Steven Anderson
Adam Cronin
Conor Doherty
Cormac McCarthy
Reid Olsen
Jason Owens
Daniel Sepler
COP4331
4/15/2015
Iteration 3 Design Documentation
Ooper Troopers

I affirm that the work submitted is my own and that the Honor Code was neither bent nor broken.

Learning Experiences

Steven Anderson

Creating this document gave me a great chance to use all of the knowledge I had gathered over the course of the semester to create a strong OOP design. This also gave me a chance to socialize with my new team and learn to work with all of them and see how they approach problems. I think this was a worthwhile assignment to solidify what we learned and to get used to our new teams. The only thing I would change is giving us less time to build the document to force us to get started and not procrastinate.

Adam Cronin

This assignment was challenging but very beneficial. There are a lot of OOP concepts that we needed to keep in mind when designing our game and I think that it was a bit intimidating. I thought that having a new group facilitated even more learning though because everyone had a completely different perspective than our last group. It took us a long time to finish the design document because we had a long discussion about pretty much every aspect of the game. In the end, I think that this iteration has been the most difficult.

Conor Doherty

Creating the design doc was a very interesting assignment. Now that we had to add some additional requirements from the third iteration, we had to think of some new ways to do things. What was really interesting about this assignment is that all of us had different ideas about what the design should be for different subsystems because of our prior experiences in our old groups. I think the hardest part was trying to come up with really good OOP solutions for the design that didn't just try to mimic our old group's designs.

Cormac McCarthy

This assignment led me to discuss every aspect of the project with my group and how to design it in such a way that followed the OOP principles we have learned in class thus far. Designing this document took us longer than expected, but we were able to talk about every little detail going into the project, creating a design that is flexible and allows for extension. We believe that the construction of this document will aid in our implementation of the project as it will serve as a reference when we have any questions. The hardest part of this assignment was ditching all of the code we had

previously written, along with the designs from before, and coming up with a brand new design that followed all OOP principles.

Reid Olsen
This design doc really forced us to think about how everything will work together in our project. The hardest part was coming up with ‘OOPy’ solutions. We went through several design changes, which will hopefully lead to an easier time coding since we have something to reference.

Jason Owens
Writing this design doc has led me to learn a lot about object oriented design in practice. We have changed a lot from how the project had been designed in previous iteration so I now much better understand what an OOP system is hoping to accomplish. (in previous iterations, it would have been very difficult to add new functionality)

Daniel Sepler
Writing this document has resulted in both clarity and confusion. On one hand, it opened the door for constructive conversations about class relationships / implementations. We have completely changed the way that the map as well as the tileable objects were structured, for example. Due to the doc, however, we often engaged in lengthy conversations that concluded in: “Well, I guess we’ll have to figure that out upon implementation.” Hopefully these conversations will expedite the coding process. It’s been quite the journey in iteration 3 so far, and I can’t wait to see where it will take us later!

VIEW SYSTEM

Table of Contents:

System Description
- Introductory Notes
- The Subsystems: An Overview
- Interacting with Model
- Foundational Classes
  - Class A: View
  - Class B: Viewport
  - Class C: Menu Viewport
  - Class D: Observer Viewport

Subsystem 1: The Crust
- Class A: Main Menu Screen
- Class B: Avatar Creation Screen
- Class C: LoadGame Screen
- Class D: Load/Save Screen
- Class E: Game Over Screen

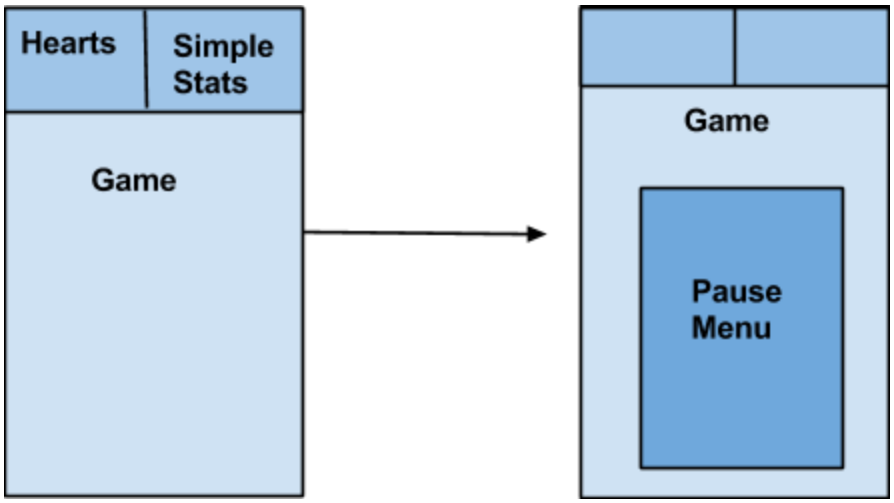
Subsystem 2: The Filling Subsystem
- Class A: Lives Left View
- Class B: Stats View
- Class C: Game Map View

Subsystem 3: The Whipped Cream Subsystem
- Subsystem 3.1: Whipped Menus
  - Class A: Save Game Screen
  - Class B: Pause View
  - Class C: TradingView
  - Class D: InventoryView
- Subsystem 3.2: Whipped Views
  - Class A: Controls Config View
  - Class B: Dialogue View

Introductory Notes

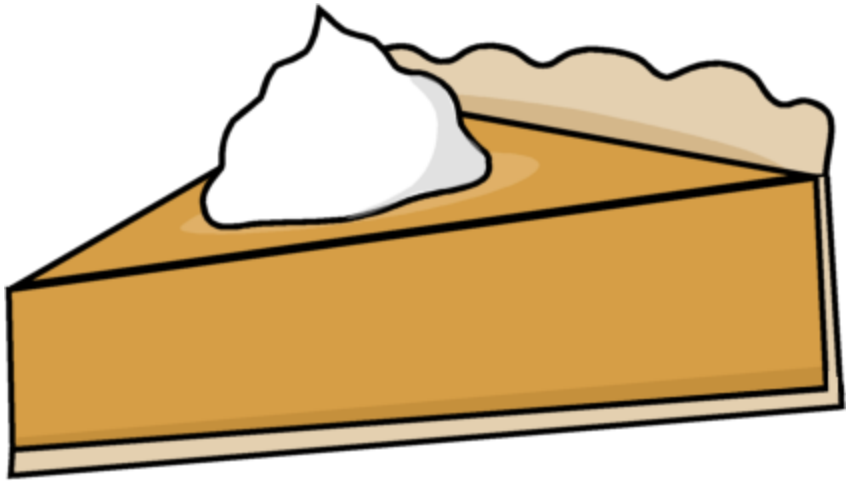
Our view will be split into three distinct subsystems. The distinction between them is based on the visual design of the game. The game's view is more or less a layering of its viewports on top of one another.

For example, during gameplay, the base layer of the View is the map. On top of that is a layer of simple statistics. Where the player to pause the game, the pause layer would appear as yet another layer on top of both the map and the statistics. Consult the following diagram for reference:



The Subsystems: An Overview

For reasons that will be explained, we encourage readers of our documentation to regard our view as a pie.



The first subsystem, named the *Crust Subsystem*, addresses all the foundational features of the game. These all occur outside of direct gameplay, and the subsystem is chiefly comprised of menus. These include introductory menu screens (the main menu, the avatar creation screen, and the load game screen), and the game over screen.

The second, the *Filling Subsystem*, covers the heart of the game -- anything that directly occurs in gameplay. This includes the game map, entity movement, items, projectiles, and the avatar.

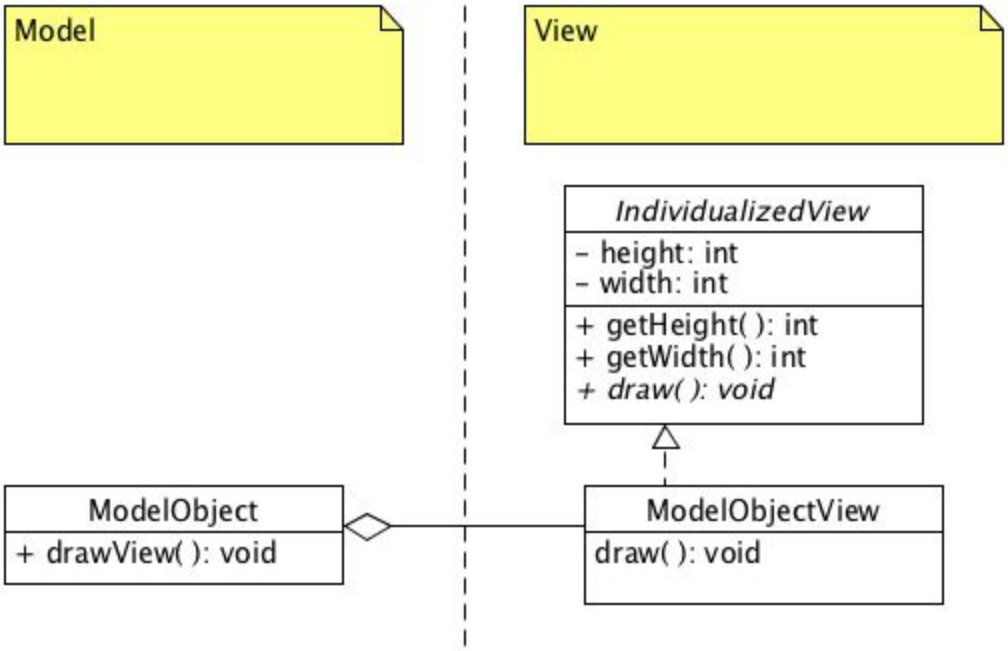
The third subsystem concerns all the views at the top layer of our game. These features, though not essential to basic map interaction, serve to offer tasty additions to an otherwise bland product. Thus, the *Whipped Cream Subsystem*. This subsystem covers views of such elements as Inventory, Configurable Controls, Saving, Dialogue, and Trading.

Before introducing the *Menu Subsystem*, we will introduce the “View” and “Viewport” systems, as they are important components of all three subsystems.

Interacting with Model

It is important to note that every single entity, tile, and <<tileable>> object has a corresponding view object. The model objects *know* their corresponding view objects as attributes, and upon any update of the model, the view's update operation is called. It was decided that conveying each of these classes in full-page format would be largely unhelpful as they are incredibly similar in composition to one another, and there are around fifty of them. Instead, we have decided to show examples of these individualized views, and list which objects will have them.

In essence, the view objects resemble….



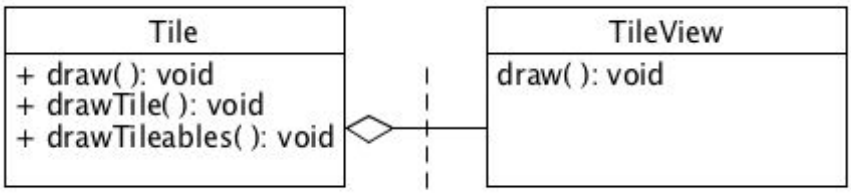
… where each such object has its own unique "draw" operation.

Responsibilities

Each of these views must understand what it looks like. It will not know whom it belongs to, nor will it know its location on the map. It will simply be told to update by the model, whenever necessary.

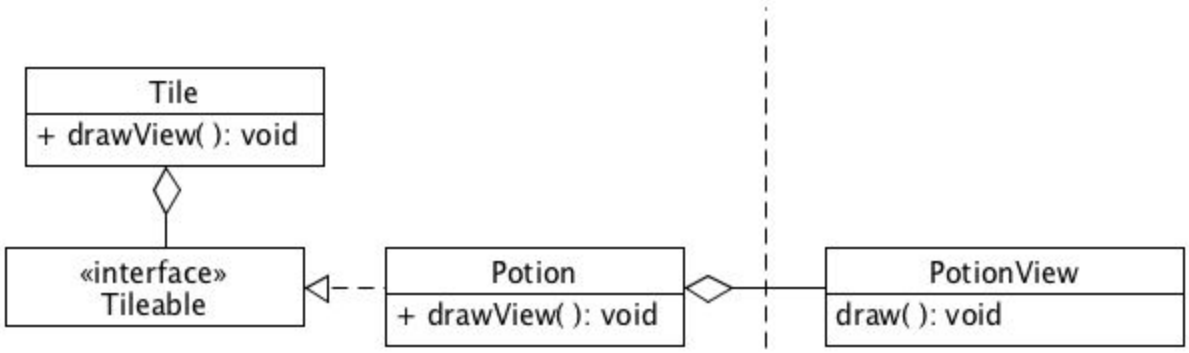
Examples

Tile views resemble the following....



Note that Tiles are aware of everything that's on them. This might include items, This includes Items, Entities, Decals, etc. Thus, the Tile has the ability to draw itself, its tileables, as well as both itself and its tileables (indicated by the *draw* operation).

A tileable view looks like.



Implementors

Danny Sepler, Jason Owens

Testers

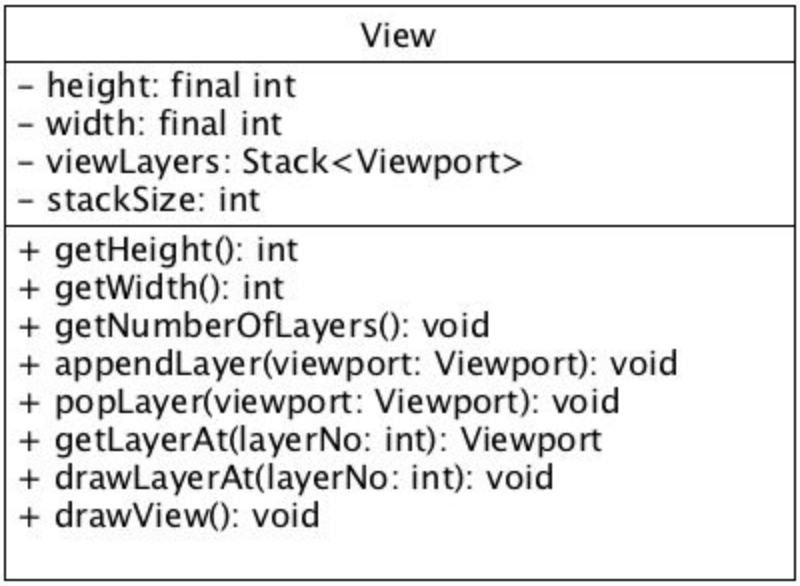
Danny Sepler

Class: View

Overview
The static class, View, is the foundation (the canvas, the table -- whatever analogy one may deem appropriate) of our visual. It is, at essence, a transparent container holding each layer of the visual (called “Viewports”) onto the screen. It will be instantiated inside of the RunGame class.

Responsibilities
To hold a stack of Viewports.

Collaborators
Works with: Viewport, RunGame



Implementors
Danny Sepler

Testers
Danny Sepler

Class: Viewport

Overview

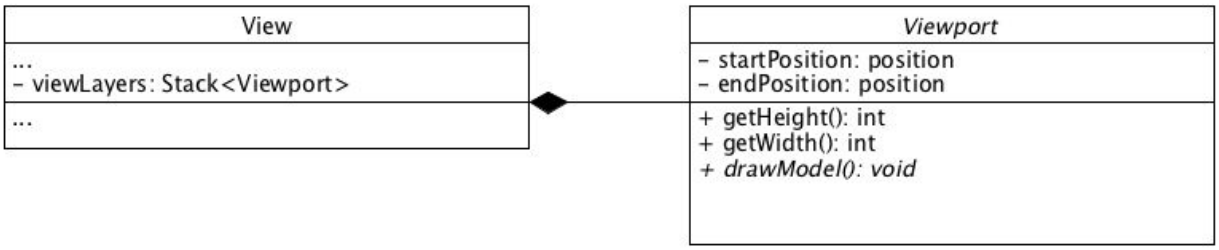
The Viewport is the abstract parent of any visual. It is instantiated using a StartPosition and EndPosition -- which means that all Viewports will be rectangles. Viewport also has the abstract operation *DrawModel( ),* which will be defined for every one of its children. All Viewports will be held on the View's stack.

Responsibilities

Provide default behaviors for all visuals.

Collaborators

Works with: View, Position,



Implementors

Danny Sepler

Testers

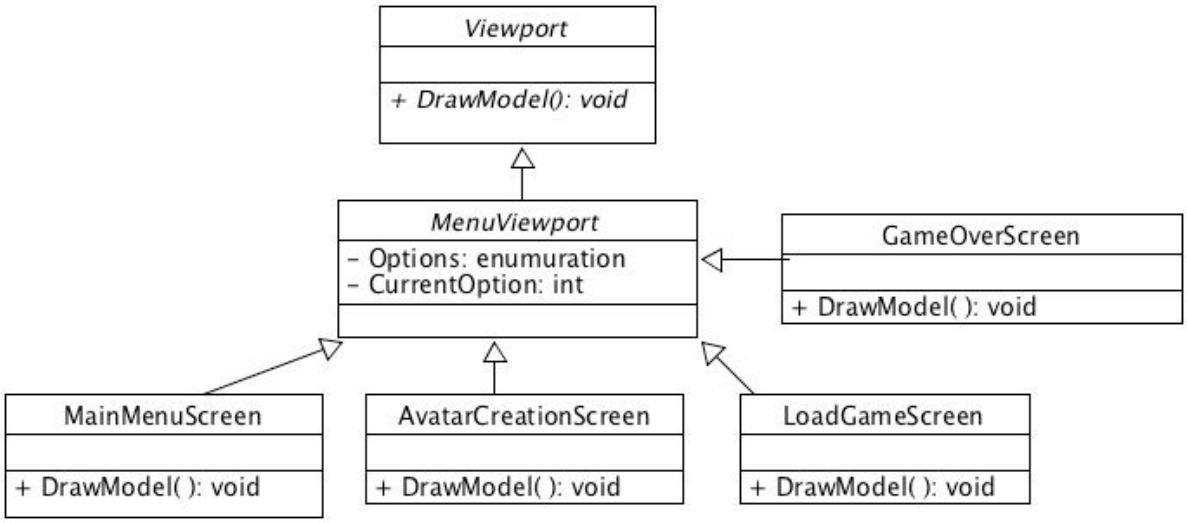
Danny Sepler

Class: MenuViewport

Overview
The MenuViewport is the abstract parent of all menus. It affects both the “Crust” and “Whipped Cream” subsystem.

Responsibilities
Present options to the user. Provide visual feedback on what the user selects/chooses.

Collaborators
Works with: Viewport



Implementors
Danny Sepler

Testers
Danny Sepler

Class: ObserverViewport

The abstract class for all viewports who have elements notified by changes in the map. These classes follow the observer pattern, who receive their notifications from elements in the model. Specifically, we are receiving notifications from changes in Statistics and the Tiles.

Responsibilities

Receive notifications from Notifiers. Redisplay the Viewports with the new information.

Collaborators

<<interface>> Notifier, Viewport, View, Statistics, Observer, StatsView, LivesLeftView, GameMapView, Tile.

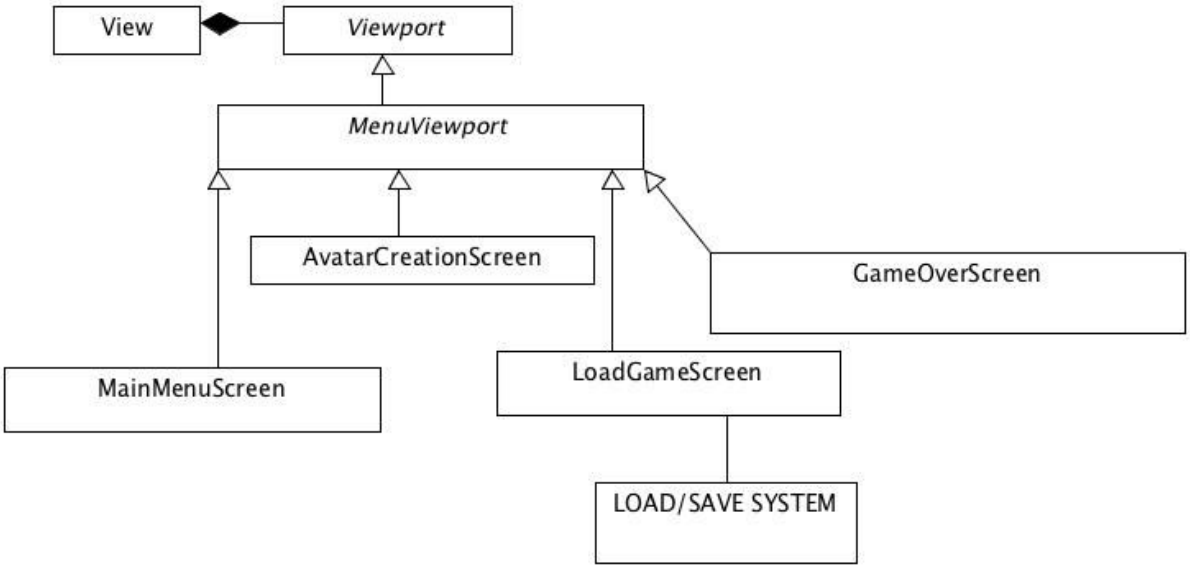
Implementors

Danny Sepler

Testers

Danny Sepler

SUBSYSTEM 1:
THE “CRUST” SUBSYSTEM



Contents

Class A: Main Menu Screen
Class B: Avatar Creation Screen
Class C: LoadGame Screen
Class D: Load/Save Screen
Class E: Game Over Screen

Class: MainMenuScreen

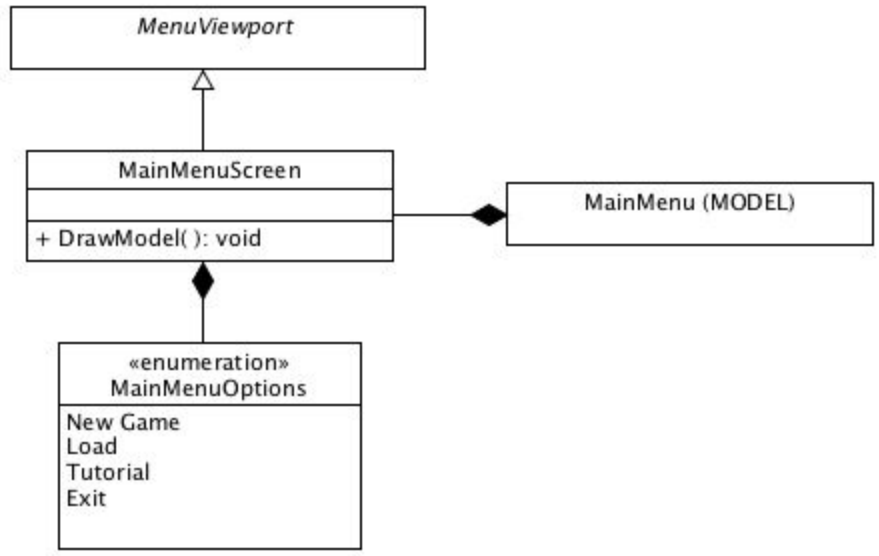
The MainMenuScreen is the initial screen that a player boots up. It offers the options of playing the tutorial, creating a new game, running a previously loaded game, and exiting the map. Its *DrawModel* operation is unique.

Responsibilities

Printing its options, visually portraying the user's input.

Collaborators

Should be an attribute of the model's *MainMenu* object. Has an enum for its options.



Implementors

Danny Sepler

Testers

Danny Sepler

Class: AvatarCreationScreen

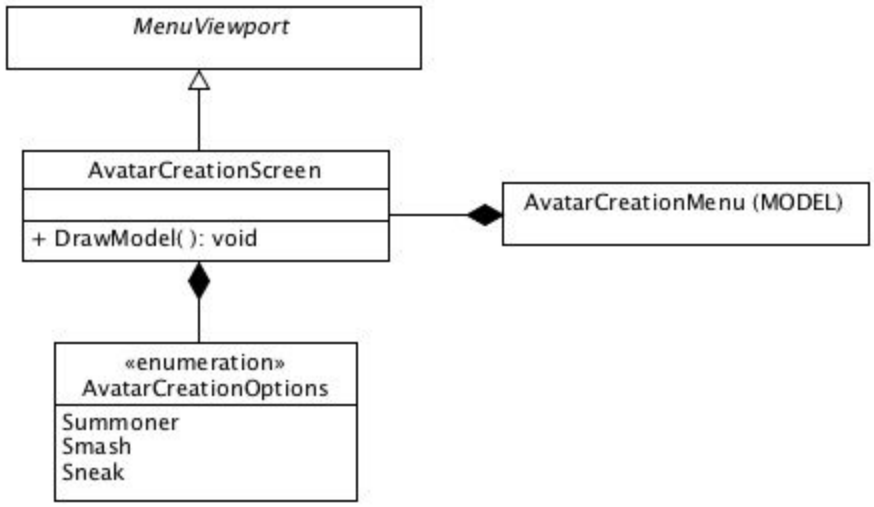
The AvatarCreationScreen offers the player the ability to choose his occupation, as well as the name of the Saved Game. Its *DrawModel* operation is unique.

Responsibilities

Printing its options, visually portraying the user's input.

Collaborators

Should be an attribute of the model's *AvatarCreationMenu* object. Has an enum for its options.



Implementors

Danny Sepler

Testers

Danny Sepler

Class: LoadGameScreen

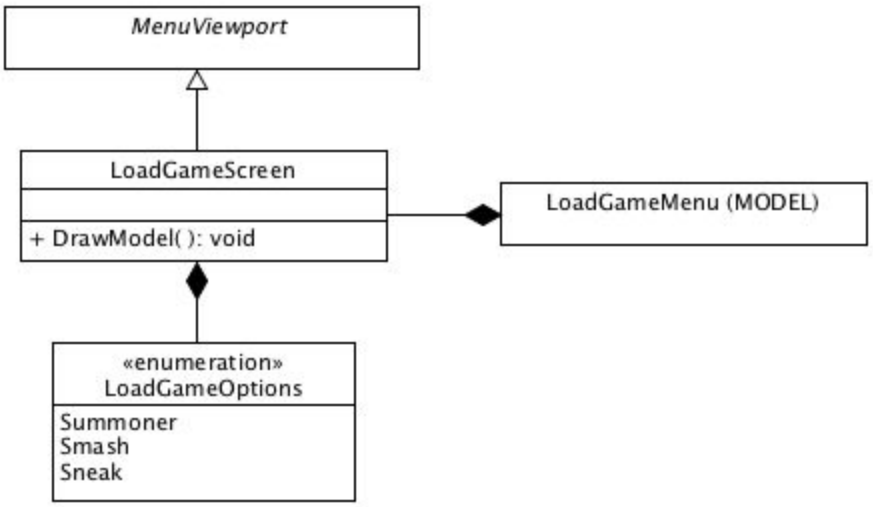
The LoadGameScreen shows the user all the previously saved games. Its *DrawModel* operation is unique.

Responsibilities

Printing saved games, visually portraying user's input.

Collaborators

Should be an attribute of the model's *LoadGameMenu* object. Has an enum for its options.



Implementors

Danny Sepler

Testers

Danny Sepler

Class: LoadGameScreen

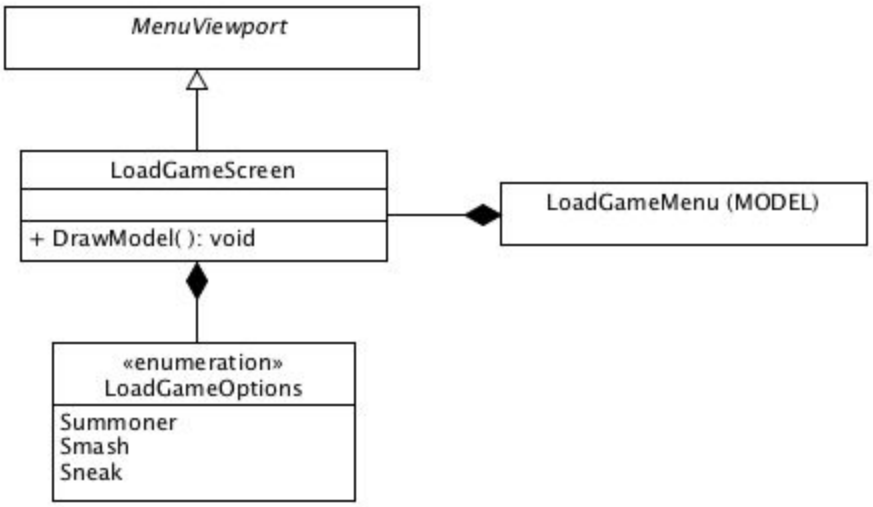
The LoadGameScreen shows the user all the previously saved games. Its *DrawModel* operation is unique.

Responsibilities

Printing saved game names, visually portraying user's input.

Collaborators

Should be an attribute of the model's *LoadGameMenu* object. Has an enum for its options.



Implementors

Danny Sepler

Testers

Danny Sepler

Class: GameOverScreen

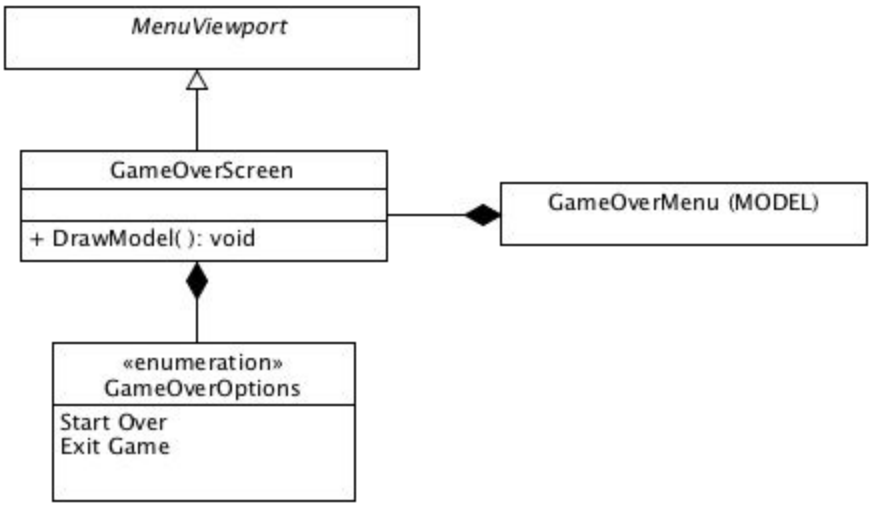
The GameOverScreen shows the user two options: start over, or exit game. Its *DrawModel* operation is unique.

Responsibilities

Printing options, visually portraying user's input.

Collaborators

Should be an attribute of the model's *GameOverMenu* object. Has an enum for its options.



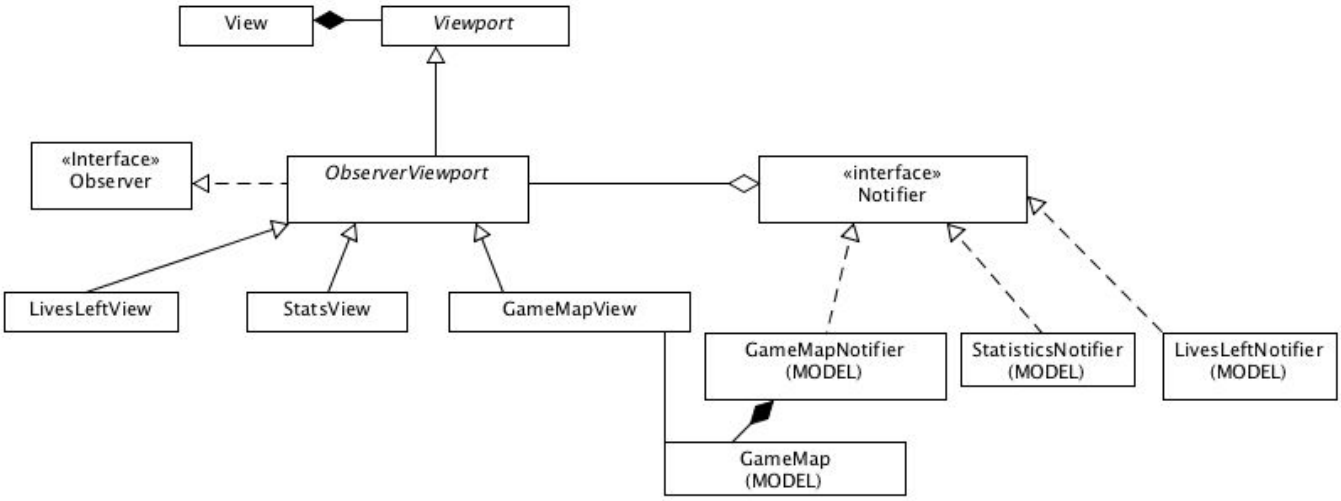
Implementors

Danny Sepler

Testers

Danny Sepler

SUBSYSTEM 2:
ACTION SUBSYSTEM



Contents:

Class A: ObserverView [located in “Foundational Classes” at top of System]
Class B: LivesLeftView
Class C: ActiveGameView
Class D: StatsView

Class: LivesLeftView

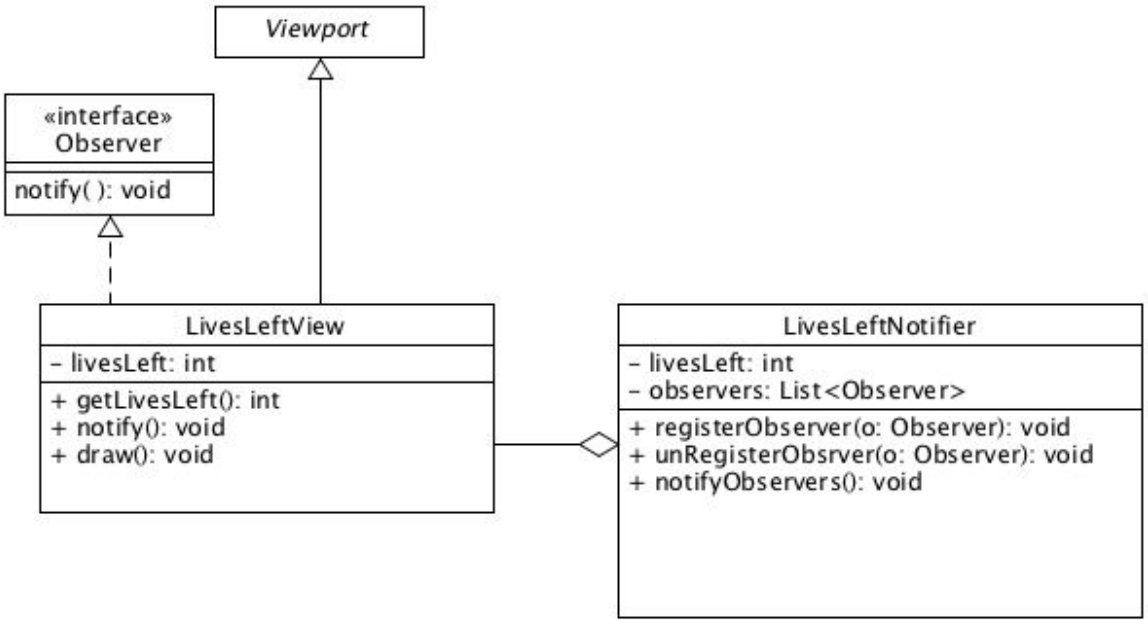
A layer on top of the ActiveGameView, which shows how many lives left (through heart decals) that the Avatar has.

Responsibilities

Receives the amount of LivesLeft stat from Avatar. Updates when necessary.

Collaborators

LivesLeftNotifier, Observer, Viewport, View



Implementors

Danny Sepler

Testers

Danny Sepler

Class: ActiveGameView

The ActiveGameView is the visual representation of the Tiles and all interactions that happen on it. Objects included by this umbrella “interaction” concept include entity movement (including that of the avatar), terrains, tiles, items, and battles.

There is an important detail to note as far as the tiles being sent to the ActiveGameView. All tiles being displayed are MemTiles (“Memory Tiles”) that are part of the Avatar’s “Journal”. This decision was originally conceived as a manner of making the transition from regular view to scrollable view as seamless as possible. The decision was made after two more details came to light:

1. All objects on the game map within the the avatar’s immediate surroundings are accurate in her Journal (because their changes are automatically refreshed).

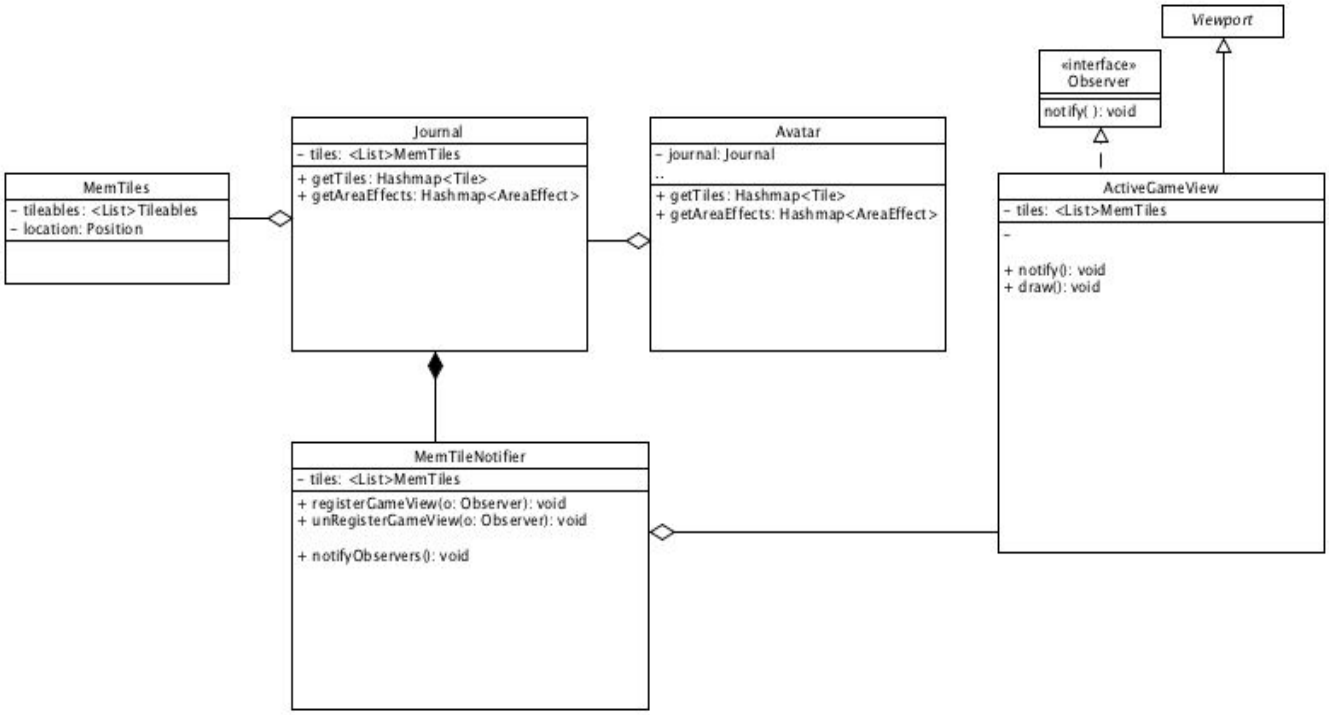
2. MemTiles are simpler than regular Tiles. They do not need to understand many interactions. They do not need logical functionality. Et cetera, et cetera.

Responsibilities

Displaying the gameplay map to the user. Updating upon any change in the map.

Collaborators

Tile, Observer, View, Avatar, MemTile, <<interface>> Tileable



Implementors
Danny Sepler, Jason Owens

Testers
Danny Sepler

Class: StatsView

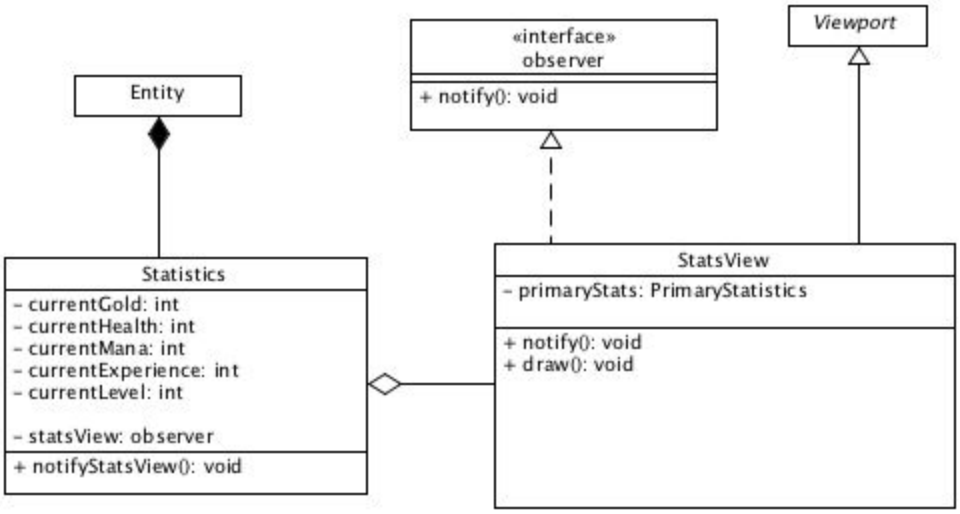
A layer on top of the ActiveGameView, which shows some of the basic stats of the Avatar. These include lifeAmount, Strength, Agility, Intellect, Hardiness, Experience, and Movement.

Responsibilities

Receives the stats from the Avatar's. Updates when necessary.

Collaborators

Statistics, Entity, Observer, Viewport, View



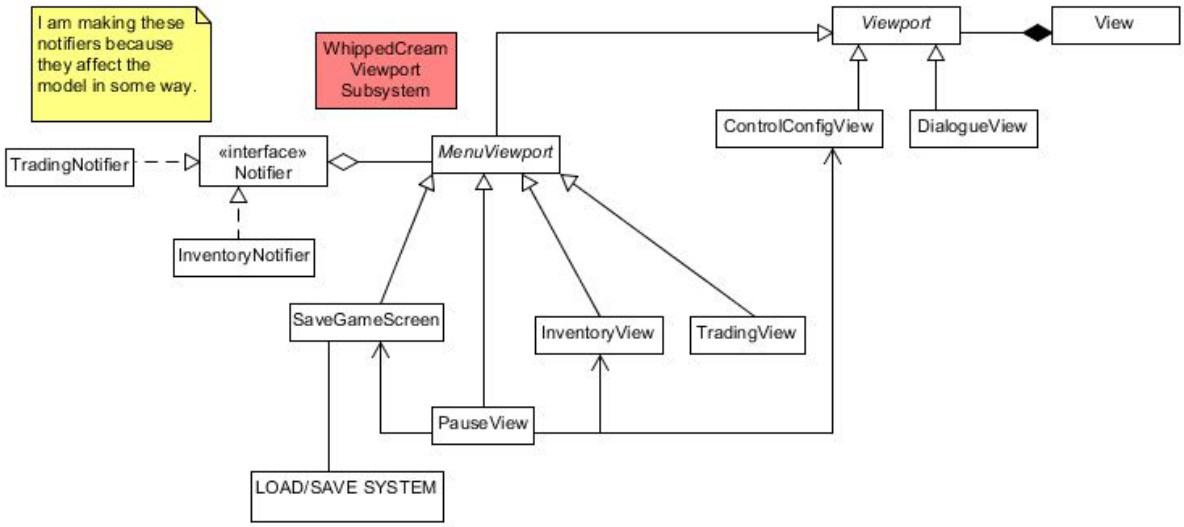
Implementors

Danny Sepler

Testers

Danny Sepler

SUBSYSTEM 3:
THE “WHIPPED CREAM” SUBSYSTEM



Contents:

Subsystem 1: Whipped Menus
- Class A: PauseView
- Class B: SaveGameScreen
- Class C: InventoryView
- Class D: TradingView

Subsystem 2: Whipped Views
- Class A: ControlsConfigView
- Class B: DialogueView

Class: PauseView

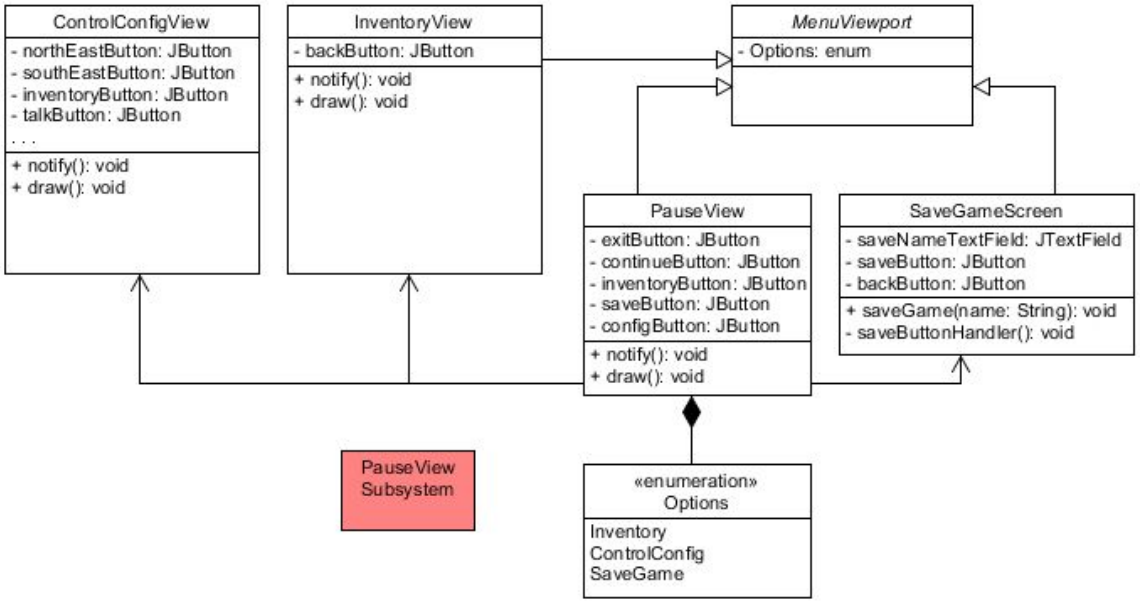
A pop-up style pause screen centered on top of the game screen when the game is paused. The pause menu should be front and center. The pause menu is supposed to house buttons to navigate to different screens, such as the SaveGameScreen.

Responsibilities

The hub of in-game menus. Mainly a navigation point to other pop-ups for inventory, saving, etc. The pausing of the game state should not be done here.

Collaborators

InventoryView, ControlConfigView, SaveGameScreen, Options, MenuViewport



Implementors & Testers

Conor Doherty

Class: SaveGameScreen

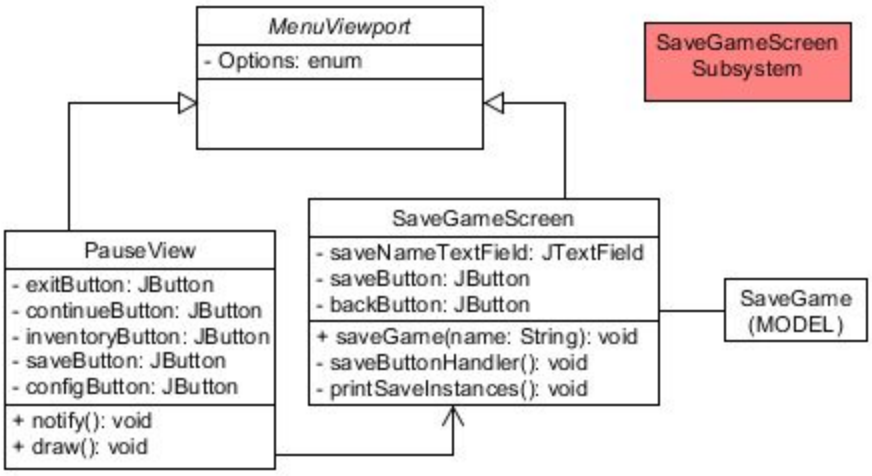
The save game screen should be the user interface for the save utility.

Responsibilities

Provide a nice user interface for saving the current game state. All previous game states should be presented, as well as a means to make new save states.

Collaborators

MenuViewport, PauseView, SaveGame (MODEL)



Implementors & Testers

Conor Doherty

Class: InventoryView

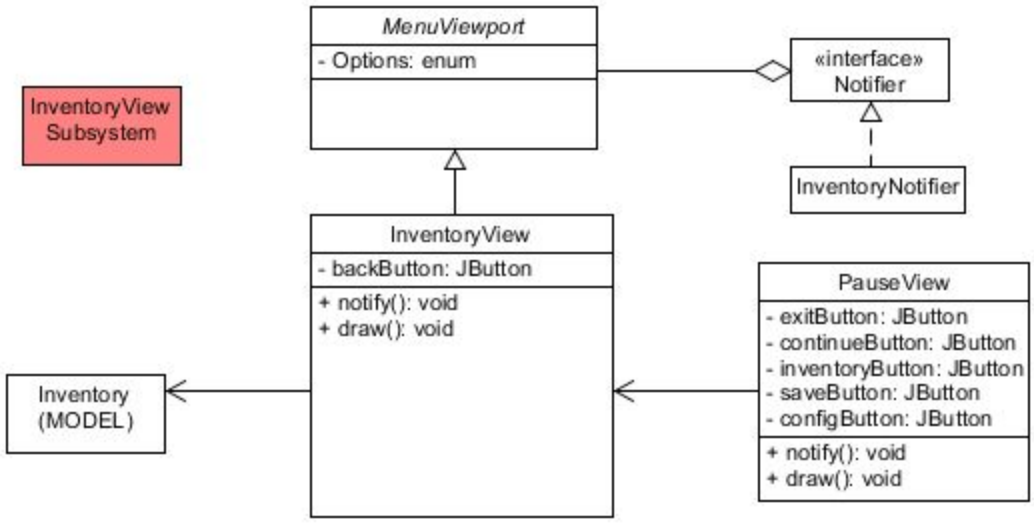
The inventory view should allow the user to see their whole inventory

Responsibilities

Provide a nice user interface for managing the inventory of the Avatar. Both the equipped inventory and the inventory should be present and it should be simple for the user to equip/unequip/drop items easily.

Collaborators

MenuViewport, PauseView, Notifier, InventoryNotifier, Inventory (MODEL)



Implementors & Testers

Conor Doherty

Class: TradingView

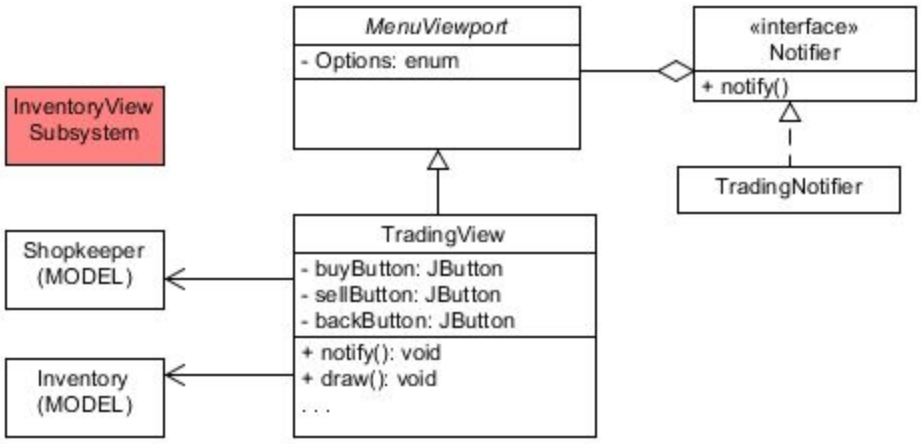
Want that shiny new sword everyone is talking about? We have you covered. The “TradingView” class displays to the user all necessary functionality for bartering with other entities in the game.

Responsibilities

Display inventory of non-user entity. (Probably shopkeeper). Display price of each item in inventory. Show user’s amount of gold. Have visual for the purchase of an item.

Collaborators

MenuViewport, Notifier, TradingNotifier, Shopkeeper, Inventory



Implementors & Testers

Danny Sepler, Conor Doherty

Class: ControlConfigView

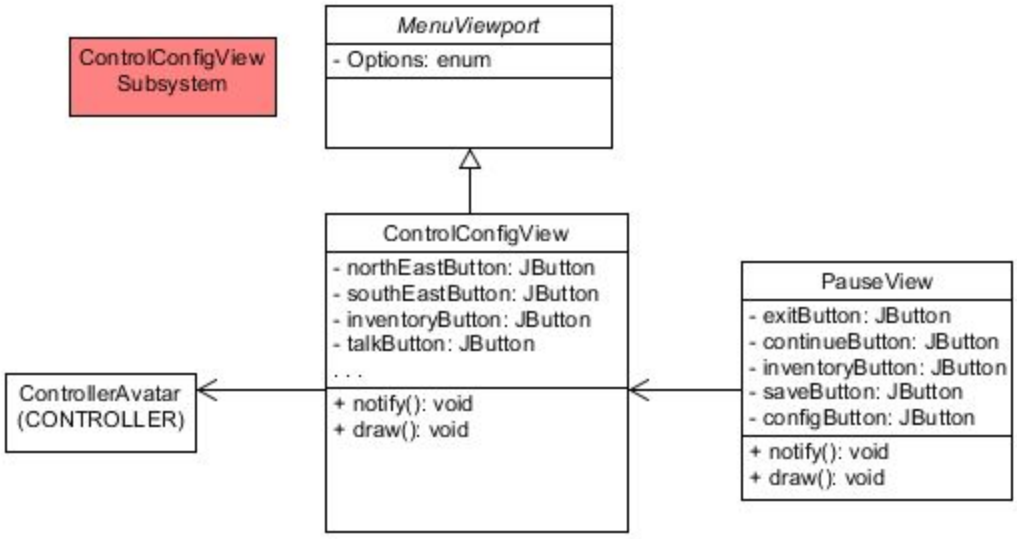
Should allow users to customize their controls and display current control settings.

Responsibilities

Have a nice user-interface for users to interactively manage the game controls. Any customized controls will notify the controller what needs to be changed. The Controller will deal with the re-bindings.

Collaborators

MenuViewport, PauseView, ControllerAvatar



Implementors & Testers

Conor Doherty

Class: DialogueView

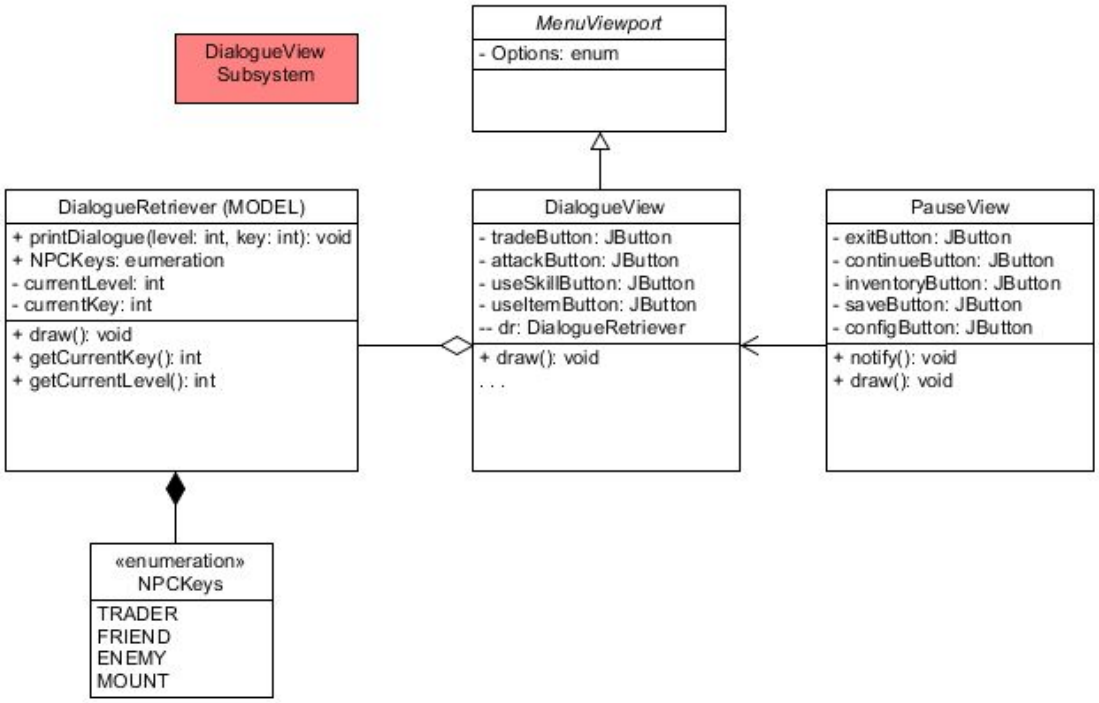
Allows the user to learn from and interact with other entities (primarily NPCs) inside the game.

Responsibilities

Take in text from other entities. Display it. Have the ability to appear when important(/recent) and disappear when not.

Collaborators

MenuViewport, PauseView, DialogueRetriever, NPCKeys



Implementors & Testers

Danny Sepler, Conor Doherty

Controller System

Table of Contents

Subsystem 1: Controller Avatar
    Class A: ControllerAvatar
    Class B: KeyListener
Subsystem 2: Run Game
    Class A: RunGame
Subsystem 3: Mouse Listener
    Class A: MouseListener

SUBSYSTEM 1:
CONTROLLER AVATAR SUBSYSTEM

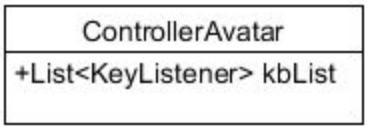
Class: ControllerAvatar

Overview
The Controller aspect of Avatar that will hold the KeyBindings set linked to that Avatar.

Responsibilities
It will hold the set of Keybindings for that specific Avatar.

Collaborators
KeyListener

UML Diagram Here



Implementors
Adam Cronin

Testers
Adam Cronin

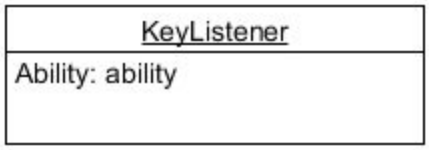
Class: KeyListener

Overview
A class that listens for key events such as a key being pressed and released.

Responsibilities
Will listen for key events and will alert the controller when such events occur.

Collaborators
RunGame

UML Diagram Here



Implementors
Adam Cronin

Testers
Adam Cronin

SUBSYSTEM 2:
RUN GAME SUBSYSTEM

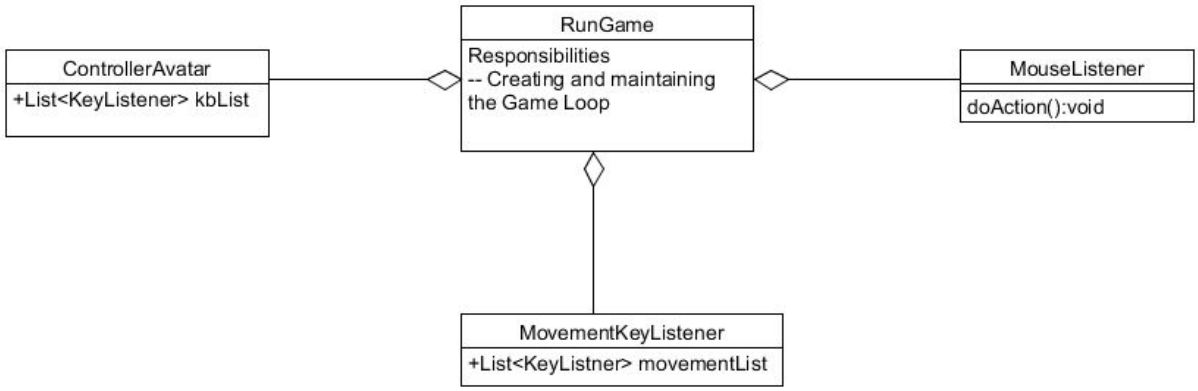
Class: RunGame

Overview
The class that starts the Controller and entire game.

Responsibilities
Creating and maintaining the Game Loop as well as having a Tile, MouseListener, and ControllerAvatar used to control the game.

Collaborators
ControllerAvatar, MouseListener, Tile

UML Diagram Here



Implementors
Adam Cronin

Testers
Adam Cronin

SUBSYSTEM 3:
MOUSE LISTENER SUBSYSTEM

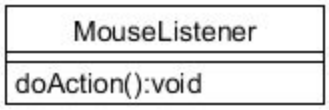
Class: MouseListener

Overview
A class that listens for mouse events such as clicking and releasing.

Responsibilities
Will listen for mouse click events and will alert the controller when such events occur.

Collaborators
RunGame

UML Diagram Here



Implementors
Adam Cronin

Testers
Adam Cronin

Model System

Subsystems

- Subsystem 1: Entity
    - Class A: Entity
    - Class B: Avatar
    - Class C: NPC
    - Class E: Shopkeeper
    - Class G: Mount
    - Class H: Pet
- Subsystem 2: Occupation
    - Class A: Occupation
    - Class B: SmasherOccupation
    - Class C: SneakOccupation
    - Class D: SummonerOccupation
    - Class E: MountOccupation
- Subsystem 3: Statistics
    - Class A: Statistics
    - Class B: SmasherStatistics
    - Class C: SummonerStatistics
    - Class D: SneakStatistics
    - Class E: MountStatistics
    - Class F: PrimaryStatistics
    - Class G: DerivedStatistics
    - Class H: EquippableStatistics
- Subsystem 4: Items
    - Class A: Item
    - Subsystem 4.1: Takeable Items
        - Class A: Takeable Item
        - Subsystem 4.1.1: Equippable Items
            - Class A: Equippable
            - Class B: Weapon
            - Class C: Ranged
            - Class D: OneHanded
            - Class E: TwoHanded
            - Class F: Brawl
            - Class G: Staff
            - Class H: Armor

                - Class I: Helmet
                - Class J: Chest
                - Class K: Arms
                - Class L: Legs
                - Class M: Shield
                - Class N: Saddle
            - Subsystem 4.1.2: Usable Items
                - Class A: Usable
                - Class B: Potion
        - Subsystem 4.2: One-Shot Items
            - Class A: OneShotItem
            - Class B: HealingOneShotItem
            - Class C: DamagingOneShotItem
        - Subsystem 4.3: Interactive Items
            - Class A: InteractiveItem
        - Subsystem 4.4: Obstacles
            - Class A: Obstacle
- Subsystem 5: Inventory
    - Class A: Inventory
- Subsystem 6: Equippable
    - Class A: EquipmentManager
    - Class B: SmasherEquipmentManager
    - Class C: SneakEquipmentManager
    - Class D: SummonerEquipmentManager
    - Class E: MountEquipmentManager
- Subsystem 8: Map
    - Class A: Location
    - Class B: Tile
    - Class C: MemTile
    - Class D: Journal
    - Class E: Tileable
    - Class F: GameTimer
    - Class G: MovementRequirments
    - Class H: MovementCapabilities
    - Class I: AreaEffect
    - Class J: HealAreaEffect
    - Class K: DealDamageAreaEffect
    - Class L: InstantDeathAreaEffect
    - Class M: LevelUpAreaEffect

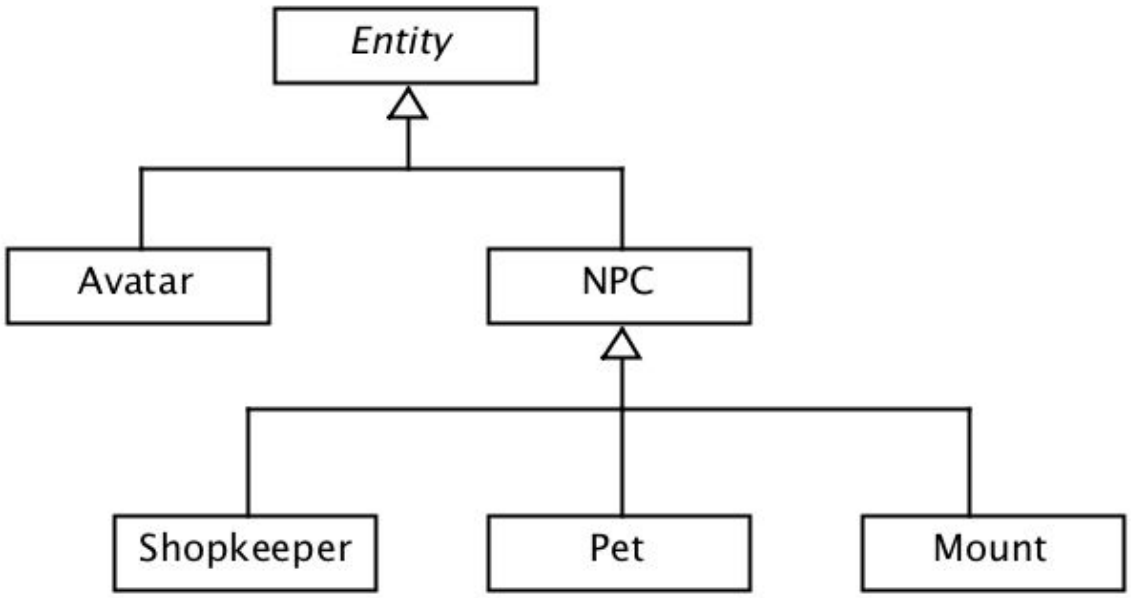
        - Class O: ForceTile
        - Class P: Terrain
        - Class Q: GrassTerrain
        - Class R: WaterTerrain
        - Class S: MountainTerrain
- Subsystem 9: Effect
    - Class A: Effect
    - Class B: AbstractEffect
    - Class C: ConcreteEffect
- Subsystem 10: Load/Save Game
    - Class A: LoadGame
    - Class B: SaveGame
- Subsystem 11: Skills/Abilities
    - Class A: Ability
    - Class B: BindWounds
    - Class C: Bargain
    - Class D: Observation
    - Subsystem 11.1: SmashAbility
        - Class A: SmasherAbility
        - Class B: OneHandedWeaponAbility
        - Class C: TwoHandedWeaponAbility
        - Class D: BrawlAbility
    - Subsystem 11.2: SneakAbility
        - Class A: SneakAbility
        - Class B: Pickpocket
        - Class C: TrapAwareness
        - Class D: Creep
        - Class E: RangedAbility
    - Subsystem 11.3: SummonerAbility
        - Class A: SummonerAbility
        - Class B: Enchantment
        - Class C: Boon
        - Class D: Bane
        - Class E: StaffAbility

Subsystem: Entity

Overview

The Entity subsystem is the hierarchy of different characters including both Player and Non-Player Characters. This subsystem is needed to meet the project requirements for Entity.

UML Diagrams



Class: Entity

Overview

An Entity is the supertype of the Entity hierarchy. This class is necessary to allow all of the different player and non-player characters to be able to share common elements, such as Statistics and the ability to be moved. This means Entity encapsulates the abstraction of a generic character. This helps meet the project requirements for multiple character types.

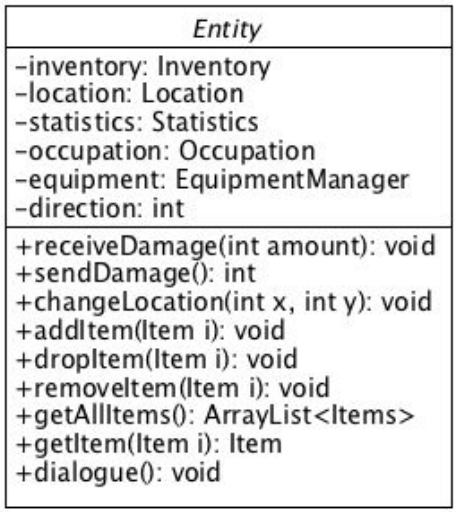
Responsibilities

To hold attributes and operations that are common among both player and non-player characters, including but not limited to location, Statistics, and Inventory. To die when HP reaches 0.

Collaborators

Works with: Statistics, Occupation, Inventory, Items, EquipmentManager

UML Diagram Here



Implementors

Steven Anderson

Testers

Steven Anderson

Class: Avatar

Overview

An Avatar is a subtype of the Entity type. This class is necessary to differentiate between player and non-player characters. This class helps construct a player character and includes features such as respawning and using vehicles. This class meets the project requirements for a user controlled character. Mounting will be handled in the Mount class.

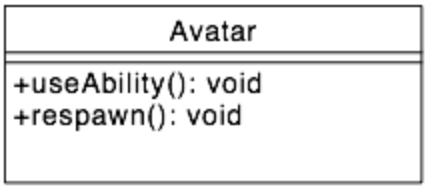
Responsibilities

To be capable able to use a mount, respawning upon death, and be used by the Player

Collaborators

Works with controllers

UML Diagram Here



Implementors

Steven Anderson

Testers

Steven Anderson

Class: NPC

Overview

An NPC is a subtype of the Entity type. This class is necessary to differentiate between player and non-player characters. This class helps clarify the attributes unique to non-player characters such as player tracking and knowing what entities to have aggro against.

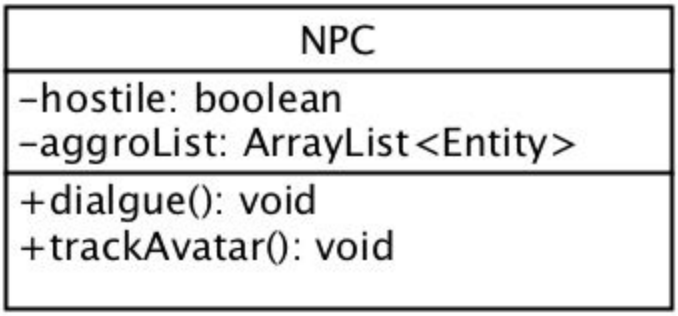
Responsibilities

To track an Avatar class when within range, to properly interact with avatar based on if hostile or not

Collaborators

Works with Avatar

UML Diagram Here



Implementors

Steven Anderson

Testers

Steven Anderson

Class: Shopkeeper

Overview

A Shopkeeper is a subtype of the NPC. This class is necessary to differentiate between ordinary NPCs and those that you can engage in commerce with. This class helps clarify the operations unique to shopkeepers, such as buying and selling items. This is to prevent mixed-instance cohesion.

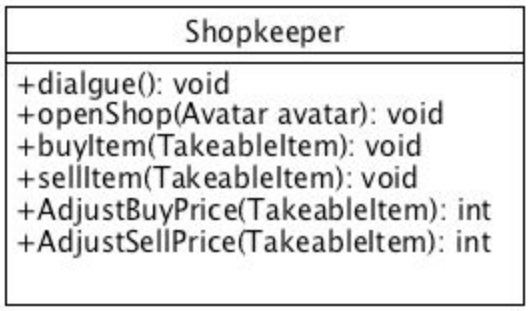
Responsibilities

To handle money and item transfers when an Avatar tries to trade.

Collaborators

Works with Avatar

UML Diagram Here



Implementors

Steven Anderson

Testers

Steven Anderson

Class: Mount

Overview

A Mount is a subtype of the NPC. This class' occupation will be the 'Mount' occupation. This class helps clarify the operations unique to Mount. An Avatar type should be able to mount a Mount entity and receive some sort of movement bonus.

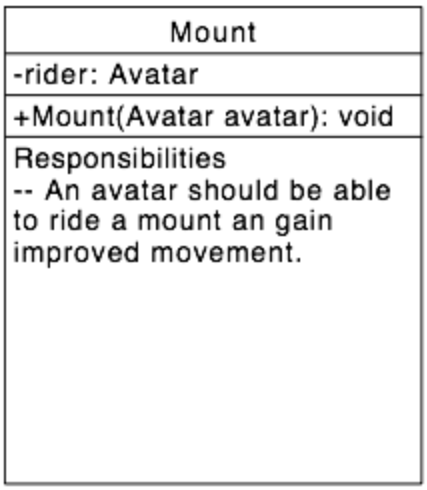
Responsibilities

Mount should be able to be ridden by an Avatar.

Collaborators

Works with Avatar

UML Diagram Here



Implementors

Reid Olsen

Testers

Reid Olsen

Class: Pet

Overview

A Pet is a subtype of the NPC. This NPC will follow the player's Avatar around on the map. It will have an occupation and will be able to cast the appropriate spells.

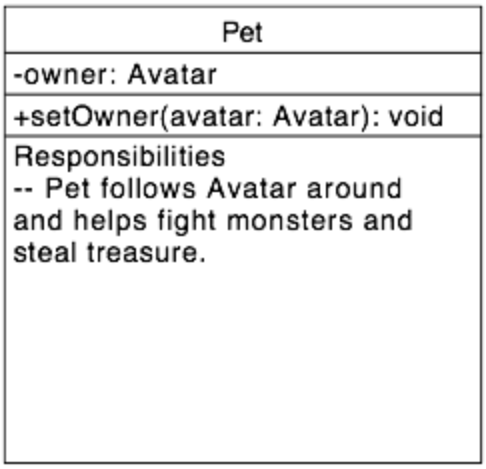
Responsibilities

Pets will follow the Avatar around on the map and cast spells and steal treasure.

Collaborators

Works with Avatar

UML Diagram Here



Implementors

Reid Olsen

Testers

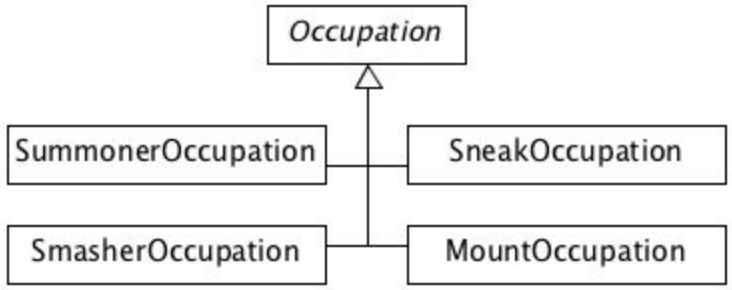
Reid Olsen

Subsystem: Occupation

Overview

The Occupation subsystem is the set of the different occupations a character can have. This subsystem is needed to meet the project requirements for Occupation and allows an Entity to be a Smasher, Sneak, Summoner, or Mount. This system interacts with Entity and Skill Manager, and helps to mediate the interactions between those two subsystems.

UML Diagrams



Class: Occupation

Overview

This class is necessary to help create the abstract concept of an Occupation. This class will hold the basic information that is the same among each of the separate occupation types, such as the skill manager for the base skills (Observation, Bind Wounds, Bargain). This class is the supertype of the Occupation hierarchy and helps determine what equipment an Avatar can equip.

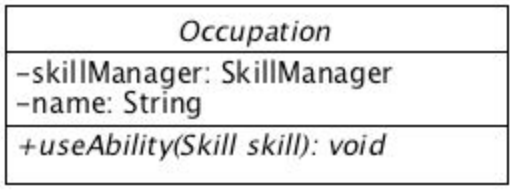
Responsibilities

To hold the Skill Manager for an Entity, and to mediate the interactions between the Skills and the Entity. To tell the Avatar what it can equip.

Collaborators

Works with Entity, Skill Manager, Skill

UML Diagram Here



Implementors

Steven Anderson

Testers

Steven Anderson

Class: SmasherOccupation

Overview

This class is the subtype of the Occupation that is specific to the Smasher type entities. This class would hold the expanded Skill Manager that holds all of the Smasher specific skills and the base skills. This class is necessary to have a class that properly abstracts the concept of a Smasher.

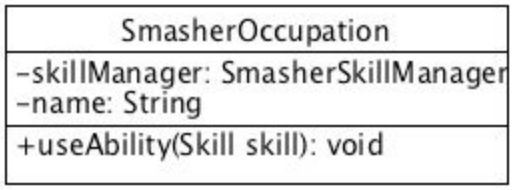
Responsibilities

To hold the Skill Manager with Smasher specific skills. To assist in equipping Smasher equipment.

Collaborators

Works with Entity, Smasher Skill Manager, Skill

UML Diagram Here



Implementors

Steven Anderson

Testers

Steven Anderson

Class: SneakOccupation

Overview

This class is the subtype of the Occupation that is specific to the Sneak type entities. This class would hold the expanded Skill Manager that holds all of the Sneak specific skills and the base skills. This class is necessary to have a class that properly abstracts the concept of a Sneak.

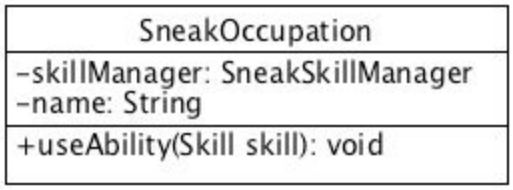
Responsibilities

To hold the Skill Manager with Sneak specific skills. To assist in equipping Sneak equipment.

Collaborators

Works with Entity, Sneak Skill Manager, Skill

UML Diagram Here



Implementors

Steven Anderson

Testers

Steven Anderson

Class: SummonerOccupation

Overview

This class is the subtype of the Occupation that is specific to the Summoner type entities. This class would hold the expanded Skill Manager that holds all of the Summoner specific skills and the base skills. This class is necessary to have a class that properly abstracts the concept of a Summoner.

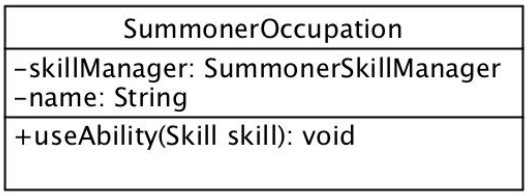
Responsibilities

To hold the Skill Manager with Summoner specific skills. To assist in equipping Summoner equipment.

Collaborators

Works with Entity, Summoner Skill Manager, Skill

UML Diagram Here



Implementors

Steven Anderson

Testers

Steven Anderson

Class: MountOccupation

Overview

This class is the subtype of the Occupation that is specific to the Mount class. This class would help clarify the attributes unique to a Mount, as all Entities have an Occupation. This would give a skills boost to the skills a Mount could provide, such as improved Observation.

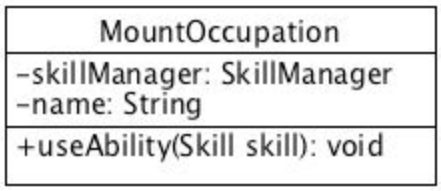
Responsibilities

To improve the Skills of the combined Mount and Avatar

Collaborators

Works with Skill Manager, Skill, Mount

UML Diagram Here



Implementors

Steven Anderson

Testers

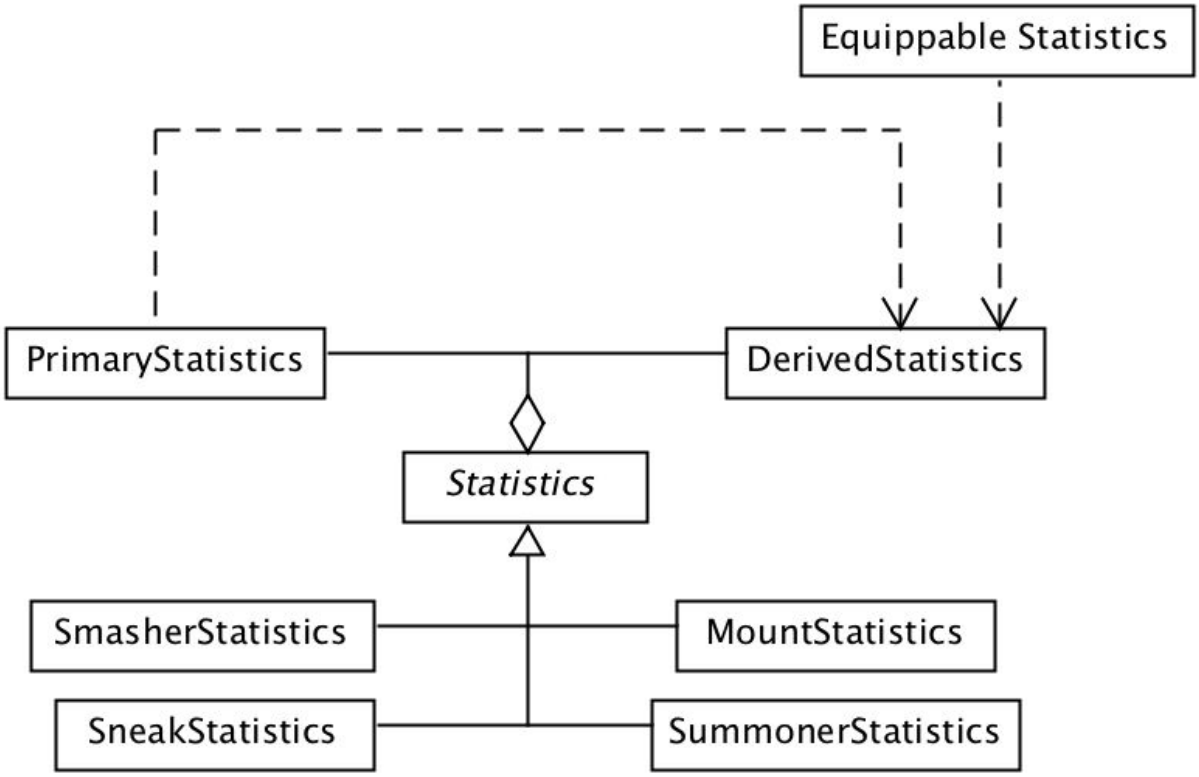
Steven Anderson

Subsystem: Statistics

Overview

This system tracks how the several different Statistics classes interact with each other. Primary Statistics and Equippable Statistics both influence Derived Statistics and all three statistics are packaged into the Statistics class to help improve system wide cohesion. This system meets the requirements of Statistics found in the project requirements. This system also influences multiple other systems, such as TakeableItem and Entity.

UML Diagrams



Class: Statistics

Overview

This class acts as a container class to hold the other three Statistics classes and to include several Entity unique statistics. This helps improve Entity internal cohesion by removing various mixed anti-cohesions. This helps meet the Statistics requirements and improves the cohesion of the Statistics subsystem.

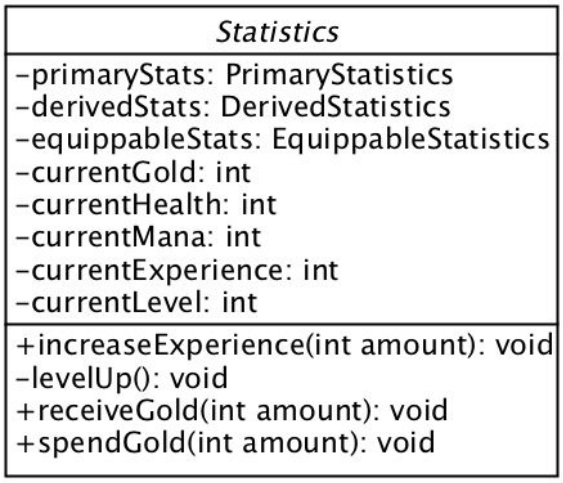
Responsibilities

To facilitate the interactions of the three other Statistics classes. To mediate the interactions of Statistics classes and Entity classes. To store Entity specific classes.

Collaborators

Works with Entity, Derived Stats, Primary Stats

UML Diagram Here



Implementors

Steven Anderson

Testers

Steven Anderson

Class: SmasherStatistics

Overview

This class acts as a subtype of Statistics that overrides several features to make the attributes and operations more appropriate for a Smasher. It also overrides the level up to better meet the needs of a Smasher

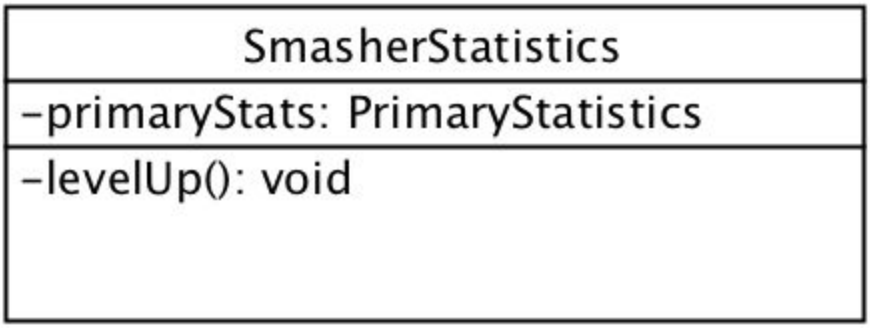
Responsibilities

To create the proper set of Statistics and level up improvements for a Smasher

Collaborators

Works with Entity, Derived Stats, Primary Stats

UML Diagram Here



Implementors

Steven Anderson

Testers

Steven Anderson

Class: SneakStatistics

Overview

This class acts as a subtype of Statistics that overrides several features to make the attributes and operations more appropriate for a Sneak. It also overrides the level up to better meet the needs of a Sneak

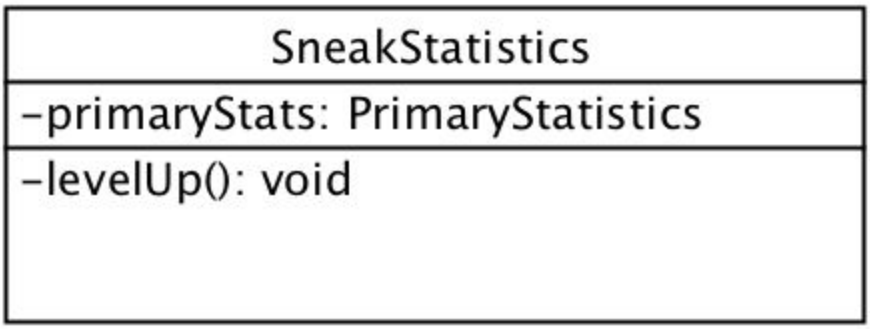
Responsibilities

To create the proper set of Statistics and level up improvements for a Sneak

Collaborators

Works with Entity, Derived Stats, Primary Stats

UML Diagram Here



Implementors

Steven Anderson

Testers

Steven Anderson

Class: SummonerStatistics

Overview

This class acts as a subtype of Statistics that overrides several features to make the attributes and operations more appropriate for a Summoner. It also overrides the level up to better meet the needs of a Summoner

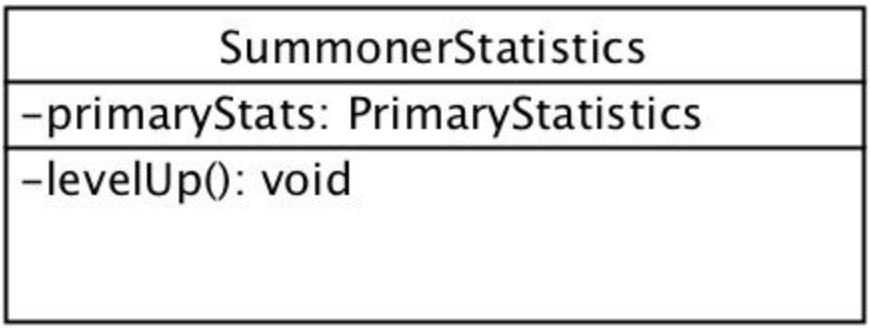
Responsibilities

To create the proper set of Statistics and level up improvements for a Summoner

Collaborators

Works with Entity, Derived Stats, Primary Stats

UML Diagram Here



Implementors

Steven Anderson

Testers

Steven Anderson

Class: MountStatistics

Overview

This class acts as a subtype of Statistics that overrides several features to make the attributes and operations more appropriate for a Mount. It also overrides the level up to better meet the needs of a Mount

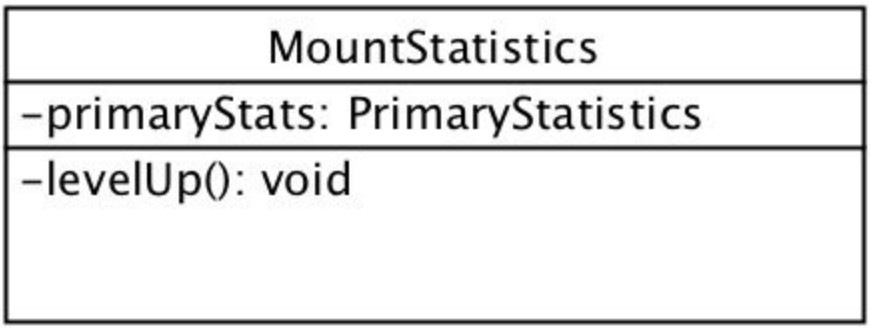
Responsibilities

To create the proper set of Statistics and level up improvements for a Mount

Collaborators

Works with Entity, Derived Stats, Primary Stats

UML Diagram Here



Implementors

Steven Anderson

Testers

Steven Anderson

Class: PrimaryStatistics

Overview

This class acts as a container class to hold the main Primary Statistics such as hardiness and strength. This helps improve cohesion of the Statistics system by removing various mixed-attribute anti-cohesions. This meets the project requirements for Primary Statistics

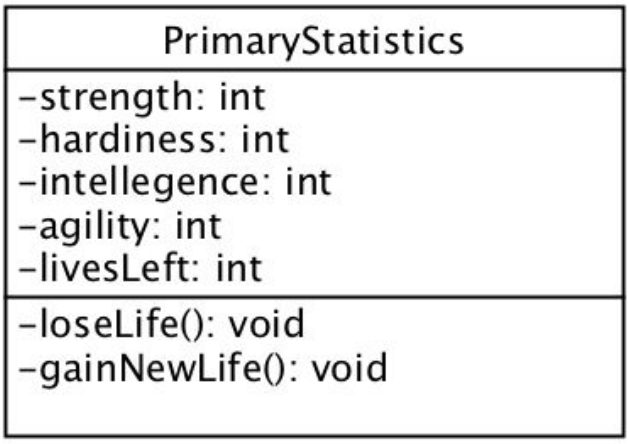
Responsibilities

To track Primary Statistics

Collaborators

Works with Statistics, Derived Statistics

UML Diagram Here



Implementors

Steven Anderson

Testers

Steven Anderson

Class: DerivedStatistics

Overview

This class acts as a container class to hold and calculate the Derived Statistics such as Offensive and Defensive Ratings. This helps improve cohesion of the Statistics system by removing various mixed-attribute anti-cohesions. This meets the project requirements for Derived Statistics.

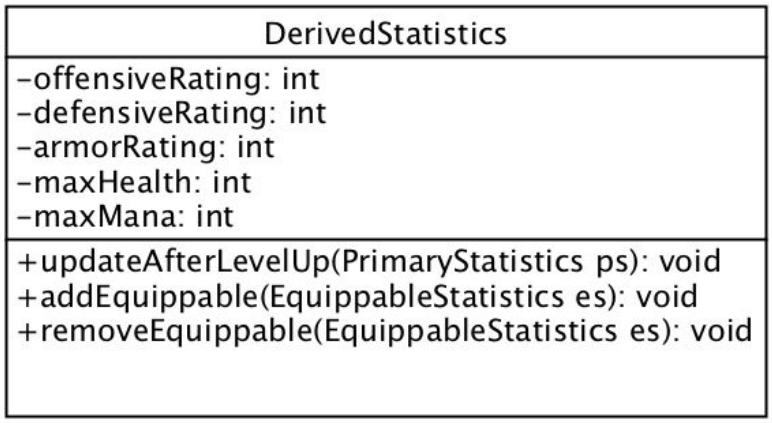
Responsibilities

To store and calculate the Derived Statistics whenever something about the Avatar's Statistics are modifed.

Collaborators

Works with Statistics, Primary Statistics, Equippable Statistics

UML Diagram



Implementors

Steven Anderson

Testers

Steven Anderson

Class: EquippableStatistics

Overview

This class acts as a container class to hold the main Statistics for Equippable items such as damage ratings and armor ratings. This helps improve cohesion of the Statistics system by removing various mixed-attribute anti-cohesions and by creating a reusable mechanism to give Equippable Items usable statistical values.

Responsibilities

To store values such as damage rating for various equippable items such as one handed weapons.

Collaborators

Works with Equipment Manager, Derived Statistics, Statistics, Skill

UML Diagram

Implementors

Steven Anderson

Testers

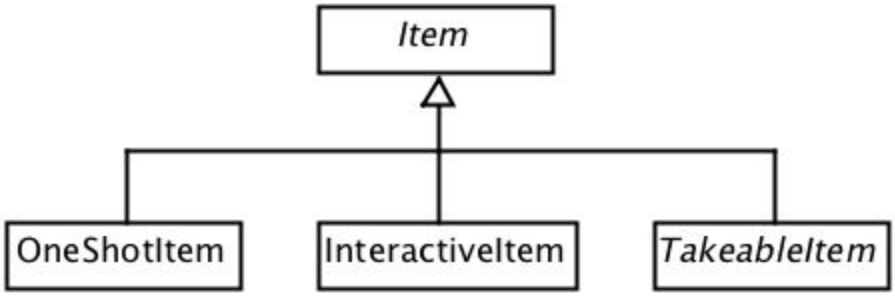
Steven Anderson

Subsystem: Item

Overview

The Item subsystem is the hierarchy of different items, including Takeable Items, Interactive Items, and One Shot Items. This subsystem is needed to meet the project requirements for Items.

UML Diagrams



Class: Item

Overview

The Item superclass is the top level of the hierarchy of different items, including Takeable Items, Interactive Items, and One Shot Items. This class is needed to meet the project requirements for Items. This class stores all attributes and abstract operations common among all Item subtypes, such as name and touching the item.

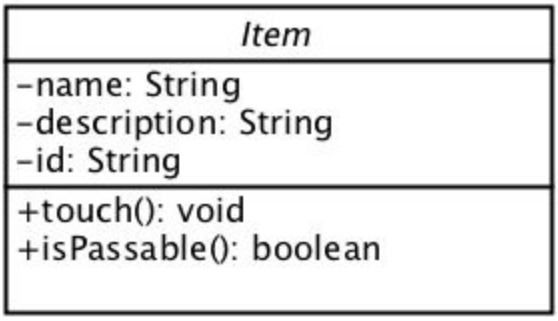
Responsibilities

To store values associated with the item and to do something when touched.

Collaborators

Works with TakeableItems, Obstacles, OneShotItems, and InteractiveItems

UML Diagram



Implementors

Steven Anderson

Testers

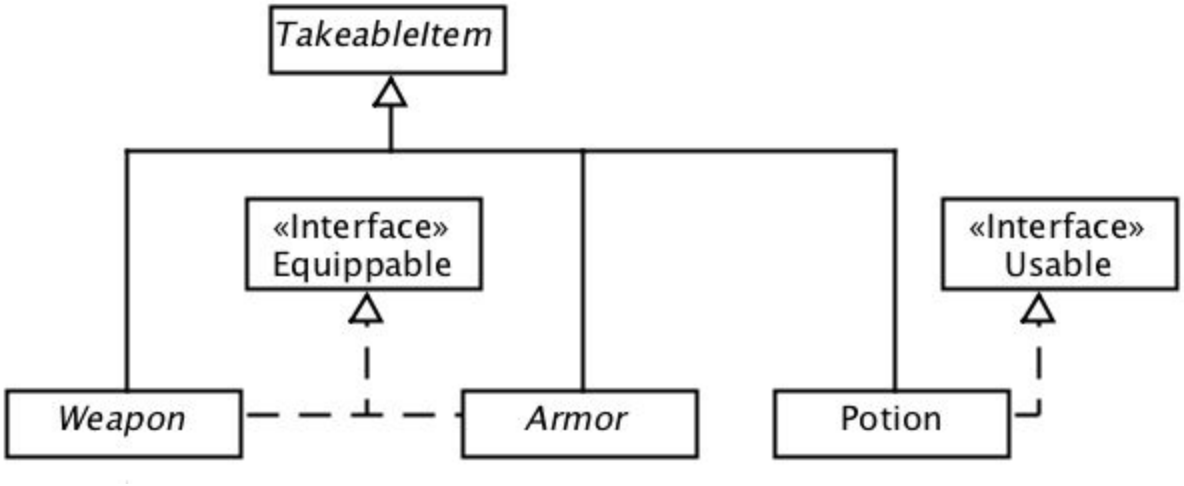
Steven Anderson

Subsystem: TakeableItem

Overview

This subsystem defines items that can be picked up and added to and removed from the Inventory. These items also have a value so that it can be bought and sold.

UML Diagrams



Class: TakeableItem

Overview

This class subclasses Item to create a subset of Items that can be picked up and added to the Inventory on touch. These items also store a value so it can be bought and sold. This class is necessary to meet the project requirements for a takeable item.

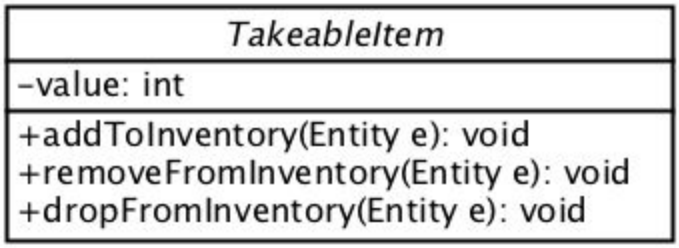
Responsibilities

To be added to the Inventory on touch. To be bought and sold.

Collaborators

Works with Item, Armor, Weapons, Usable, Potion, Inventory

UML Diagram



Implementors

Steven Anderson

Testers

Steven Anderson

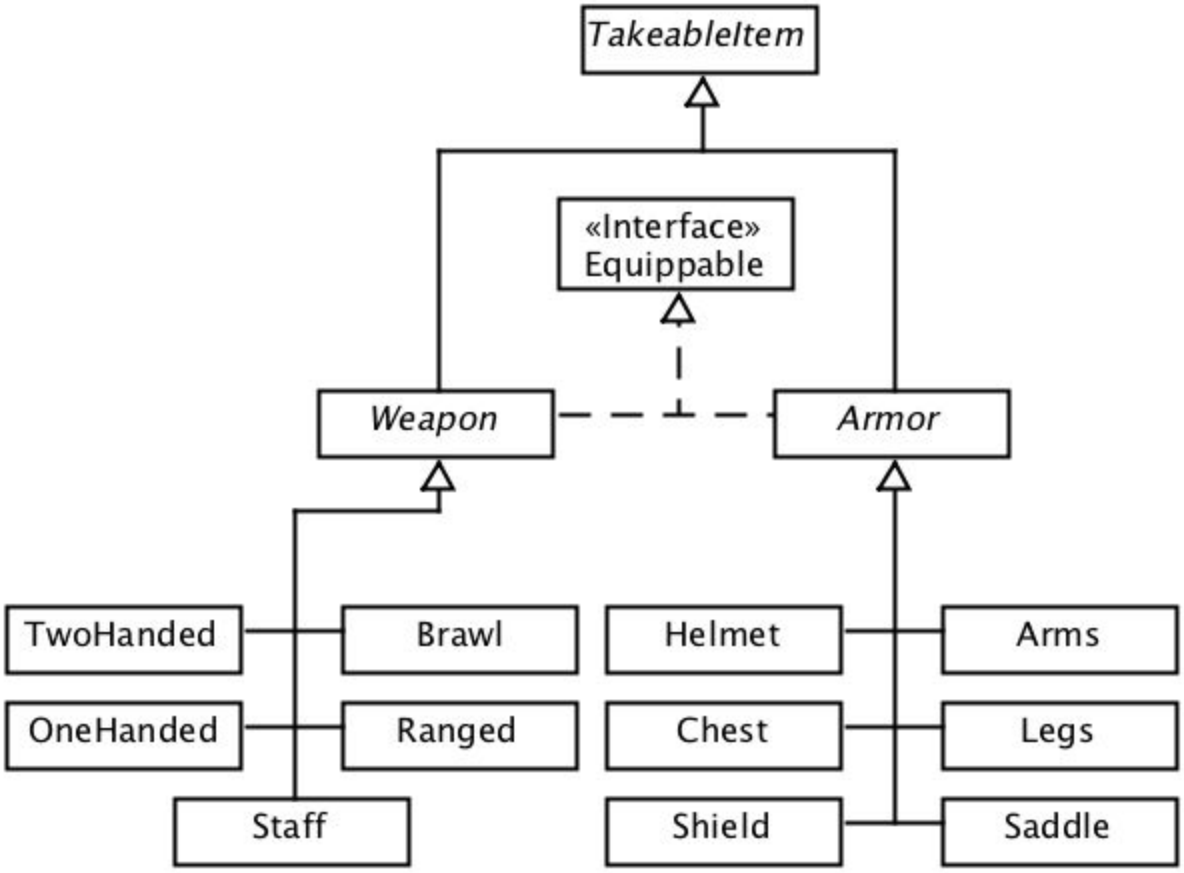
Subsystem: EquippableItem

Overview

This subsystem defines TakeableItems that can be equipped to the Avatar via the Equipment Manager. It uses an Equippable interface to allow the item to be equipped and unequipped from the Equipment Manager. This helps to meet the project requirements for equippable items.

interface is dotted line

UML Diagrams



Class: Equippable

Overview

This interface allows a subset of TakeableItems to be equipped to the character and modify the characters stats when equipped. This will modify the Avatar's stats according to its own stats. This helps meet the project's requirements to allow items to be equipped to the character.

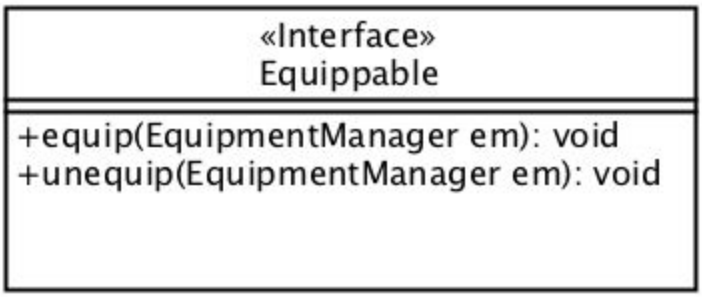
Responsibilities

To allow an item to be equipped. To modify an Avatar's stats when equipped.

Collaborators

Works with Armor, Weapon

UML Diagram



Implementors

Steven Anderson

Testers

Steven Anderson

Class: Weapon

Overview

The subtype of TakeableItems that will be subtyped into the various different weapon types including the Sneak's Ranged weapons and the Smasher's Two Handed weapons. This will differentiate between what type of equippable slot (armor or weapon) the weapon will go into. This class will help meet the requirement for the different weapon types in the project requirements.

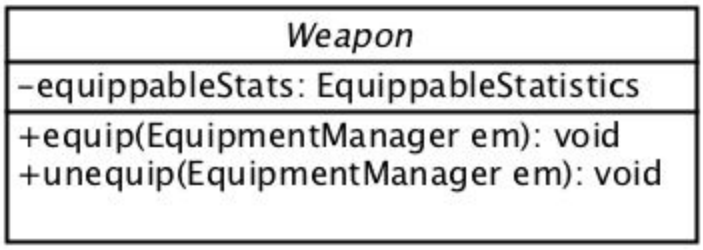
Responsibilities

To be equippable to the weapon slots in the Equipment Manager.

Collaborators

Works with Equippable, TakeableItem, Ranged, TwoHanded, OneHanded, Brawl, EquipmentManager, Inventory

UML Diagram



Implementors

Steven Anderson

Testers

Steven Anderson

Class: Ranged

Overview

The subtype of Weapons that will be subtyped into the Sneak specific weapon. This will differentiate between which equippable slots the weapon will go into. This class will have it's equippable stats modified by the Range skill. This class will help meet the requirement for the Sneak Ranged weapon type in the project requirements.

Responsibilities

To be equipped to the equipment manager. To modify the avatar's stats.

Collaborators

Works with Weapon, EquipmentManager, Inventory

UML Diagram

Implementors

Steven Anderson

Testers

Steven Anderson

Class: OneHanded

Overview

The subtype of Weapons that will be subtyped into a Smasher specific weapon. This will differentiate between which equippable slots the weapon will go into. This class will have it's equippable stats modified by the OneHanded skill. This class will help meet the requirement for the Smasher OneHanded weapon type in the project requirements.

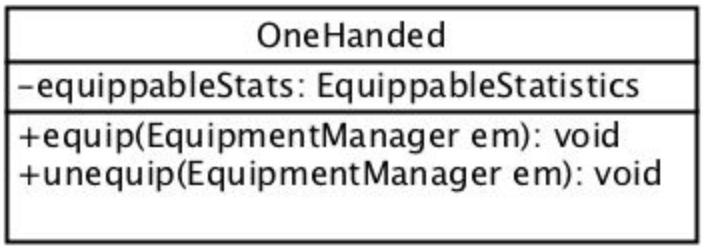
Responsibilities

To be equipped to the equipment manager. To modify the avatar's stats.

Collaborators

Works with Weapon, EquipmentManager, Inventory

UML Diagram



Implementors

Steven Anderson

Testers

Steven Anderson

Class: TwoHanded

Overview

The subtype of Weapons that will be subtyped into a Smasher specific weapon. This will differentiate between which equippable slots the weapon will go into. This class will have it's equippable stats modified by the TwoHanded skill. This class will help meet the requirement for the Smasher TwoHanded weapon type in the project requirements.

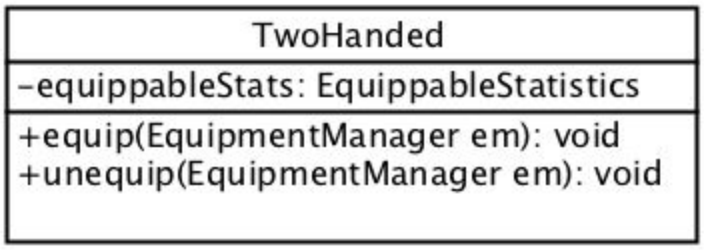
Responsibilities

To be equipped to the equipment manager. To modify the avatar's stats.

Collaborators

Works with Weapon, EquipmentManager, Inventory

UML Diagram



Implementors

Steven Anderson

Testers

Steven Anderson

Class: Brawl

Overview

The subtype of Weapons that will be subtyped into a Smasher specific weapon. This will differentiate between which equippable slots the weapon will go into. This class will have it's equippable stats modified by the Brawl skill. This class will help meet the requirement for the Smasher Brawl weapon type in the project requirements.

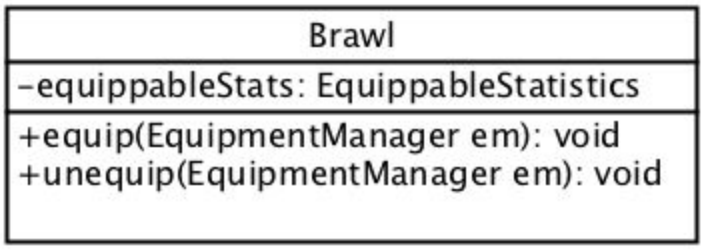
Responsibilities

To be equipped to the equipment manager. To modify the avatar's stats.

Collaborators

Works with Weapon, EquipmentManager, Inventory

UML Diagram



Implementors

Steven Anderson

Testers

Steven Anderson

Class: Staff

Overview

The subtype of Weapon that will be subtyped into the staff equippable slot. This class will help meet the requirement for weapons in the project requirements. This is specialized to work with the Summoner as its melee weapon when mana runs out.

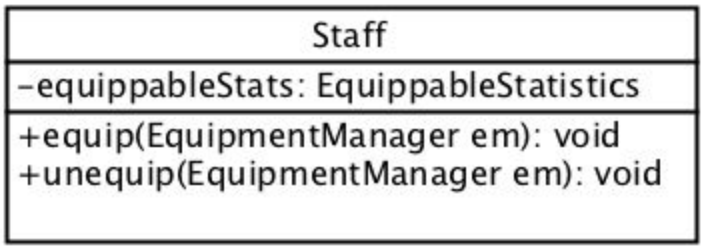
Responsibilities

To be equipped to the equipment manager. To modify the avatar's stats.

Collaborators

Works with Armor, EquipmentManager, Inventory

UML Diagram



Implementors

Steven Anderson

Testers

Steven Anderson

Class: Armor

Overview

The subtype of TakeableItems that will be subtyped into the various different armor types including helmet and chest. This will differentiate between what type of equippable slot (armor or weapon) the weapon will go into. This class will help meet the requirement for equippable armor types in the project requirements.

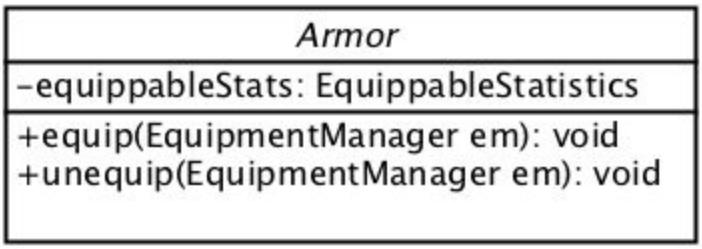
Responsibilities

To be equippable into an armor slot in the Equipment Manager.

Collaborators

Works with Equippable, TakeableItem, Helmet, Chest, Arms, Legs, EquipmentManager, Inventory

UML Diagram



Implementors

Steven Anderson

Testers

Steven Anderson

Class: Helmet

Overview

The subtype of Armor that will be subtyped into the helmet equippable slot. This class will help meet the requirement for equippable armor in the project requirements.

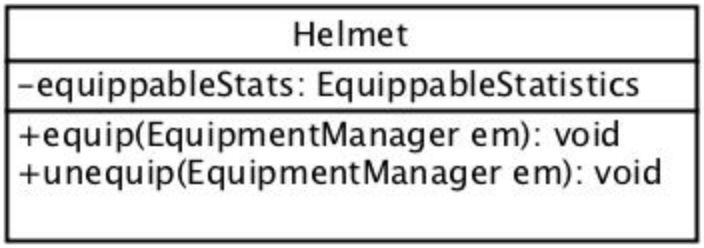
Responsibilities

To be equipped to the equipment manager. To modify the avatar's stats.

Collaborators

Works with Armor, EquipmentManager, Inventory

UML Diagram



Implementors

Steven Anderson

Testers

Steven Anderson

Class: Chest

Overview

The subtype of Armor that will be subtyped into the chest equippable slot. This class will help meet the requirement for equippable armor in the project requirements.

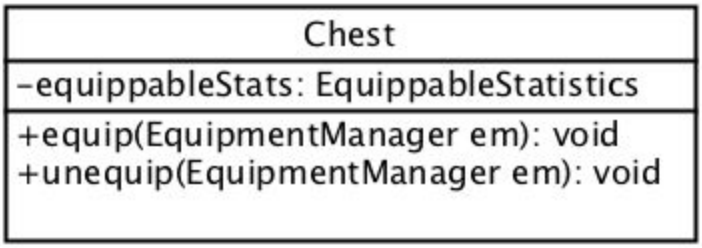
Responsibilities

To be equipped to the equipment manager. To modify the avatar's stats.

Collaborators

Works with Armor, EquipmentManager, Inventory

UML Diagram



Implementors

Steven Anderson

Testers

Steven Anderson

Class: Arms

Overview

The subtype of Armor that will be subtyped into the arms equippable slot. This class will help meet the requirement for equippable armor in the project requirements.

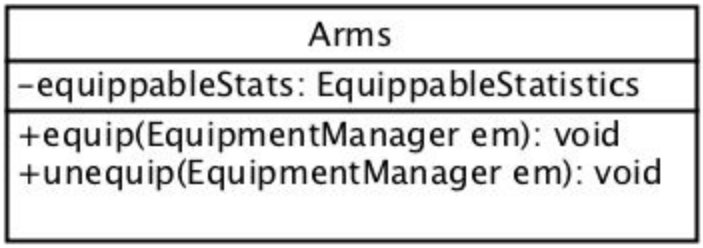
Responsibilities

To be equipped to the equipment manager. To modify the avatar's stats.

Collaborators

Works with Armor, EquipmentManager, Inventory

UML Diagram



Implementors

Steven Anderson

Testers

Steven Anderson

Class: Legs

Overview

The subtype of Armor that will be subtyped into the legs equippable slot. This class will help meet the requirement for equippable armor in the project requirements.

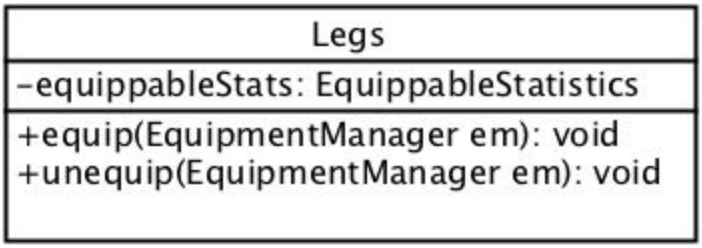
Responsibilities

To be equipped to the equipment manager. To modify the avatar's stats.

Collaborators

Works with Armor, EquipmentManager, Inventory

UML Diagram



Implementors

Steven Anderson

Testers

Steven Anderson

Class: Shield

Overview

The subtype of Armor that will be subtyped into the shield equippable slot. This class will help meet the requirement for equippable armor in the project requirements.

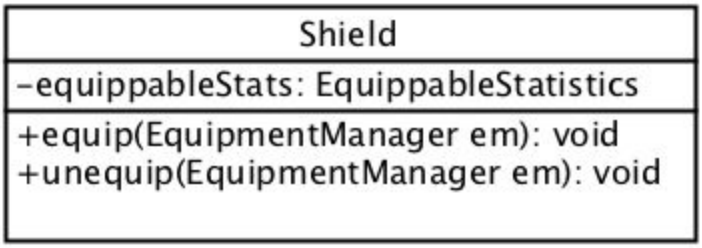
Responsibilities

To be equipped to the equipment manager. To modify the avatar's stats.

Collaborators

Works with Armor, EquipmentManager, Inventory

UML Diagram



Implementors

Steven Anderson

Testers

Steven Anderson

Class: Saddle

Overview

The subtype of Armor that will be subtyped into the saddle equippable slot. This class will help meet the requirement for equippable armor in the project requirements. This is specialized to work with the mount class.

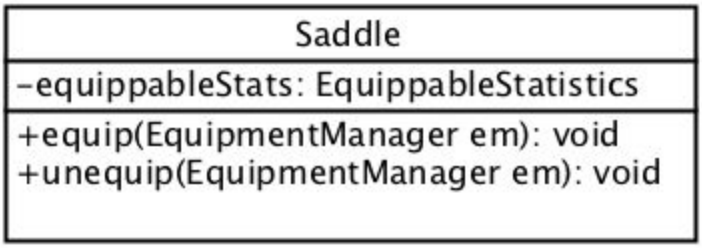
Responsibilities

To be equipped to the equipment manager. To modify the mount's stats.

Collaborators

Works with Armor, EquipmentManager, Inventory

UML Diagram



Implementors

Steven Anderson

Testers

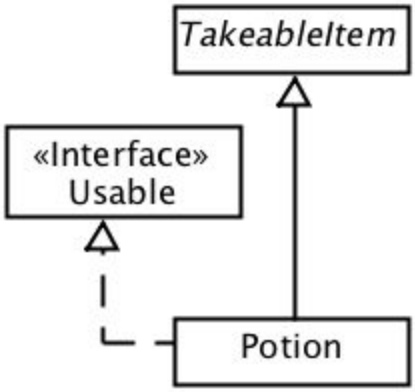
Steven Anderson

Subsystem: UsableItem

Overview

This subsystem defines TakeableItems that can be used after being picked up. It uses a Usable interface to allow the item to be used once and removed from the Inventory. This helps to meet the project requirements for items such as potions.

UML Diagrams



Class: Usable

Overview

This interface allows a subset of TakeableItems to be used once by the character and modify the character appropriately. This helps meet the project's requirements to allow items such as potions to be used.

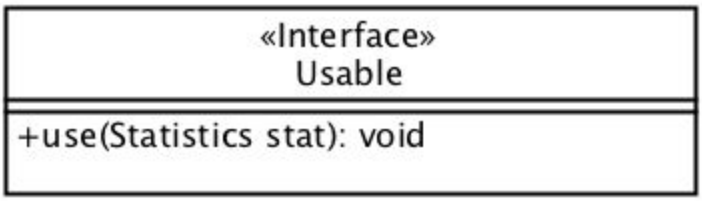
Responsibilities

To allow an item in the inventory to be used and then discarded.

Collaborators

Works with TakeableItem, Potion, Statistics

UML Diagram Here



Implementors

Steven Anderson

Testers

Steven Anderson

Class: Potion

Overview

This is a TakeableItem that implements Usable. It will restore the Avatar's mana and health and then be discarded. This will meet the requirements to allow one time use items to be added to the inventory.

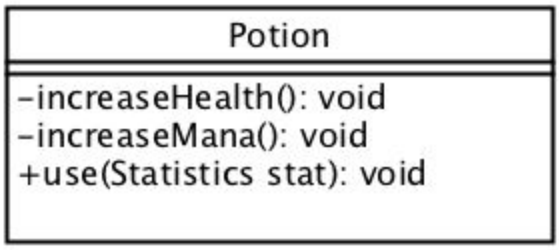
Responsibilities

To restore the health and mana of the avatar. To be discarded after use.

Collaborators

Works with Usable, Statistics

UML Diagram Here



Implementors

Steven Anderson

Testers

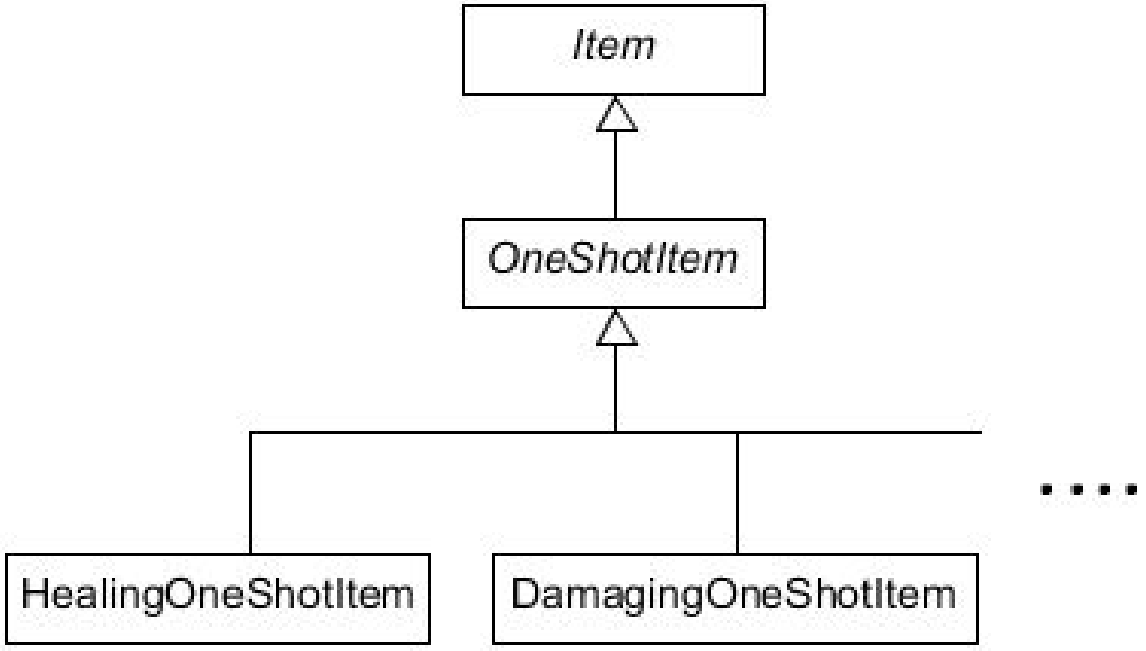
Steven Anderson

Subsystem: One-Shot Items

Overview

This subsystem defines items that are on the map, and, on touch, will be removed from the map and can heal, damage, slow down, etc. an **Entity**. For this iteration, we will be implementing **HealingOneShotItem** and **DamagingOneShotItem**, which will heal and damage an **Entity**, respectively. However, this design leaves **OneShotItem** open for extension if more item types (slow down **Entity**, affects other stats, etc.) want to be added later.

UML Diagrams



Class: OneShotItem

Overview

This class subclasses **Item** to create a subset of **Items** that are activated on touch and have some sort of effect on the **Entity** that activated it.

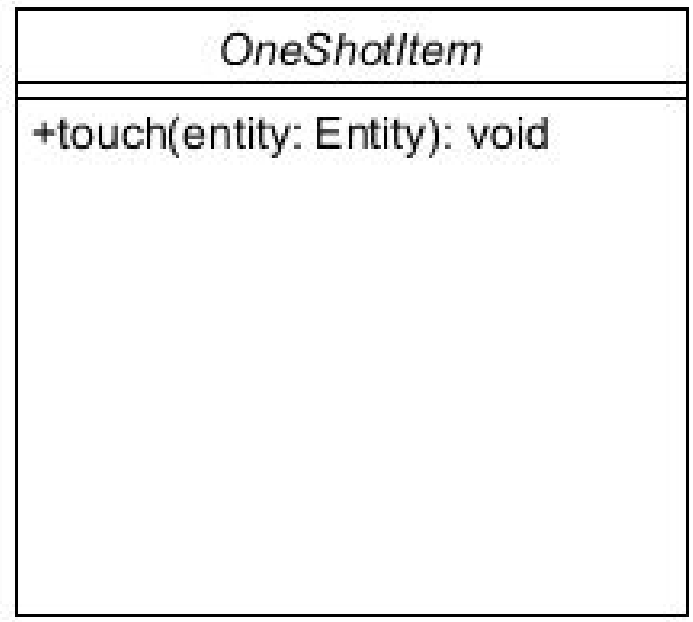
Responsibilities

To occupy a hextile and have some sort of effect on an **Entity** that touches it.

Collaborators

Works with **Item**, **Tile**, and **Entity**

UML Diagram



Implementors

Cormac McCarthy

Testers

Cormac McCarthy

Class: HealingOneShotItem

Overview

This class subclasses **OneShotItem** to create an object that gives health that **Entity** that touches it and then disappears from the map. The attribute **value** represents the amount of health that the **HealingOneShotItem** gives back to the **Entity**.

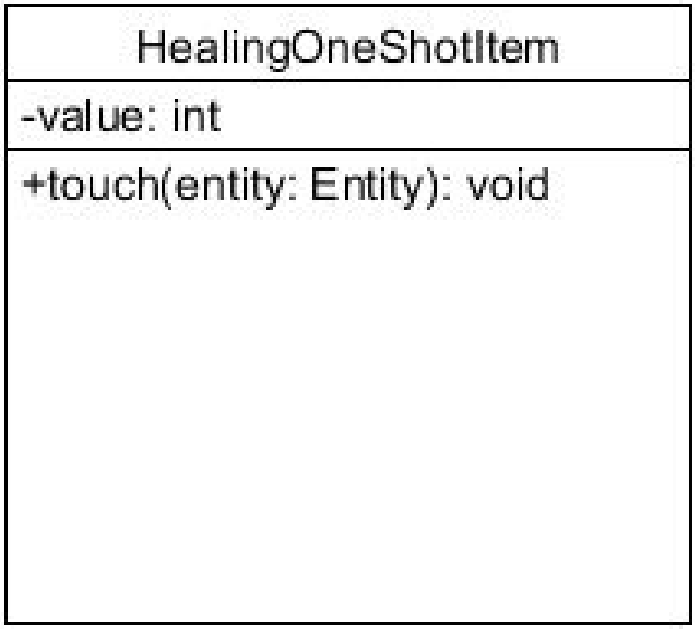
Responsibilities

To occupy a hextile and heal the **Entity** that touches it by a fixed amount.

Collaborators

Works with **OneShotItem**, **Tile**, and **Entity**

UML Diagram



Implementors

Cormac McCarthy

Testers

Cormac McCarthy

Class: DamagingOneShotItem

Overview

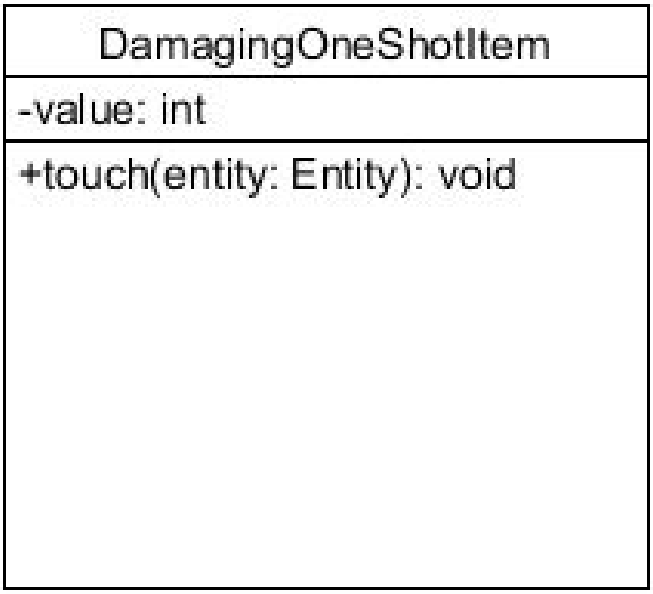
This class subclasses **OneShotItem** to create an object that deals damage to the **Entity** that touches it and then disappears from the map. The attribute **value** represents the amount of damage that the **DamagingOneShotItem** deals to the **Entity**.

Responsibilities

To occupy a hextile and damage the **Entity** that touches it by a fixed amount.

Collaborators

Works with **OneShotItem**, **Tile**, and **Entity**



Implementors

Cormac McCarthy

Testers

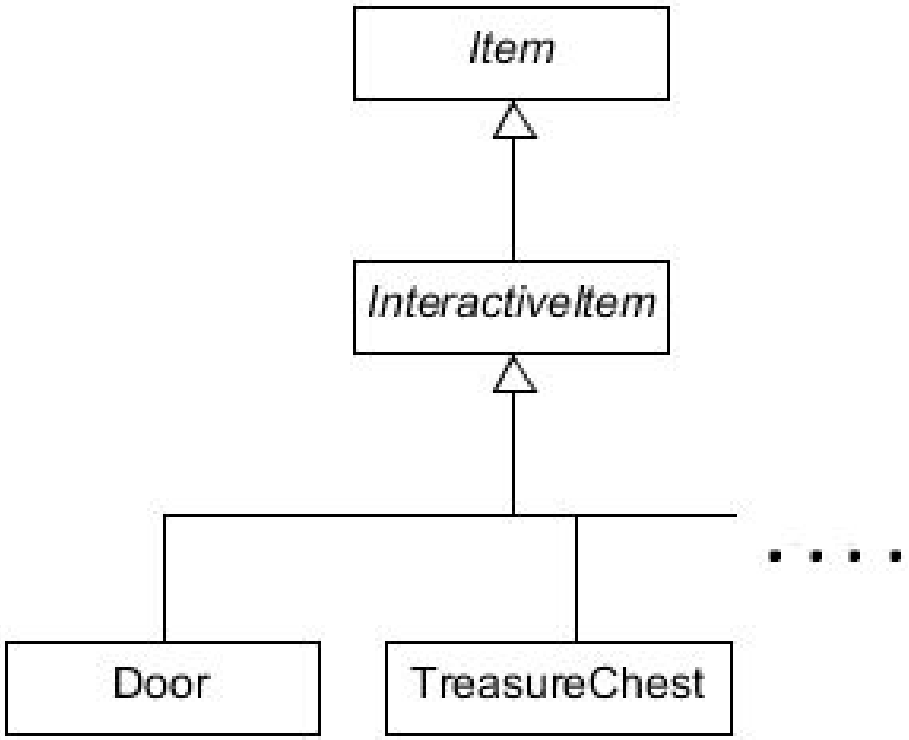
Cormac McCarthy

Subsystem: InteractiveItem

Overview

This subsystem defines items that are (possibly) activated on touch, and may require possession of other items or certain statistics in order to activate.

UML Diagram



Class: InteractiveItem

Overview

This class subclasses **Item** to create a subset of **Items** that are (possibly) activated on touch, and may require other items or certain statistics in order to activate.

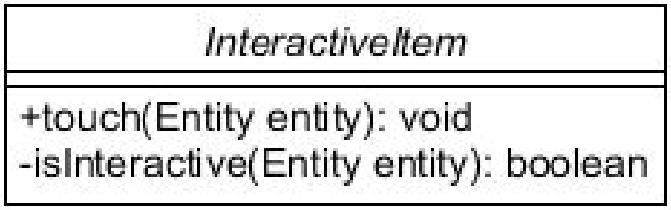
Responsibilities

To occupy a hextile, and possibly have a role in a quest or puzzle, and rewards the **Entity** that touches it for having the necessary items or statistics to activate it.

Collaborators

Works with **Item**, **Tile**, and **Entity**

UML Diagram



Implementors

Cormac McCarthy

Testers

Cormac McCarthy

Class: Door

Overview

This class subclasses **InteractiveItem** to create a concrete class that requires a key in the **Entity**'s inventory in order to open.

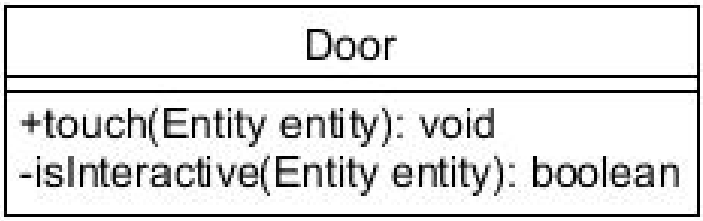
Responsibilities

To occupy a hextile and open when the **Entity** has acquired a key on the map and touches the door.

Collaborators

Works with **InteractiveItem**, **Tile**, and **Entity**

UML Diagram



Implementors

Cormac McCarthy

Testers

Cormac McCarthy

Class: TreasureChest

Overview

This class subclasses **InteractiveItem** to create a concrete class that requires an **Entity** to be a certain level in order to open and receive a reward.

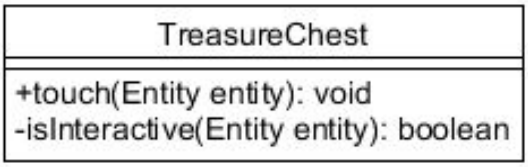
Responsibilities

To occupy a hextile and open when the **Entity** has become a high enough level, and then reward the **Entity**.

Collaborators

Works with **InteractiveItem**, **Tile**, and **Entity**

UML Diagram



Implementors

Cormac McCarthy

Testers

Cormac McCarthy

Subsystem: Inventory

Overview

This system exists to allow an Entity to hold multiple Items. This system adds and removes items held by the Entity and it passes the items to and from the Equipment Manager. This system meets the requirements for an Inventory set forth by the project requirements.

UML Diagrams

Class: Inventory

Overview

This class allows an Entity to hold multiple Items. This class can add, drop, and remove items held by the Entity. It also passes the equippable items to and from the Equipment Manager. Usable items are also accessible from the inventory. This system meets the requirements for an Inventory set forth by the project requirements.

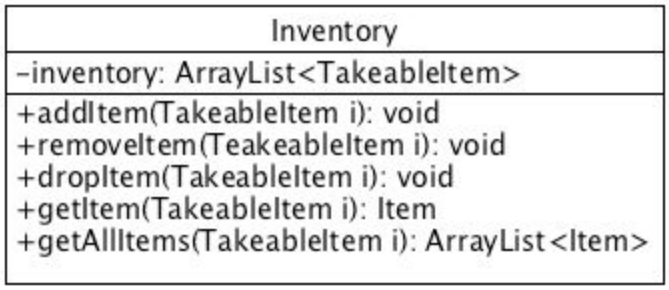
Responsibilities

To add, drop, and remove items held by entity. To manage the transition of items to the equipment manager. To allow usable items to be used after being picked up.

Collaborators

Works with TakeableItem, Entity, EquipmentManager

UML Diagram Here



Implementors

Steven Anderson

Testers

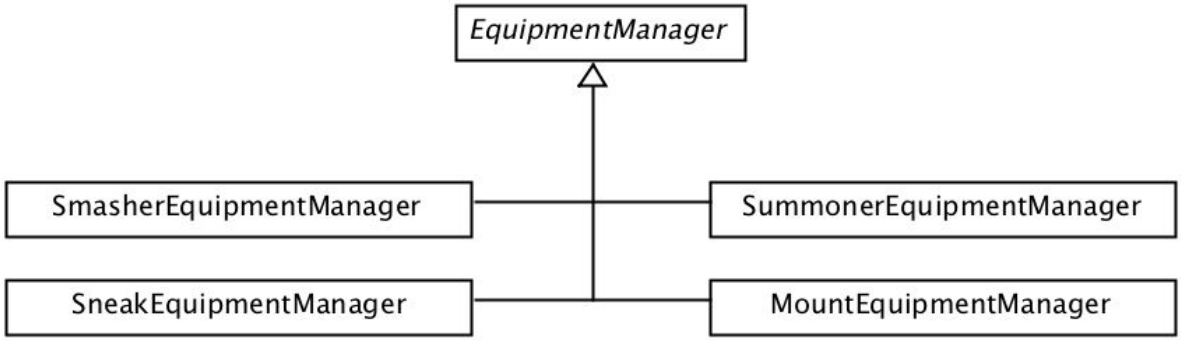
Steven Anderson

Subsystem: Equipment Manager

Overview

This subsystem exists to manage what Items an Avatar has equipped and to modify Statistics accordingly. This system meets the project requirements about allowing an Avatar to equip and unequip items.

UML Diagrams



Class: EquipmentManager

Overview

This class acts as the supertype of the EquipmentManager hierarchy. It holds the operations for modifying statistics of both the items equipped and the avatar that the item is equipped to. This class also includes the operations for equipping and unequipping items. This class meets the project requirements for being able to equip and unequip items.

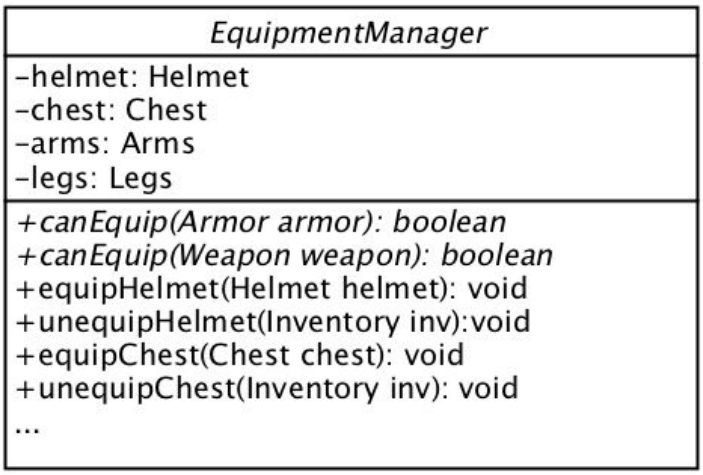
Responsibilities

To equip and unequip items. To modify equippable statistics according to the skill associated with that equipment. To modify the avatar's statistics according to the equipped item's statistics.

Collaborators

Works with Inventory, Avatar, Equippable Statistics, Statistics, Occupation, make mount manager

UML Diagram



Implementors

Steven Anderson

Testers

Steven Anderson

Class: SmasherEquipmentManager

Overview

This class is a subtype of the EquipmentManager supertype.This class serves to ensure that only Smasher items can be equipped to a Smasher. This helps prevent mixed-role cohesion.

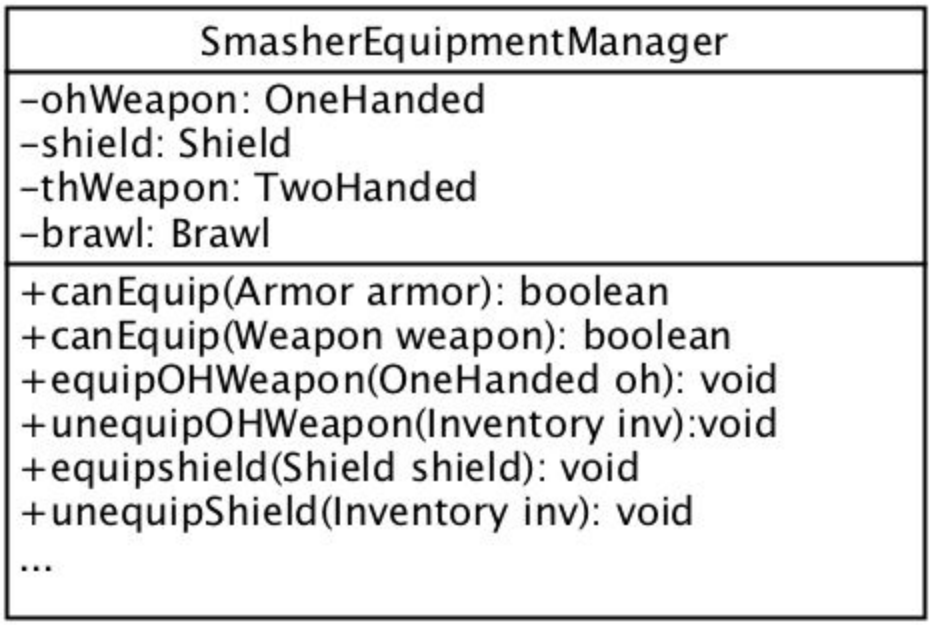
Responsibilities

To ensure only Smasher items are equipped by a Smasher.

Collaborators

Works with Inventory, Avatar, Equippable Statistics, Statistics, Smasher

UML Diagram



Implementors

Steven Anderson

Testers

Steven Anderson

Class: SneakEquipmentManager

Overview

This class is a subtype of the EquipmentManager supertype.This class serves to ensure that only Sneak items can be equipped to a Sneak. This helps prevent mixed-role cohesion.

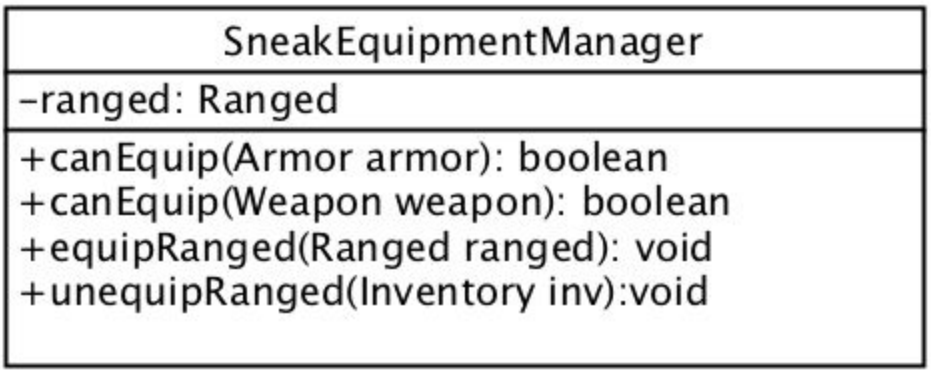
Responsibilities

To ensure only Sneak items are equipped by a Sneak.

Collaborators

Works with Inventory, Avatar, Equippable Statistics, Statistics, Sneak

UML Diagram



Implementors

Steven Anderson

Testers

Steven Anderson

Class: SummonerEquipmentManager

Overview

This class is a subtype of the EquipmentManager supertype.This class serves to ensure that only Summoner items can be equipped to a Summoner. This helps prevent mixed-role cohesion.

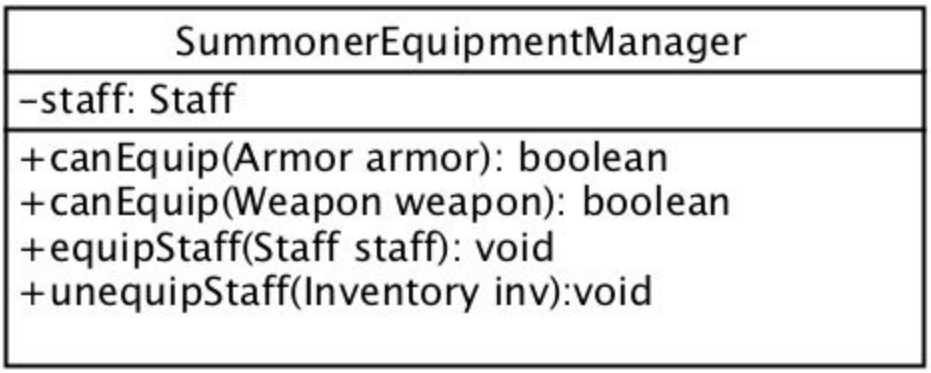
Responsibilities

To ensure only Summoner items are equipped by a Summoner.

Collaborators

Works with Inventory, Avatar, Equippable Statistics, Statistics, Summoner

UML Diagram



Implementors

Steven Anderson

Testers

Steven Anderson

Class: MountEquipmentManager

Overview

This class is a subtype of the EquipmentManager supertype.This class serves to ensure that only Mount items can be equipped to a Mount. This helps prevent mixed-role cohesion.

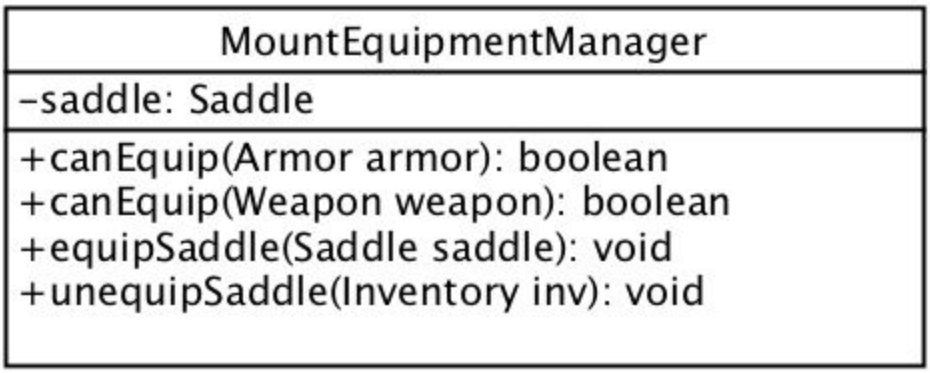
Responsibilities

To ensure only Mount items are equipped by a Mount.

Collaborators

Works with Inventory, Avatar, Equippable Statistics, Statistics, Mount

UML Diagram



Implementors

Steven Anderson

Testers

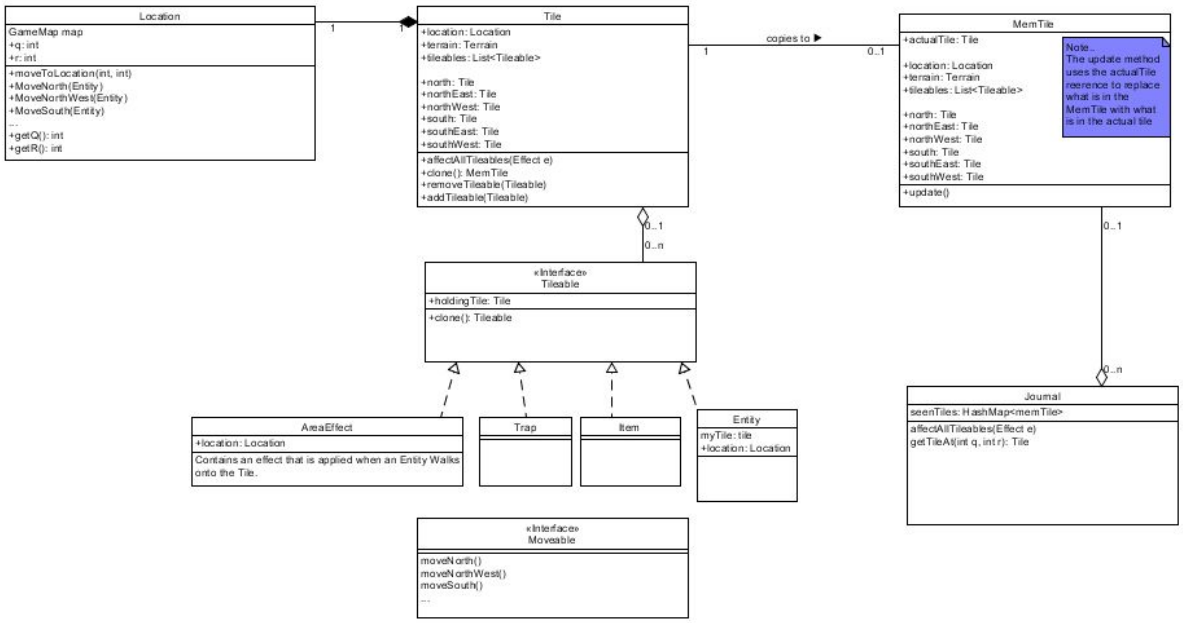
Steven Anderson

Subsystem: Map

Overview

The Map subsystem is the hierarchy of different classes that coordinate the interaction general game objects including the Tiles, Entities, Items, Traps, and Area Effects. It helps manage output to view, interaction with controllers, and combat.

UML Diagrams



Class: Location

Overview

The purpose of **Location** is to keep track of where any Tileable object is. **Location** should not actually be handling the logic of movement (i.e. whether or not a move is legal). Location makes printing easier. Location will also be used to determined distances between tiles.

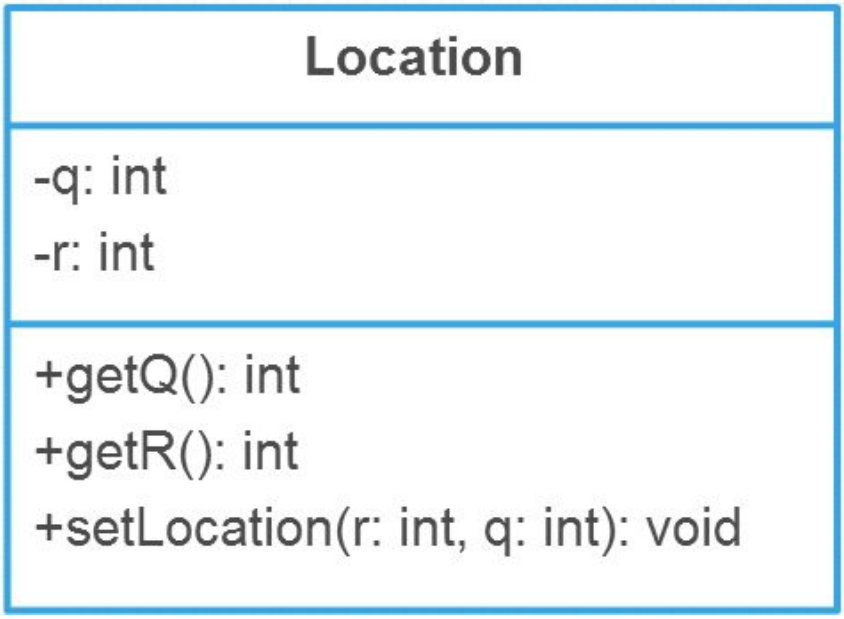
Responsibilities

Holds coordinates for any Tileable object.

Collaborators

Clients: Entity, Item, Tile.
Subcontractors: Tileable.

UML Diagram



Implementors

Jason Owens

Testers

Jason Owens

Class: Tile

Overview

The purpose of **Tile** is to contain all the Tileables within it. Any logic that requires moving from one location on the map to another (including teleportation) would be done using the **Tileable** that is moving and Tile's add/remove operations. Individual objects are responsible for knowing their movement permissions. Tile would also be observable and would notify all observers on any noticeable changes.

Responsibilities

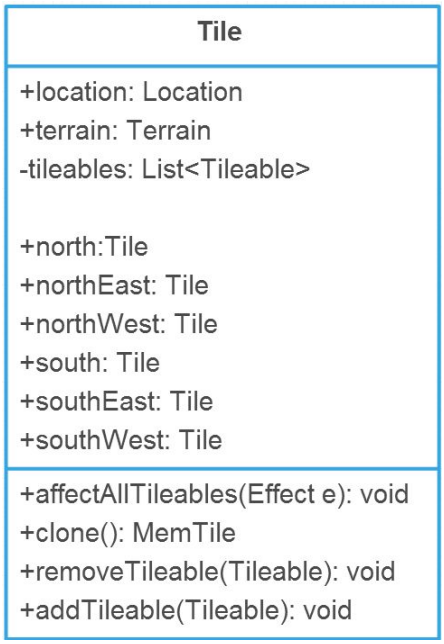
Holds a list of Tileable objects that are on it. Also contains references to all adjacent tiles.

Collaborators

Clients: Entity, MemTile

Subcontractors: Location, Terrain, Tileables, Tile

UML Diagram



Implementors

Jason Owens

Testers

Jason Owens

Class: MemTile

Overview

The purpose of **MemTile** is to store a functional copy of a **Tile** that can be used by an **Avatar**'s memory/Journal. The **MemTile** does not need to support commands that aren't necessary for memory. Note: The update method uses the **actualTile** reference to replace what is in the **MemTile** with what is in the actual tile.

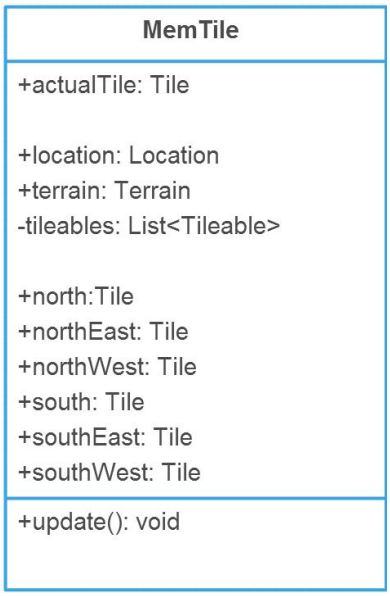
Responsibilities

Holds a 'snapshot' of what was last seen on a tile. Used to make printing the map and fog of war easier.

Collaborators

Clients: Tile
Subcontractors: Location, Terrain, Tileables, Tile

UML Diagram



Implementors

Jason Owens

Testers

Jason Owens

Class: Journal

Overview

The purpose of **Journal** is to store all of the **MemTiles** that an Avatar has seen. This will be used by the view to print the Avatar's active memory. The **Journal** could also theoretically hold anything an Entity might want to remember about its journey, such as statistics and relationships, though it probably won't for this iteration.

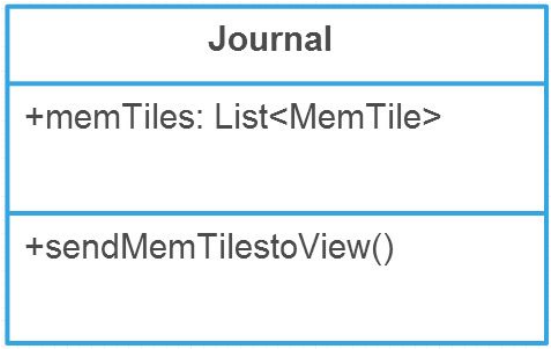
Responsibilities

Store a list of **MemTile** that an Avatar can use, mostly for viewing purposes.

Collaborators

MemTile, Avatar, GameViewport

UML Diagram



Implementors

Jason Owens

Testers

Jason Owens

Interface: Tileable

Overview

The purpose of **Tileable** is to force all objects that can be within tiles to support all of the public methods required of Tileable objects. **Tileables** will implement the visitor pattern to determine how to affect/be affected by what traverses them.

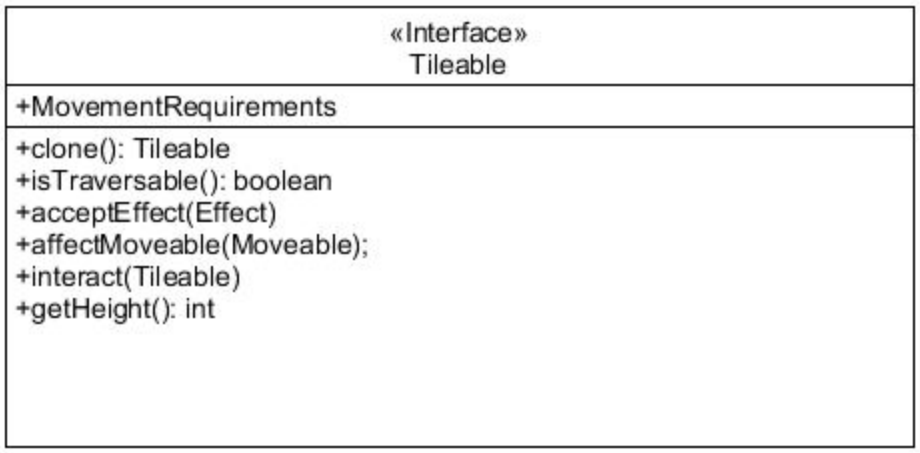
Responsibilities

Maintains the list of functions every Tileable must support

Collaborators

Clients: Entity, Item, Trap, AreaEffect.
Subcontractor: MovementRequirements

UML Diagram



Implementors

Jason Owens

Testers

Jason Owens

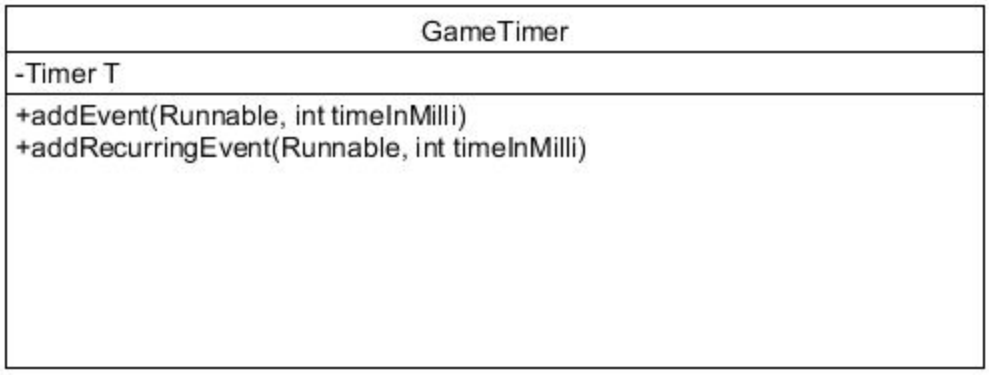
Class: GameTimer

The purpose of **GameTimer** is to allow for events such as movement, which need to be based on time, to be effectively turned into real world times. **GameTimer** would maintain a list of recurring or waiting events that would either repeat every *refresh rate(e.g. movement)* or run just once but after a certain wait time(e.g. removing a temporary effect).

Responsibilities
Keeps track of and executes all objects in the game that need to use timing functionalitiy.

Collaborators
Clients: Entity, Tile, Timed Items, Tileable, Tile
Subcontractor: Runnable

UML Diagram



Implementors
Jason Owens

Testers
Jason Owens

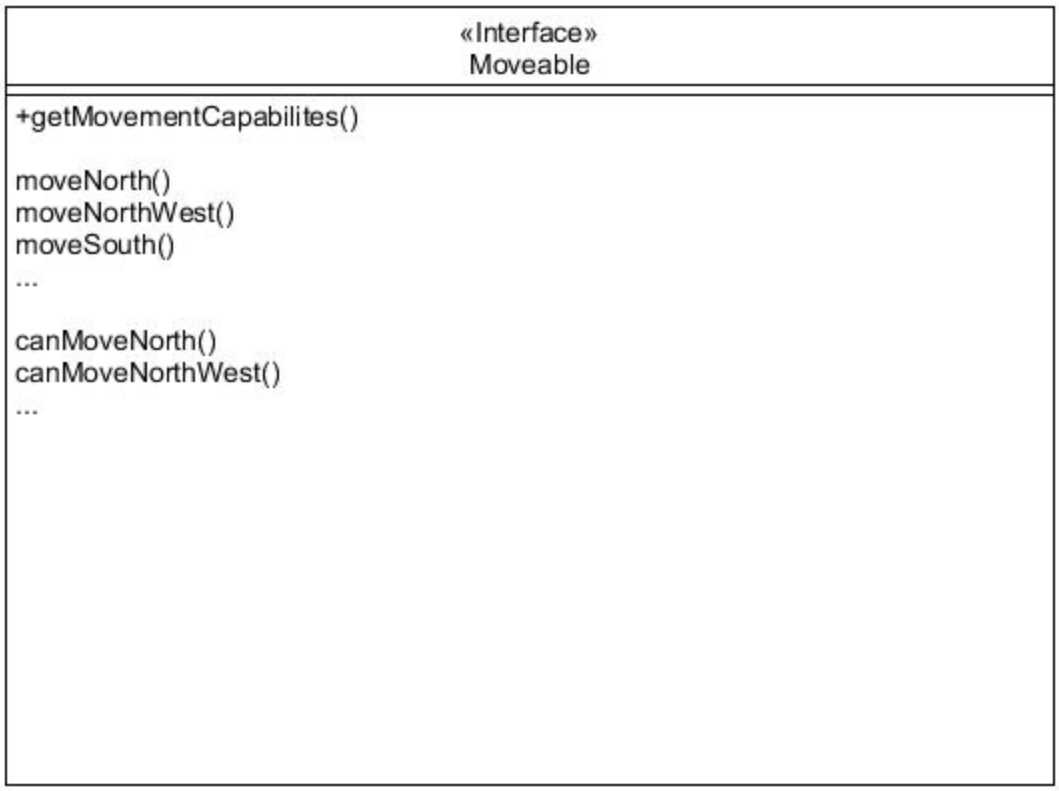
Interface: Moveable

The moveable interface supplies a public interface for any object in our game that wants to be able to traverse the map. It must support all methods in Moveable in order to do that.

Responsibilities
Maintains the methods a Moveable must support

Collaborators
Entities, MovementCapabilities

UML Diagram



Class: MovementRequirements

The purpose of **MovementRequirements** is to store the movement requirements of a Tileable or Terrain. When moving to a new Tile, an object's **MovementCapabilities** must meet the requirements of the additive **MovementRequirements** of every Tileable and Terrain on the Tile.

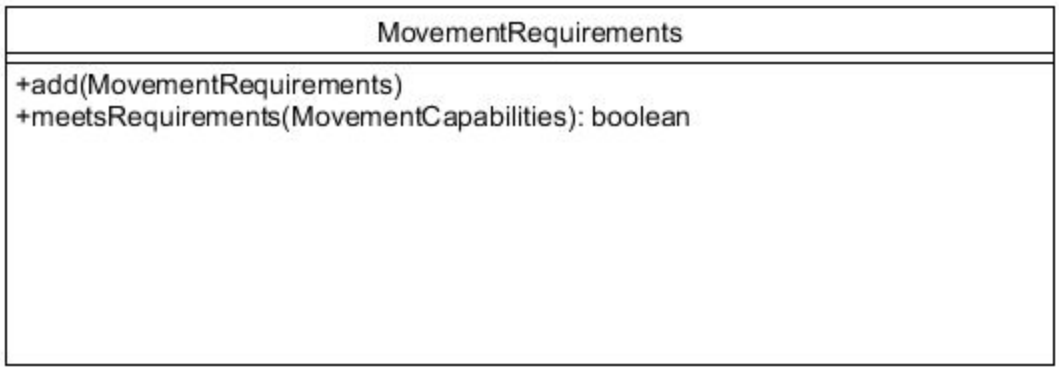
Responsibilities

Maintains the requirements a Moveable must meet in order to pass into this object

Collaborators

Clients: Tileable, Terrain
Contractor: Tile, MovementCapabilities

UML Diagram



Implementors

Jason Owens

Testers

Jason Owens

Class: MovementCapabilities

The purpose of **MovementCapabilities** is to store the movement capabilities of a Moveable object. When moving to a new Tile, an object's **MovementCapabilities** must meet the requirements of the additive **MovementRequirements** of every Tileable and Terrain on the Tile.

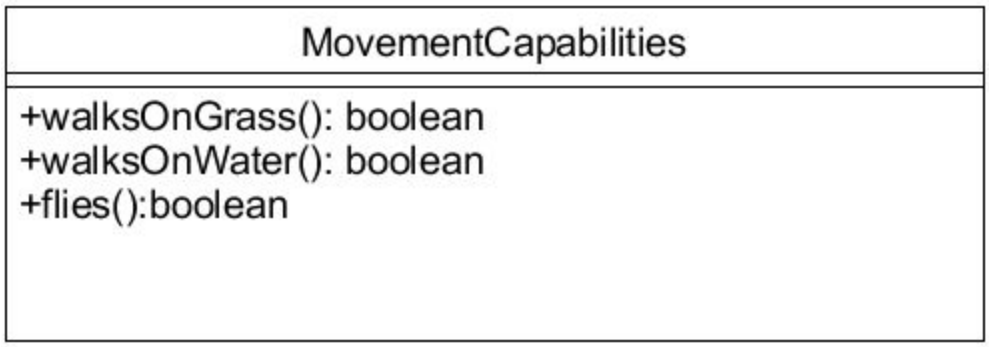
Responsibilities

Maintains the movement capabilities of a moveable

Collaborators

contractors: MovementRequirements, <<interface>> Moveable

UML Diagram



Implementors

Jason Owens

Testers

Jason Owens

Class: AreaEffect

The purpose of **AreaEffect** is to affect the entity that enters the **AreaEffect's** containing tile with the AreaEffect's corresponding effect. **AreaEffect** contains an **Effect** object which is applied to the entity that activates it every time it enters the tile.

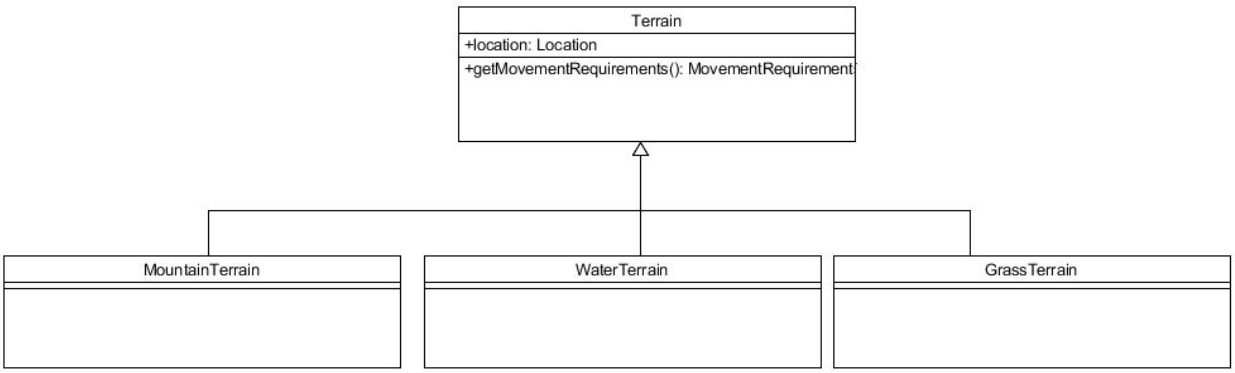
Responsibilities

Hold an Effect to apply to Moveables when they enter a tile

Collaborators

Location, Effect, Tileable, Tile

UML Diagram



Implementors

Jason Owens

Testers

Jason Owens

Class: HealAreaEffect

The purpose of **HealAreaEffect** to be used as an **AreaEffect**, and to apply healing effects to an entity inside of it. The **HealAreaEffect** will have a magnitude of healing that will determine how the entity inside of the **HealAreaEffect** will have it's health affected.

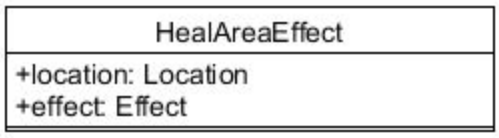
Responsibilities

Heals a moveable when it enters a Tile

Collaborators

AreaEffect, Tileable

UML Diagram



Implementors

Jason Owens

Testers

Jason Owens

Class: DealDamageAreaEffect

The purpose of **DealDamageAreaEffect** to be used as an **AreaEffect**, and to deal damage to an entity inside of it. The **DealDamageAreaEffect** will have a magnitude of damage that will determine how the entity inside of the **DealDamageAreaEffect** will have it's health affected.

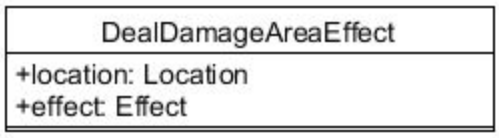
Responsibilities

Damages a moveable when it enters a Tile

Collaborators

AreaEffect, Tileable

UML Diagram



Implementors

Jason Owens

Testers

Jason Owens

Class: InstantDeathAreaEffect

The purpose of **InstantDeathAreaEffect** is to be used as an **AreaEffect**, and to kill an entity inside of it when the Entity enters the tile.

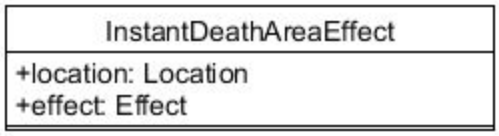
Responsibilities

Kills a moveable when it enters a Tile

Collaborators

AreaEffect, Tileable

UML Diagram



Implementors

Jason Owens

Testers

Jason Owens

Class: LevelUpAreaEffect

The purpose of **LevelUpAreaEffect** is to be used as an **AreaEffect**, and to level-up an Entity inside of it when the Entity enters the tile.

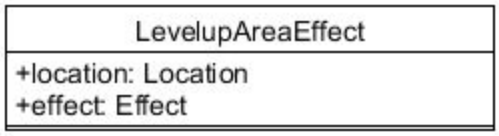
Responsibilities

Levels up a moveable when it enters a Tile

Collaborators

AreaEffect, Tileable

UML Diagram



Implementors

Jason Owens

Testers

Jason Owens

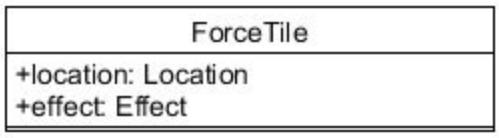
Class: ForceTile (“Rivers”)

The purpose of **ForceTile** to be used as an **AreaEffect**, except to also apply force to an entity inside of it. The **ForceTile** will have a direction and magnitude of force that will determine how the entity inside of the **ForceTile** will have it’s movement affected.

Responsibilities
Moves a moveable after a certain amount of time that it’s been in a Tile.

Collaborators
AreaEffect, Moveable

UML Diagram



Implementors
Jason Owens

Testers
Jason Owens

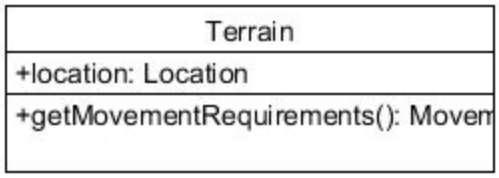
Class: Terrain

The purpose of **Terrain** is to store the movement requirements to move onto a tile and to be used by the view to properly draw the scene. (That is the view will know how to draw a certain Terrain, Terrain will NOT be maintaining a reference to a file path) All **Terrains** will have a getMovementRequirements() function that returnsa MovementRequirements object.

Responsibilities
Holds a **MovementRequirement** for a Tile

Collaborators
Tile, MovementRequirement

UML Diagram



Implementors
Jason Owens

Testers
Jason Owens

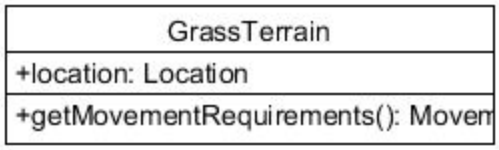
Class: GrassTerrain

The purpose of **GrassTerrain** is to store the movement requirements to move onto a “grass” tile and to be used by the view to properly draw the scene. Its canMove(Entity) function should always return true. Though we may call this “GrassTerrain”, it does not have to be limited to that. We, instead, denote it to whatever ground is normally walkable (be it tile, gravel, magical rainbows, etc).

Responsibilities
Holds a **MovementRequirement** for a Tile

Collaborators
Tile, MovementRequirement

UML Diagram



Implementors
Jason Owens

Testers
Jason Owens

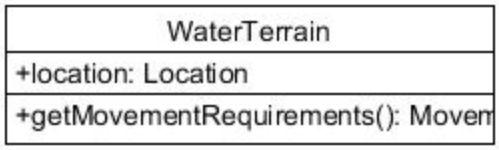
Class: WaterTerrain

The purpose of **WaterTerrain** is to store the movement requirements to move onto a “water” tile and to be used by the view to properly draw the scene. Its canMove(Entity) function should generally return false unless the Entity is flying, can walk on water, or can “swim”. Similarly to Grass, this terrain is not restricted just to water. Instead, we define it as whatever starter level entities can’t walk on normally (i.e. magma, pits of acid, fields of legos, etc).

Responsibilities
Holds a **MovementRequirement** for a Tile

Collaborators
Tile, MovementRequirement

UML Diagram



Implementors
Jason Owens

Testers
Jason Owens

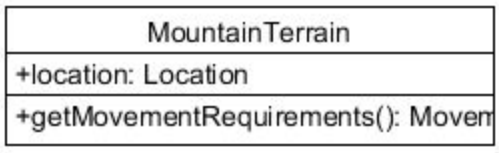
Class: MountainTerrain

The purpose of **MountainTerrain** is to store the movement requirements to move onto a “mountain” tile and to be used by the view to properly draw the scene. Its canMove(Entity) function should always return false.

Responsibilities
Holds a **MovementRequirement** for a Tile

Collaborators
Tile, MovementRequirement

UML Diagram



Implementors
Jason Owens

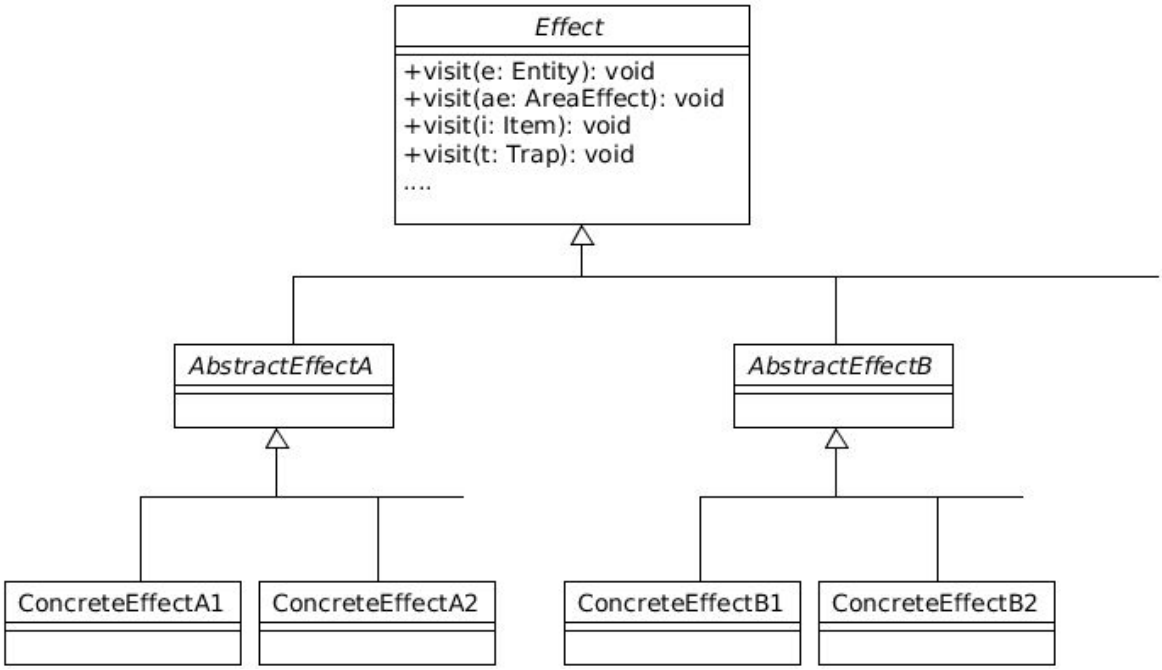
Testers
Jason Owens

Subsystem: Effect

Overview

The **Effect** subsystem is the hierarchy of different classes that coordinate the interaction between **Effect**s and **Tileable** objects. This subsystem utilizes the visitor pattern in order to allow concrete **Effect**s to have a (possibly) unique effect on various **Tileable** objects.

UML Diagrams



Class: Effect

Overview

The abstract class **Effect** is the supertype of all possible effects that can be performed on a **Tileable** object. Subclasses of **Effect** can perform actions like healing/damaging an **Entity**, setting a treasure chest or specific terrain on fire, poisoning an item, etc. **Effect** utilizes the visitor pattern in order to allow **ConcreteEffect**s to have various effects on Tileable objects that it could affect.

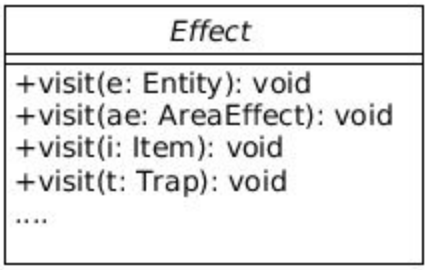
Responsibilities

This class is responsible for defining all of the visit methods in order to capture all of the possible Tileable objects that an **Effect** could possibly have an effect on.

Collaborators

Works with all **Tileable** subclasses, **AbstractEffect**s

UML Diagram



Implementors

Cormac McCarthy

Testers

Cormac McCarthy

Class: AbstractEffect

Overview

The abstract class **AbstractEffect** defines a subset of **Effect**s that differentiates between the different types of **Effect**s that can be implemented. These types of **Effect**s can include **DamagingEffect**, **HealingEffect**, **BurningEffect**, etc. These types of **Effect**s are determined by the developer and can be extended to include more types **Effect**s if needed.

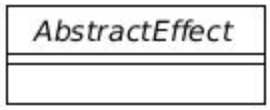
Responsibilities

If there is a **Tileable** object that is affected (in the same way) by all subclasses of an **AbstractEffect**, it will override the necessary visit method.

Collaborators

Works with Tileable objects, **ConcreteEffect**s

UML Diagram



Implementors

Cormac McCarthy

Testers

Cormac McCarthy

Class: ConcreteEffect

Overview

A **ConcreteEffect** defines a subset of **AbstractEffect**s that represents the specific effects within an **AbstractEffect**. For example, corresponding **ConcreteEffect**s of **HealingEffect** could be **MushroomEffect**, **GoldStarEffect**, etc.

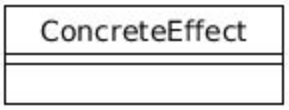
Responsibilities

ConcreteEffects are responsible for causing an effect to certain **Tileable** objects, and these corresponding visit methods must be overridden.

Collaborators

Works with Tileable objects, **ConcreteEffect**s

UML Diagram



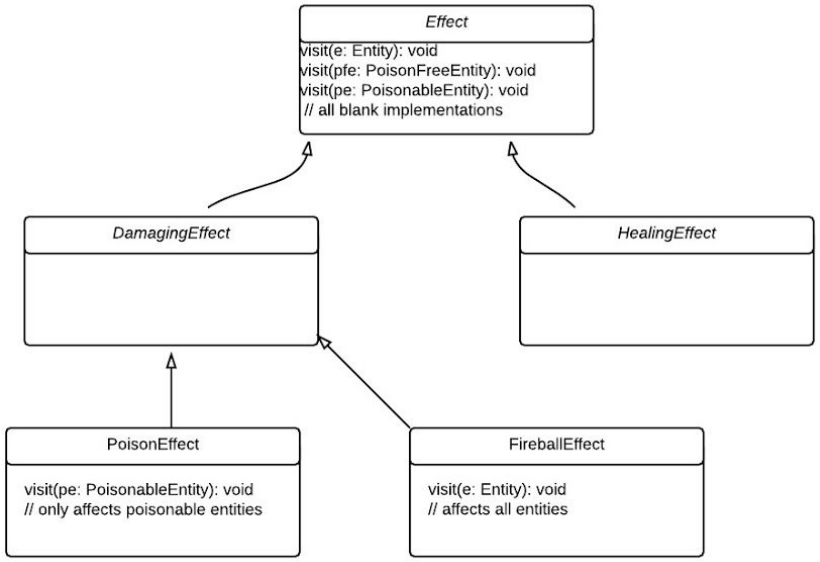
Implementors

Cormac McCarthy

Testers

Cormac McCarthy

Sample implementation:



To help illustrate the Effects paradigm, we have created a sample scenario.

- *Effect* is of the abstract class “Effect” type.
- *DamagingEffect* and *HealingEffect* are of the abstract class “AbstractEffect” type.
- *PoisonEffect* and *FireballEffect* are of the class “ConcreteEffect” type.

As the diagram shows, Effect includes *visit* operations for every type of effectable object. Each *visit* operation is blank, setting the default *visit* behavior to be nothing. If we want an object to be affected, that behavior must be overridden.

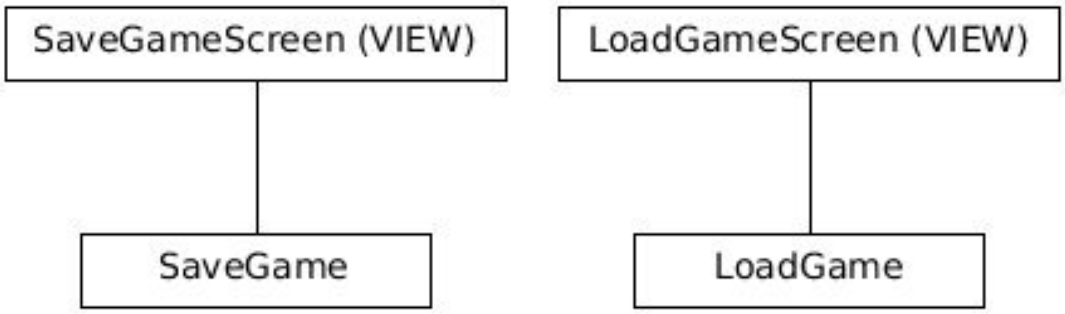
One might notice that (a) DamagingEffect and HealingEffect have no overridden visitor methods. They are primarily used as classes for organization. However, including these abstract classes leaves the door open in the future for default damaging behaviors. One might also notice that (b) PoisonEffect only overrides PoisonableEntity’s *visit* operation. PoisonFreeEntity would instead reference the operation most appropriate for it, which is the default *visit(pfe: PoisonFreeEntity)* operation implemented in *Effect*. This means that PoisonFreeEntity would not be affected.

Subsystem: Load/Save Game

Overview

The load/save game subsystem is responsible for writing the state of the game to a configuration file that can be easily manipulated by the user, as well as read a previously saved (or default) configuration file and load the state of that game.

UML Diagram



Class: SaveGame

Overview

This class holds all the knowledge for saving the state of a game to a specified configuration that can be edited by a user outside of the game.

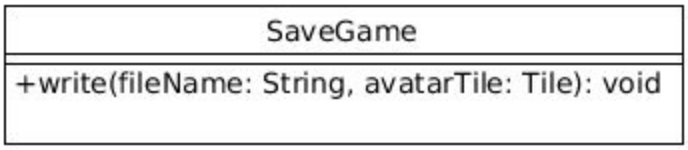
Responsibilities

This class is responsible for writing the state of the game to a configuration file. It takes the **Tile** that the **Avatar** is currently on, and writes the information about all **Tile**s in the map by using breadth-first search.

Collaborators

Works with **SaveGameView**, **Tile**, **Tileable**

UML Diagram



Implementors

Cormac McCarthy

Testers

Cormac McCarthy

Class: LoadGame

Overview

This class holds all the knowledge for loading the state of a game from a specified configuration and running that game.

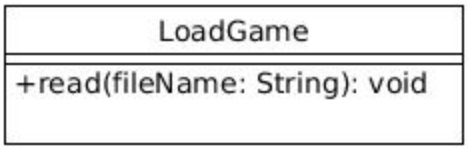
Responsibilities

This class is responsible for restoring the state of a previously saved game so the user can resume gameplay from where they left off. It will create all of the necessary objects in order to restore the state.

Collaborators

Works with **LoadGameView**, **Tile**, **Tileable**

UML Diagram



Implementors

Cormac McCarthy

Testers

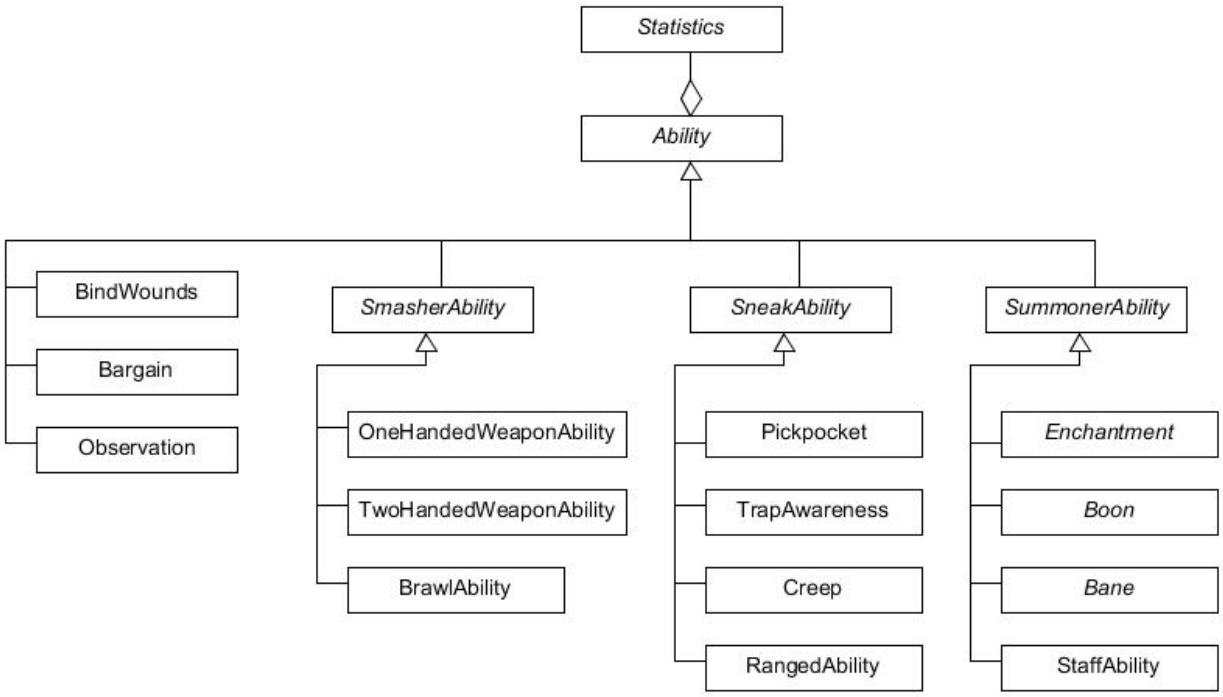
Cormac McCarthy

Subsystem: Skills/Abilities

Overview

The **Skill**/**Ability** subsystem defines all of the possible skills that an **Entity** could have. An **Ability** will have a skill level, which could define the ability's success rate, how much damage it does, how much health it heals, etc. Every **Ability** has a use() method, which is left abstract until implementation is needed by a concrete skill, where the action of the ability will be defined. Each **Ability** will call its use() method whenever the **KeyListener** associated with that ability is activated.

UML Diagram



Class: Ability

Overview

The **Ability** class defines an abstract class that all concrete abilities will inherit from. It will include an integer skill level attribute, as well as a **Statistics** attribute, which represents the corresponding **Entity's Statistics.** This class has a *levelUp* method that will be called whenever the current skill level is wished to be incremented by one. Finally, there is an abstract *use* method that will be overridden by the concrete abilities in order to specify what action occurs when the ability is used.

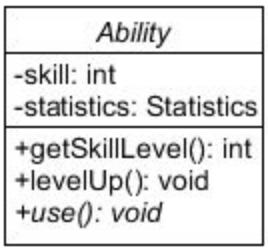
Responsibilities

This class is responsible for providing all concrete abilities with a skill level that can be increased when the player gains skill points and providing a method that will be overridden by subclasses to give functionality to the abilities.

Collaborators

Works with **Statistics**, **Entity**

UML Diagram



Implementors

Cormac McCarthy

Testers

Cormac McCarthy

Class: BindWounds

Overview

This class represents a concrete ability that will heal some amount of the **Entity**'s health when used. The amount of health to be healed will depend on the skill level of the **BindWounds** ability.

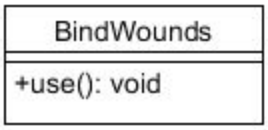
Responsibilities

When the *use* method is called, the **Entity** will gain some amount of health, which depends on the skill level.

Collaborators

Works with **Entity**, **Ability**

UML Diagram



Implementors

Cormac McCarthy

Testers

Cormac McCarthy

Class: Bargain

Overview

This class represents a concrete ability that will allow an **Entity** to buy items for a lower price and sell items for a higher price when interacting with a shopkeeper. How much the **Entity** benefits will depend on the skill level of the **Bargain** object.

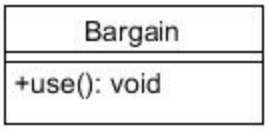
Responsibilities

When the *use* method is called, allow the **Entity** that called it to receive discounts when interacting with the shopkeeper, as well as sell items for a higher price.

Collaborators

Works with **Entity**, **Ability**, **Shopkeeper**

UML Diagram



Implementors

Cormac McCarthy

Testers

Cormac McCarthy

Class: Observation

Overview

This class represents a concrete ability that will allow an **Entity** to gather information about **Tileable** objects on a certain **Tile**. The **Entity** will receive more accurate information about the objects on a **Tile** as the **Observation**'s skill level increases.

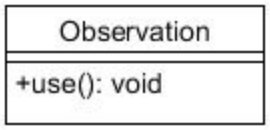
Responsibilities

When the *use* method is called, allow the **Entity** to receive information about the **Tile** that was selected to be observed.

Collaborators

Works with **Entity**, **Tile**, **Tileable**, **Skill**

UML Diagram



Implementors

Cormac McCarthy

Testers

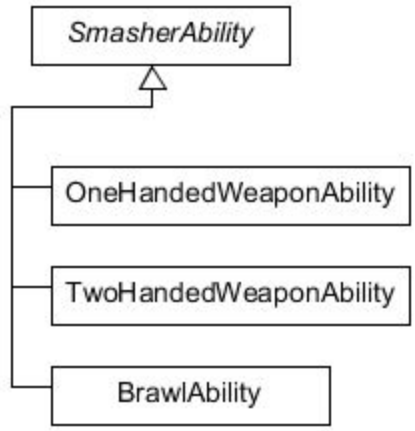
Cormac McCarthy

Subsystem: SmasherAbility

Overview

This subsystem represents the hierarchy of abilities that are subclassed from **Ability** and can only be used by a **Smasher**. These abilities have a reference to **SmasherStatistics** and will be added to the list of abilities that an **Entity** with an occupation of **Smasher** has.

UML Diagram



Class: SmasherAbility

Overview

This class represents a superclass for all concrete abilities that a **Smasher** would have in its list of abilities.

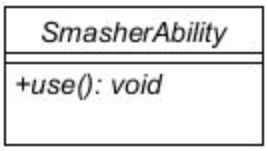
Responsibilities

Define a parent class for the specific abilities that a **Smasher** would use.

Collaborators

Works with **Ability**, **Entity**, **Smasher**

UML Diagram



Implementors

Cormac McCarthy

Testers

Cormac McCarthy

Class: OneHandedWeaponAbility

Overview

This class represents a concrete **SmasherAbility** that will allow a **Smasher** to attack with a one-handed weapon. As the skill level of **OneHandedWeaponAbility** increases, the time it takes between attacks decreases, and the success rate of landing an attack increases.

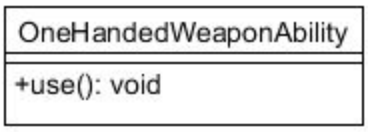
Responsibilities

This class is responsible for housing the knowledge of how much damage a **Smasher**'s attack causes with the current one-handed weapon. This class will pull information from the skill level and the **SmasherStatistics.**

Collaborators

Works with **Smasher, Entity, SmasherAbility, SmasherStatistics, OneHandedWeapon**

UML Diagram



Implementors

Cormac McCarthy

Testers

Cormac McCarthy

Class: TwoHandedWeaponAbility

Overview

This class represents a concrete **SmasherAbility** that will allow a **Smasher** to attack with a two-handed weapon. As the skill level of **TwoHandedWeaponAbility** increases, the time it takes between attacks decreases, and the success rate of landing an attack increases.

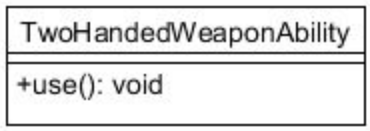
Responsibilities

This class is responsible for housing the knowledge of how much damage a **Smasher**'s attack causes with the current two-handed weapon. This class will pull information from the skill level and the **SmasherStatistics.**

Collaborators

Works with **Smasher, Entity, SmasherAbility, SmasherStatistics, TwoHandedWeapon**

UML Diagram



Implementors

Cormac McCarthy

Testers

Cormac McCarthy

Class: BrawlAbility

Overview

This class represents a concrete **SmasherAbility** that will allow a **Smasher** to attack with their hands. As the skill level of **BrawlAbility** increases, the time it takes between attacks decreases, and the success rate of landing an attack increases.

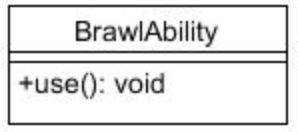
Responsibilities

This class is responsible for housing the knowledge of how much damage a **Smasher**'s attack causes with the just its hands. This class will pull information from the skill level and the **SmasherStatistics.**

Collaborators

Works with **Smasher, Entity, SmasherAbility, SmasherStatistics, Brawl**

UML Diagram



Implementors

Cormac McCarthy

Testers

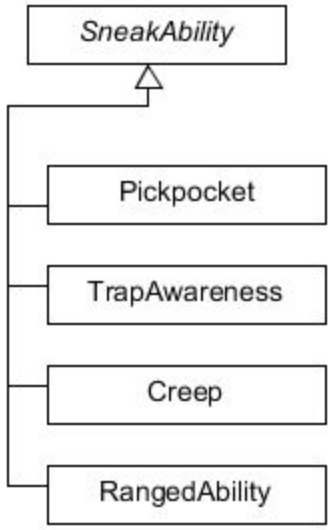
Cormac McCarthy

Subsystem: SneakAbility

Overview

This subsystem represents the hierarchy of abilities that are subclassed from **Ability** and can only be used by a **Sneak**. These abilities have a reference to **SneakStatistics** and will be added to the list of abilities that an **Entity** with an occupation of **Sneak** has.

UML Diagram



Class: SneakAbility

Overview

This class represents a superclass for all concrete abilities that a **Sneak** would have in its list of abilities.

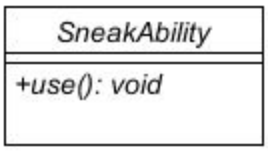
Responsibilities

Define a parent class for the specific abilities that a **Sneak** would use.

Collaborators

Works with **Ability**, **Entity**, **Sneak**

UML Diagram



Implementors

Cormac McCarthy

Testers

Cormac McCarthy

Class: Pickpocket

Overview

This class represents a concrete **SneakAbility** that will allow a **Sneak** to take items from the inventory of another **Entity**. As the skill level of **Pickpocket** increases, the success rate of stealing items will increase.

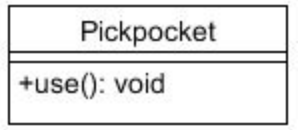
Responsibilities

This class is responsible for housing the knowledge of how to take items from another **Entity**'s inventory, as well as how to place it into the **Sneak**'s inventory.

Collaborators

Works with **Sneak, Entity, SneakAbility, Inventory, Item**

UML Diagram



Implementors

Cormac McCarthy

Testers

Cormac McCarthy

Class: TrapAwareness

Overview

This class represents a concrete **SneakAbility** that will allow a **Sneak** to detect and remove **Trap**s that are located on the map. As the skill level of **TrapAwareness** increases, the success rate of finding a trap and removing it from the map will increase.

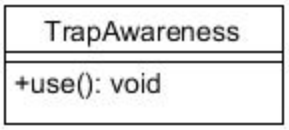
Responsibilities

This class is responsible for determining the success rate of detecting and removing traps from the map based on **Sneak**'s skill level.

Collaborators

Works with **Sneak, Entity, SneakAbility, Trap**

UML Diagram



Implementors

Cormac McCarthy

Testers

Cormac McCarthy

Class: Creep

Overview

This class represents a concrete **SneakAbility** that will allow a **Sneak** to enter a stealth mode while moving around the map, and can surprise an **Entity** from behind and kill it. As the skill level of **Creep** increases, the percent chance of getting caught is lowered and the success rate of killing an **Entity** from behind increases.

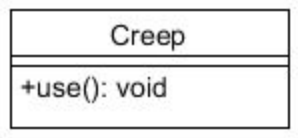
Responsibilities

This class is responsible for determining the success rate of getting caught while moving around the map in stealth mode, as well as housing the knowledge for attacking an **Entity** from behind

Collaborators

Works with **Sneak, Entity, SneakAbility**

UML Diagram



Implementors

Cormac McCarthy

Testers

Cormac McCarthy

Class: RangedAbility

Overview

This class represents a concrete **SneakAbility** that will allow a **Sneak** to attack an **Entity** from a distance with a **RangedWeapon**. As the skill level of **RangedAbility** increases, the success rate of hitting an **Entity** with an arrow increases.

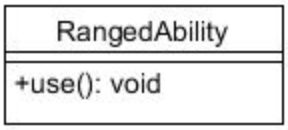
Responsibilities

This class is responsible for determining the success rate of hitting an **Entity** with an arrow based on the skill level of the **RangedAbility.**

Collaborators

Works with **Sneak, Entity, SneakAbility, RangedWeapon**

UML Diagram



Implementors

Cormac McCarthy

Testers

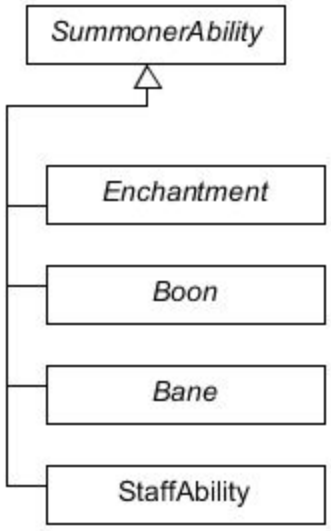
Cormac McCarthy

Subsystem: SummonerAbility

Overview

This subsystem represents the hierarchy of abilities that are subclassed from **Ability** and can only be used by a **Summoner**. These abilities have a reference to **SummonerStatistics** and will be added to the list of abilities that an **Entity** with an occupation of **Summoner** has.

UML Diagram



Class: SummonerAbility

Overview

This class represents a superclass for all concrete abilities that a **Summoner** would have in its list of abilities.

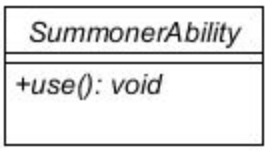
Responsibilities

Define a parent class for the specific abilities that a **Summoner** would use.

Collaborators

Works with **Ability**, **Entity**, **Summoner**

UML Diagram



Implementors

Cormac McCarthy

Testers

Cormac McCarthy

Class: Enchantment

Overview

This class represents an abstract **SummonerAbility** that will allow a **Summoner** to alter the behavior of another **Entity**. **Enchantment** will be subclassed by concrete abilities that will allow the **Entity** to make another **Entity** fall asleep, move away from the current **Entity**, turn into an animal, etc.)

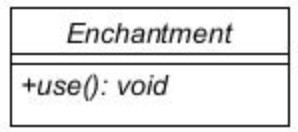
Responsibilities

This class is responsible for being the parent class for all concrete abilities that alter the behavior of another **Entity**.

Collaborators

Works with **Summoner, Entity, SummonerAbility**

UML Diagram



Implementors

Cormac McCarthy

Testers

Cormac McCarthy

Class: Boon

Overview

This class represents an abstract **SummonerAbility** that will allow a **Summoner** to heal, give temporary immunities and defense bonuses, improve statistics, and other benefits. **Boon** will be subclassed by concrete abilities that can do any of these things.

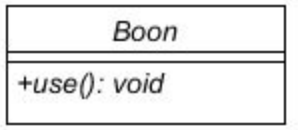
Responsibilities

This class is responsible for being the parent class for all concrete abilities that will benefit the **Entity**.

Collaborators

Works with **Summoner, Entity, SummonerAbility, SummonerStatistics**

UML Diagram



Implementors

Cormac McCarthy

Testers

Cormac McCarthy

Class: Bane

Overview

This class represents an abstract **SummonerAbility** that will allow a **Summoner** to deal damage or harm another **Entity**. **Bane** will be subclassed to allow different concrete abilities that can do damage to be created.

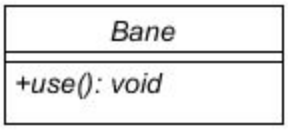
Responsibilities

This class is responsible for being the parent class for all concrete abilities that will do damage to another **Entity**.

Collaborators

Works with **Summoner, Entity, SummonerAbility, SummonerStatistics, Effect**

UML Diagram



Implementors

Cormac McCarthy

Testers

Cormac McCarthy

Class: StaffAbility

Overview

This class represents a **SummonerAbility** that will allow a **Summoner** to attack another **Entity** with a staff. As the skill level of **StaffAbility** increases, the success rate of hitting an **Entity** with a **Staff** will increase.

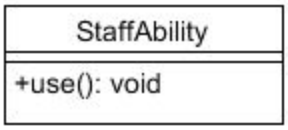
Responsibilities

This class is responsible for determining the success rate of hitting another **Entity** with a **Staff**.

Collaborators

Works with **Summoner, Entity, SummonerAbility, SummonerStatistics, Staff**

UML Diagram



Implementors

Cormac McCarthy

Testers

Cormac McCarthy