeleeWeapon
Class
WeaponItem
Fields
arc
Properties
Arc
Methods
clone
InitAttackData
MeleeWeapon...
RangedWeapon
Class
WeaponItem
Fields
penetration
Properties
Penetration
Methods
clone
InitAttackData
RangedWeapo...

Tile Types


BaseObject
Abstract Class
ScriptableObject
Fields
sprite
Methods
GetSprite
TileType
Abstract Class
BaseObject
Fields
walkable
Methods
PopulateGame...
GameTile
Abstract Class
BaseObject
BaseTileType
Class
TileType
Methods
PopulateGame...
ColliderTileType
Class
TileType
Methods
PopulateGame...
FloorTile
Class
GameTile
WallTile
Class
GameTile

Description and motivation of system architecture

Our system consists of several components/modules that interact with each other. Each component has its own sub-components that inherit features from their ancestors and have additional features of their own. This component based design allows for modularity and scalability.

Several important components inherit from `ScriptableObject`. A `ScriptableObject` is an object that can be serialized and saved as an asset in Unity and its properties can be modified and configured in the Unity Editor. By allowing components to inherit from `ScriptableObject`, we can create custom assets that game designers are able to easily modify within Unity. `ScriptableObject`s can also contain functions and methods that are specific to certain types of assets which allows us to create scalable and extensible assets for our game.

A `BaseObject` is a simple `ScriptableObject` that has a sprite. All components that have sprites therefore inherit from `BaseObject`. An `Entity` is a BaseObject that represents any interactive object or character in the world. `Entity` has a `Spawn()` method that is called by our map generation algorithm to generate a physical representation of the entity in game. A `Player` is an `Entity` that represents the character the user controls in the game. `NPC` is another type of `Entity`, which is an acronym for non-player character and is an `Entity` that is controlled by the AI. `Monster` is a subtype of `NPC` that is strictly hostile to the player. `Player` contains an array of `EntityAction` which represent a list of all the actions the player character can execute in the game. Actions are described in more detail below.

An `Action` is a `ScriptableObject` that represents all actions that can be executed in the game. `Action` has a `Validate()` method that is called before execution to check whether an action can and should be executed and an `Execute()` method to execute it. Both `Validate()` and `Execute()` return a `bool` that represents whether an action should be executed and whether an action was successfully executed respectively. `EntityAction` is any `Action` that is performed by an `Entity`. Therefore, any custom `Action` that is executed by an `Entity` should be an `EntityAction`.

A `State` is a `ScriptableObject` that represents a specific state within the AI State Machine. Every State contains an array of `Action` that represents actions that need to be executed in the current state. `State` links up with `StateController` which controls the flow between different states and acts as a state machine. Each `NPC` stores an initial `State`. `State` contains an array of `Transition`. Each `Transition` is simply a container for a `Decision` and a resulting `State`. A `Decision` is a ScriptableObject that determines whether

a specific `Transition` should be taken or not. This functionality is provided by the `Decide()` function which returns a `bool` that determines whether the object should transition to the next `State` or remain in the current `State`. This State-Transition-Decision paradigm allows us to efficiently create a finite state machine for entities within Unity and create reusable states and transitions that can be used by different kinds of entities.

Our current design has several implications with respect to the overall system. We use `ScriptableObject` which has several key advantages and disadvantages. Advantages include asset serialization, object-oriented design and modularity of each component. One key disadvantage is that a `ScriptableObject` is not the physical representation of an object in the game, which is a `GameObject`. Therefore, we must have both `GameObject`s and `ScriptableObject`s and connect them. However, we cannot implement custom `GameObject`s within Unity that allow us to have the same polymorphic capabilities and object-oriented structure of a `ScriptableObject`. Since our goal is to build a game that is intuitively modular and extensible where designers and other developers can easily expand upon existing features, we believe that the advantages of using the `ScriptableObject` outweigh the disadvantages. We have crafted an example of a similar project that takes advantage of this design on our Wiki:
https://wiki.illinois.edu/wiki/display/cs428sp18/LUM%3A+Unity+Guidelines

5. Personal Reflections

Sheel Parekh (sdparek2) -

I believe that this project has taught me the importance of testing and team communication in Software Engineering.

Our focus has been to deliver multiple features each iteration and I believe that our process has allowed us to accomplish this. However, by putting our focus on implementing a multitude of features, we have often had features that were not completely tested. These issues persisted and affected us in later iterations and we had to focus on implementing many fixes and refactorings.

If we had reduced the scope of our project and focused on developing well-tested features, those issues would likely not have persisted. I believe that our team has performed well - although we have had slight communication barriers as some of our members are online students and some of us have busy schedules, we still managed to satisfy our requirements.

Will Song (wsong9) -

It turns out that I really disliked the way that Unity handled things, and during almost every iteration I had a huge urge to purge almost every Unity reference in our code base unless it was absolutely required. In the end, though, I got used to it and my tolerance levels increased significantly.

When reflecting upon the development process, I strongly feel that we almost certainly tried to get too much done every iteration. This led to a general decrease in code quality, which may have been one of the reasons why I disliked Unity so much. In the future, instead of having eight members working on four user stories, it may be a better idea to have six members working on two user stories and having two members planning out the framework for the next user stories, without actually doing them. When working on the next set of user stories, this allows related groups to use the pre-existing framework, assuming it works, without having to either write a framework that will not be used and delete it in the merge or wait until all their dependencies have the framework ready, merge related commits, and then starting the integration.

Patrick Millar (pmillar2) -

I think perhaps one of the largest things I've taken away from this is just all of what it takes as far as organization to keep a project going smoothly and the importance of strong leadership roles in getting things done.

Specifically there needs to be a lot of effort in maintaining communication and general documentation of what is expected (in no uncertain terms) and scheduling habits, as well as making sure deadlines are met and standards are followed. For best effect I believe these sorts of things need to be done by either a single person (leader) or done with a few stronger personalities among the group to make sure everyone stays on task and focused whenever necessary; and that giving structure to the structure-less can be the key to success in any project (not just software engineering)

Craig Swearingen (cswea3) -

This project has been challenging and interesting. It has been interesting because it is the biggest game project I personally have worked on. As for challenging, communication and lining up everyone's schedules was certainly a challenge. We really had to stay on top of communication in order to make sure we were getting everything done. Luckily we had a few strong leaders who were able to manage this, as well as the overall project. One thing I will take away from this project is that you can never have enough testing. It seems no matter how many test we write there are always more bugs to be found.

Wyatt Richter (wrichte2) -

Working on Lumiere throughout this semester has taught me lessons ranging from technical challenges like how to use Unity to organizational challenges like how to collaboratively work on a large codebase with eight people at the same time.

Unity proved to be very helpful after mastering the learning curve it presents. Unity has a very strict philosophy on how a game should be developed; adhere to that philosophy and your life will be easy but fight against it and you will waste development time. We had a little difficulty at first using Unity correctly, but after a few refactorings many of the following iterations went smoothly due to our code complying with Unity's prescribed development process. I'm glad we choose technologies that I did not have much experience with as learning Unity along with C# will be very helpful in my future development career.

Coding in a large group on the same codebase is an experience I never truly had before. It required that all of us utilized effective communication in order to clearly and concisely convey

our ideas to each other. It also required that we organize our code, branching, merging, and more as an unorganized coding structure could have been detrimental to our development process.

Lukasz Borowczak (lboro2) -

For me, this project taught me a lot about Unity, both conceptually and technically. I learned that there are ways to write code for Unity such that the code does most of the work and the engine just provides the graphics rendering, that Unity has a lot of stuff built in such as animation controllers, UI, and physics, that you can use Unity for 2D games without having to hack up the engine, and that Unity has built in support for testing.

I also learned a lot about working in a large group as an online student. Since there weren't too many meetings during the week and everyone talked on Slack, I didn't feel left out of the group just because I was an online student, and I managed to do pair programming pretty easily using free remote desktop control software like TeamViewer. Even during meetings, I could hear most of what was being said and could contribute to the conversation pretty easily.

Overall, I liked working on this project, and learned a lot about Unity, C#, and working in a group, even as an online student.

Rohan Kasiviswanathan (kasivis2) -

From working on Lumiere throughout the semester, I learned a lot about technical aspects such as coding in C# and unity in general, as well as the importance of planning carefully for group projects with regards to both teamwork and project structure.

Overall, Unity did not have too steep of a learning curve, and was a good choice for game development, though at times, it did present challenges. I also learned a lot about how to do game development through unity as well as write test cases, both of which are useful skills to have. Unity was also good in that it provided a lot of the base for you in various different aspects such as physics and graphics rendering.

With regards to teamwork, I learned a lot about the importance of planning carefully in general. The way we split and divided work throughout the semester was really good, though at times, we had did have trouble merging at various points. Following a specific process and structure did help us a lot, and is something that can be applied to other scenarios/projects as well.

Overall, it was a good project to work on. We could have improved in areas such as not trying to do too much for specific iterations or planning a little better for the future, but outside of that, we did well overall.

Scott Wolfskill (wolfski2) -

The Lumiere project greatly expanded my horizons. I was not new to game development, but I had never developed a game on a team of more than two before and had not worked with online members remotely on a long-term project. At first I was skeptical that we could effectively communicate and organize while not all of us could meet in-person for meetings and group work, but I was pleasantly surprised at how successful our efforts were with the online students. They were as committed to the project and pair programming as the rest of us, and that made it all possible.

This project taught me the importance of task scheduling and timely work, and that you get out of a team project what you put into it. When the team submitted work in a timely manner we had plenty of time to test, merge, and write documentation before a demo, while working at the last minute burned us on one occasion when our code base was not quite ready for the demo. Writing documentation helped us highlight our code functionality to other team members and would be a great aid to any interested in expanding our code base in the future.