DesignSpecification

Group

April 8, 2015

Contents

1 Subsystem Model

The model is comprised of the grunt work of the program. Everything that happens in the game goes to or through the model. The Controllers for the game are set up so that they communicate to the model to do requests from the actions of the user. The model then sends an update to the view to change its state, because the model has changed. The model is well defined and incorporates all the bits and logic of the game. If the Controllers and view did not exist, each subsystem of the game would work, provided that the controller and views where checked for null within the model itself.

1.1 Model

1.1.1 Overview

The model provides an interface for the view and controller to interact with. It properly encapsulates every aspect of the model without revealing too much about the internals of implementation.

1.1.2 Responsibilities

1. Encapsulates all the data in the model
2. Holds all the managers
3. Provides an interface for the model and view to interact with

1.1.3 Collaborators

1. StructuredMap
2. Controller
3. View
4. KeyController
5. The whole model essentially

1.1.4 UML class diagram

{{ extends StateMachine<GameState> }}

- keyPreferences: KeyPreferences
- eventManager: EventManager
- entityManager: EntityManager
- gameMap: GameManager
- itemMap: ItemMap
- projectileManager: ProjectileManager
- triggerManager: TriggerManager
- getKeyPreferences(): void // A bunch of methods

- move/ability functions():
- save(): StructuredMap
- load(structuredMap): void
- push(T): void
- swap(T): void
- pop(): void
- update(): void
- getCurrentUnit(): Entity

1.1.5 Implementor

Josh

1.1.6 Tester

Josh

2 Subsystem Entity

2.0.1 Overview

An entity is a mobile thing, either an Avatar or an NPC or a Mount that has the ability to move itself. It keeps track of its own location on the map and the direction it is facing. An Entity has a name, EntityStatistics, set of Abilities, an Inventory, and an EquipmentManager. Entities are managed by the EntityManager.

2.1 EntityManager

2.1.1 Overview

An EntityManager is responsible for keeping track of all the entities currently on the map. There are three sub divisions of entities. PartyNPCs, which are the Avatar's NPC friends. Avatar. nonPartyNPCs, which are all other NPCs.

2.1.2 Responsibilities

1. Tracks all entities in the game
2. Separates the entities into different categories
3. Calls update methods on each entity

2.1.3 Collaborators

1. Entities
2. StructuredMap

2.1.4 UML class diagram

- partyNpcs: List<NPC>
- nonPartyNpcs: List<NPC>
- avatar: Avatar
- update(): void //calls update on all entities
- addPartyNpc(npc: NPC)
- removePartyNPC(npc: NPC)
- getEntityAtLocation(location: Location): Entity // returns NULL if not found
- save(): StructuredMap
- load(structuredMap): void

2.1.5 Implementor

Josh

2.1.6 Tester

Josh

2.2 Entity

2.2.1 Overview

An entity is a mobile actor. It will be subclassed for the player, npcs, or mounts. It keeps track of its own location on the map and the direction it is facing. An Entity has a name, EntityStatistics, set of Abilities, and an ItemManager.

2.2.2 Responsibilities

1. Equip and unequip equipment.
2. Add and remove inventory items.
3. Maintain its stats
4. Hold a set of Abilities
5. Maintain Direction that it is facing
6. Maintain Location
7. Responsible for moving itself around the map

2.2.3 Collaborators

1. Item
2. Ability
3. Statistics
4. ItemManager
5. Location
6. Angle (Direction)
7. EntityView
8. StructuredMap

2.2.4 UML class diagram

{{Abstract}}

- isFlying(): boolean
- name: Name;
- stats: Stats
- move(Direction);
- addItem(item: takeableItem): void
- removeItem(item: takeableItem): void
- equipItem(item: Item): void
- unequipItem(item: Item): void
- attack(): void
- save(): StructuredMap
- load(structuredMap): void

2.2.5 Implementor

Josh

2.2.6 Tester

Josh

2.3 NPC

2.3.1 Overview

This is a specific entity that maintains its own behaviors. On update, it is responsible for carrying out its behaviors. It is not playable by the Avatar. The Avatar may interact with an NPC by attempting to move into a Tile where an Entity is present.

2.3.2 Responsibilities

1. Maintain Behavior

2.3.3 Collaborators

1. Item
2. Abilities
3. Statistics
4. Inventory
5. Equipment Inventory
6. Location
7. Angle (Direction)
8. StructuredMap

2.3.4 UML class diagram

extends Entity

- behavior: Behavior
- attackAbility: Ability
- save(): StructuredMap
- load(structuredMap): void

2.3.5 Implementor

Josh

2.3.6 Tester

Josh

2.4 Pet

2.4.1 Overview

This is an NPC that will follow you. You can send him commands to Scout for you, and do other Behaviors.

2.4.2 Responsibilities

1. Stay near you
2. Increase your vision
3. Pick up items for you
4. Attack enemies

2.4.3 Collaborators

1. Avatar
2. StructuredMap

2.4.4 UML class diagram

extends NPC

- behavior: Behavior
- save(): StructuredMap
- load(structuredMap): void

2.4.5 Implementor

Josh

2.4.6 Tester

Josh

2.5 Mount

2.5.1 Overview

This is an npc, which, if you so desire, can be mounted. It can can be moved by an AI, here this Mount doesn't have anything special about it all it does it move around when told to, but is also mountable. The avatar may mount a 'Mount' by attempting to move onto a Tile where a Mount NPC is present, and then selecting the 'Mount' option from the popup menu.

2.5.2 Responsibilities

1. Maintain Behavior
2. Allows an Avatar to mount it
3. Can be moved around by an AI
4. Forwards damage & other events to its rider.
5. Returns stats that includes it's riders, and riders inventory.

2.5.3 Collaborators

1. Item
2. Abilities
3. Statistics
4. Inventory
5. Equipment Inventory
6. Location
7. Angle (Direction)
8. StructuredMap

2.5.4 UML class diagram

extends NPC

- behavior: Behavior
- attackAbility: Ability

- rider: Avatar
- setRider(Avatar) //Mount contains rider, and rider is set by the mount in the interaction dialog.
- save(): StructuredMap
- load(structuredMap): void

2.5.5 Implementor

Josh

2.5.6 Tester

Josh

2.6 Avatar

2.6.1 Overview

This is a special entity, one that is controlled by the player and can mount a Mount.

2.6.2 Responsibilities

1. Mount entities
2. Interact with NPCs

2.6.3 Collaborators

1. Item
2. Abilities
3. Statistics
4. ItemManager
5. Location
6. Angle (Direction)
7. Mount
8. StructureMap

2.6.4 UML class diagram

{{Abstract}} Extends Entity

- isFlying()
- name: Name;
- stats: Stats
- move(Direction);
- SkillManager: Skills
- abilities: List<Ability>

- controlManager: ControlManager
- addItem(item: takeableItem): void
- removeItem(item: takeableItem): void
- equipItem(item: Item): void
- unequipItem(item: Item): void
- getListeners():List<Listener>
- save(): StructuredMap
- load(structuredMap): void

2.6.5 Implementor

Josh

2.6.6 Tester

Josh

2.7 Smasher

2.7.1 Overview

An Avatar that is specialized with Smasher characteristics.

2.7.2 Responsibilities

1. Contain Smasher functionality
2. May Equip OneHandedWeapons
3. May Equip TwoHandedWeapons
4. May Equip BrawlingWeapons

2.7.3 Collaborators

1. SmasherSkillManager
2. StructuredMap

2.7.4 UML class diagram

Extends Avatar

- attack(): void
- save(): StructuredMap
- load(structuredMap): void

2.7.5 Implementor

Josh

2.7.6 Tester

Josh

2.8 Summoner

2.8.1 Overview

An Avatar that is specialized with Summoner characteristics.

2.8.2 Responsibilities

1. Contain Summoner functionality
2. May Cast spells
3. May Equip Staff Weapons

2.8.3 Collaborators

1. SummonerSkillManager
2. StructuredMap

2.8.4 UML class diagram

Extends Avatar

- attack(): void //Handles staff skill
- bane(): void
- boon(): void
- enchantment():void
- save(): StructuredMap
- load(structuredMap): void

2.8.5 Implementor

Josh

2.8.6 Tester

Josh

2.9 Sneak

2.9.1 Overview

An Avatar that is specialized with Sneak characteristics.

2.9.2 Responsibilities

1. Contain Sneak functionality
2. May Equip RangedWeapons

2.9.3 Collaborators

1. SneakSkillManager
2. StructuredMap

2.9.4 UML class diagram

Extends Avatar

- save(): StructuredMap
- load(structuredMap): void

2.9.5 Implementor

Josh

2.9.6 Tester

Josh

3 Subsystem Abilities and Skills

3.0.1 Overview

An ability is an abstract class with a perform method. When called, it does something, such as making a fireball, or raising stats, etc. EX) Fireball Ability might create a fireball with damage based on the bane skill. NPC on the other hand, would just have a fireball of a fixed power.

SkillManagers have the set of skills that an avatar has.

3.1 Ability

3.1.1 Overview

3.1.2 Responsibilities

1. Carry out it's respective ability

3.1.3 Collaborators

1. SkillManager

3.1.4 UML class diagram

{{abstract}}

- perform() : void

3.2 SkillManager

3.2.1 Overview

Has the set of abilities. Subclassed for each avatar type. As when they are made, an avatar knows what subclass it is, it knows what skillManager to get, and thus, can then also get the correct abilities.

3.2.2 Responsibilities

3.2.3 Collaborators

1. StructuredMap

3.2.4 UML class diagram

{{abstract}}

- getBarterSkill():int
- getObserveSkill():int
- getBindWoundsSkill():int
- getAttackSkill():int
- save(): StructuredMap
- load(structuredMap): void

3.2.5 Implementor

Josh

3.2.6 Tester

Josh

3.3 SmasherSkillManager

3.3.1 Overview

Has the set of abilities. Subclassed for each avatar type.

3.3.2 Responsibilities

3.3.3 Collaborators

1. StructuredMap

3.3.4 UML class diagram

extends SkillManager

- getTwoHandedSkill():int
- getSingleHandedSkill():int
- get0BrawlingSkill():int
- save(): StructuredMap
- load(structuredMap): void

3.3.5 Implementor

Josh

3.3.6 Tester

Josh

3.4 SneakSkillManager

3.4.1 Overview

Has the set of abilities. Subclassed for each avatar type.

3.4.2 Responsibilities

3.4.3 Collaborators

1. StructuredMap

3.4.4 UML class diagram

extends SkillManager

- getCreepSkill():int
- getPickPocketSkill():int
- getTrapRemoveSkill():int
- getRangedWeaponSkill():int
- save(): StructuredMap
- load(structuredMap): void

3.4.5 Implementor

Josh

3.4.6 Tester

Josh

3.5 SummonerSkillManager

3.5.1 Overview

Has the set of abilities. Subclassed for each avatar type.

3.5.2 Responsibilities

3.5.3 Collaborators

1. StructuredMap

3.5.4 UML class diagram

extends SkillManager

- getBoonSkill():int
- getBaneSkill():int
- getEnchantSkill():int
- getStaffSkill():int
- save(): StructuredMap
- load(structuredMap): void

3.5.5 Implementor

Josh

3.5.6 Tester

Josh

4 Subsystem Map

4.0.1 Overview

This system concerns itself with a collection of tiles, which are the physical terrain.

4.1 GameMap

4.1.1 Overview

The collection of physical tiles that make up a map. A map also manages whether or not an Entity may successfully move to a location.

4.1.2 Responsibilities

1. Hold the set of tiles that defines the area's terrain.

4.1.3 Collaborators

1. Tile
2. Entity
3. StructuredMap

4.1.4 UML class diagram

- tiles: Tile[][]
- canPass(entity, Location): boolean
- touch(entity, Location): void
- save(): StructuredMap
- load(structuredMap): void

4.1.5 Implementor

Matt

4.1.6 Tester

Matt

4.2 ItemMap

4.2.1 Overview

The collection of items that exist on a map, these items do not know their own location, and can be touched to trigger action upon them.

4.2.2 Responsibilities

1. Maintain all the items on the map based on tile location

4.2.3 Collaborators

1. Item
2. StructuredMap

4.2.4 UML class diagram

- items: Collection<Location, Item>
- add(item, location)
- touch(entity, location): void // adds item if it can and removes it from map
- save(): StructuredMap
- load(structuredMap): void

4.2.5 Implementor

Matt

4.2.6 Tester

Matt

4.3 Tile

4.3.1 Overview

A tile represents a single hexagonal tile on the map. It is an abstract class that specific tiles extend from. This class is here to determine what entities can pass through a specific tile. Will be associated with a particular view (e.g. a tile that looks like grass, or a lava, or etc.). A tile also holds an inventory of items.

4.3.2 Responsibilities

1. Define whether or not an entity can stand on the location the tile represents.
2. Maintain a set of items in an Inventory.
3. Pass touch events to each item when an Entity touches a tile.

4.3.3 Collaborators

1. TileRenderer
2. Entity
3. Inventory
4. Item
5. TileView
6. StructuredMap

4.3.4 UML class diagram

{{ Abstract }}

- tileView: TileView
- isPassable(Entity): boolean
- touch(Entity): void
- save(): StructuredMap
- load(structuredMap): void

4.3.5 Implementor

Matt

4.3.6 Tester

Matt

4.4 PassableTile

4.4.1 Overview

This tile exists as a tile that all entities can move through. (Flying or not Flying)

4.4.2 Responsibilities

1. Permit movement.

4.4.3 Collaborators

1. TileRenderer
2. Entity
3. Inventory
4. Item
5. StucturedMap

4.4.4 UML class diagram

PassableTile extends Tile

- isPassable(Entity): boolean
- touch(Entity): void
- save(): StructuredMap
- load(structuredMap): void

4.4.5 Implementor

Matt

4.4.6 Tester

Matt

4.5 ImpassableTile

4.5.1 Overview

This tile specializes Tile to restrict movement of any entity.

4.5.2 Responsibilities

1. Blocks entities from moving through the tile.

4.5.3 Collaborators

1. TileRenderer
2. Entity
3. Inventory
4. Item
5. StructuredMap

4.5.4 UML class diagram

ImpassableTile extends Tile

- isPassable(Entity): boolean
- touch(Entity): void
- save(): StructuredMap
- load(structuredMap): void

4.5.5 Implementor

Matt

4.5.6 Tester

Matt

4.6 AirPassableTile

4.6.1 Overview

This tile specializes Tile to restrict movement of an entity that does not have flying capabilities.

4.6.2 Responsibilities

1. Blocks entities without the capability to fly from moving through the tile.

4.6.3 Collaborators

1. TileRenderer
2. Entity
3. Inventory
4. Item
5. StructuredMap

4.6.4 UML class diagram

AirPassableTile extends Tile

- isPassable(Entity): boolean
- touch(Entity): void
- save(): StructuredMap
- load(structuredMap): void

4.6.5 Implementor

Matt

4.6.6 Tester

Matt

5 Subsystem Triggers

5.0.1 Overview

The trigger system is the primary way that events are applied to entities.They come into action whenever a geometric area of influence is entered. Once this occurs, an event is spawned and pushed to the EventManager which carries out the actual actions. A TriggerManager checks every 'tick' of the game to see if there is an Entity who has entered a Trigger's Geometry. If it has, then the Trigger in question spawns an event, targeted at that Entity, and forwards that to the EventManager.

5.1 TriggerManager

5.1.1 Overview

This is responsible for checking and tracking all existing triggers. At every game tick, it loops through through all applicable entities and passes them to each trigger.

5.1.2 Responsibilities

1. Tracks active triggers.
2. Sends all appropriate entities to triggers' handle method.
3. Distinguishes between non-party triggers and party-applicable triggers.

5.1.3 Collaborators

1. Triggers
2. Entities
3. EntityManager
4. StructuredMap

5.1.4 UML class diagram

- partyTriggers: List<Trigger> // triggers that affect the player & co.
- nonPartyTrigger: List<Trigger> // affect nonParty entities
- neutralTriggers: List<Trigger. // affect anyone
- update(): void // Checks to see if any triggers have been activated and activates them
- addPartyTrigger(Trigger): void
- addNonPartyTrigger(Trigger): void
- addNeutralTrigger(Trigger): void
- save(): StructuredMap
- load(structuredMap): void

5.1.5 Implementor

Kyle

5.1.6 Tester

Kyle

5.2 Trigger

5.2.1 Overview

Triggers spawn events that are pushed to the EventManager when a trigger condition is met. Trigger conditions are met when an Entity crosses into the trigger's range as described by an area of influence. The evaluate() method is responsible for determining if an Entity has caused a trigger condition and consequently creates an event.

5.2.2 Responsibilities

1. Spawns an event when a trigger condition is met.
2. Tracks if it has expired.

5.2.3 Collaborators

1. Events
2. EventManager
3. Entity
4. Area
5. StructuredMap

5.2.4 UML class diagram

- event
- area: Area
- handle(Entity): void
- moveLocation(location: Location)
- hasExpired() : boolean
- save(): StructuredMap
- load(structuredMap): void

5.2.5 Implementor

Kyle

5.2.6 Tester

Kyle

5.3 SingleUseTrigger

5.3.1 Overview

A one time use trigger that will be removed by the trigger manager after it has successfully spawned at least one event. (Multiple events could be spawned if multiple Entities caused the trigger condition to be activated simultaneously).

5.3.2 Responsibilities

Spawns a single series of events.

5.3.3 Collaborators

1. Event
2. EventManager
3. Area
4. Entity
5. StructuredMap

5.3.4 UML class diagram

Implements Trigger

- event
- area: Area
- handle(Entity): void
- hasExpired() : boolean
- save(): StructuredMap
- load(structuredMap): void

5.3.5 Implementor

Kyle

5.3.6 Tester

Kyle

5.4 PermanentTrigger

5.4.1 Overview

This trigger is a trigger that will stay on the map continuously. Consider a lava pit that will always trigger damage.

5.4.2 Responsibilities

Spawns events whenever entities cross the trigger radius.

5.4.3 Collaborators

1. Events
2. EventManager
3. Area
4. Entity
5. StructuredMap

5.4.4 UML class diagram

Implements Trigger

- event
- area: Area
- handle(Entity): void
- hasExpired() : boolean false
- save(): StructuredMap
- load(structuredMap): void

5.4.5 Implementor

Kyle

5.4.6 Tester

Kyle

5.5 TimedTrigger

5.5.1 Overview

This trigger will vanish from the map after a specified period of time. Before that it will apply events to whatever entity crosses its influence area.

5.5.2 Responsibilities

1. Spawns events whenever entities cross the trigger radius until it expires.

5.5.3 Collaborators

1. EventManager
2. Area
3. Entity
4. StructuedMap

5.5.4 UML class diagram

Implements Trigger

- expirationTime : long (ms)
- event
- area: Area
- duration: long (ms)
- handle(Entity): void
- hasExpired() : boolean
- save(): StructuredMap
- load(structuredMap): void

5.5.5 Implementor

Kyle

5.5.6 Tester

Kyle

5.6 ViewableTriggerDecorator

5.6.1 Overview

This object wraps a regular trigger with a view representation. Most triggers do not need views associated with them, but some things, such as area effects (implemented as triggers) should have Decals associated with them (skull & crossbones, etc).

5.6.2 Responsibilities

1. Maintains a TriggerView
2. Forwards behavioral requests to the Trigger it wraps

5.6.3 Collaborators

1. Trigger
2. TriggerView
3. StructuredMap

5.6.4 UML class diagram

Contains a trigger, not implements!

- trigger: Trigger
- -decal: Decal
- handle(Entity): void
- hasExpired() : boolean
- save(): StructuredMap
- load(structuredMap): void

5.6.5 Implementor

Kyle

5.6.6 Tester

Kyle

6 Subsystem Area of Influence

6.0.1 Overview

The Area of Influence system is supported by a few Area abstractions that are used in a few contexts to determine whether or not a location or set of locations is within an area of influence. Primarily used for light sources and triggers.

6.1 Area

6.1.1 Overview

Defines an region of tiles.

6.1.2 Responsibilities

1. Check if a specific location is contained in the area.
2. Provide the locations that define an area.

6.1.3 Collaborators

1. Location
2. StructuredMap

6.1.4 UML class diagram

{{abstract}}

- range: Int
- location: Location
- isInRange(Location)
- getCoveredLocations(): List<Location> // Returns locations in this area
- save(): StructuredMap
- load(structuredMap): void

6.1.5 Implementor

Kyle

6.1.6 Tester

Kyle

6.2 DirectionalArea

6.2.1 Overview

Extends a Area by also specifying an angle at which it is oriented. This is appropriate for Linear and Conical effects which are not omni-directional.

6.2.2 Responsibilities

Checks if a location is contained within the area.

6.2.3 Collaborators

1. Check if a specific location is contained in the area.
2. Provide the locations that define an area.
3. StructuredMap

6.2.4 UML class diagram

Extends Area

- range: Int
- direction: Angle
- isInRange(Location)
- getCoveredLocations(): List<Location> // Returns locations in this area
- save(): StructuredMap
- load(structuredMap): void

6.2.5 Implementor

Kyle

6.2.6 Tester

Kyle

6.3 LinearArea

6.3.1 Overview

Defines a linear region of tiles.

6.3.2 Responsibilities

1. Check if a specific location is contained in the area.
2. Provide the locations that define an area.

6.3.3 Collaborators

1. Location
2. StructuredMap

6.3.4 UML class diagram

Extends DirectionalArea

- isInRange(Location)
- getCoveredLocations(): List<Location> // Returns locations in this area
- save(): StructuredMap
- load(structuredMap): void

6.3.5 Implementor

Kyle

6.3.6 Tester

Kyle

6.4 ConicalArea

6.4.1 Overview

Defines a conical area of tiles.

6.4.2 Responsibilities

1. Determines whether or not a location is within an area.

6.4.3 Collaborators

1. Location
2. StructuredMap

6.4.4 UML class diagram

Extends DirectionalArea

- isInRange(Location) : boolean
- getCoveredLocations(): List<Location> // Returns locations in this area
- save(): StructuredMap
- load(structuredMap): void

6.4.5 Implementor

Kyle

6.4.6 Tester

Kyle

6.5 RadialArea

6.5.1 Overview

Defines a radial area. (Circle)

6.5.2 Responsibilities

1. Checks whether a location is within its radius.

6.5.3 Collaborators

1. Location
2. StructuredMap

6.5.4 UML class diagram

Extends Area

- isInRange(Location) : boolean
- getCoveredLocations(): List<Location> // Returns locations in this area
- save(): StructuredMap
- load(structuredMap): void

6.5.5 Implementor

Kyle

6.5.6 Tester

Kyle

7 Subsystem Events

7.0.1 Overview

Events are objects that perform an action at one time or over a period of time when created. These events will be tied to triggers which are specific areas on the map that fire an event when triggered. There will be many different subclasses of events that have the ability to perform a variety of actions. There might be a StatModifierEvent that heals an entity over a period of time or a button pressed event that plays music when activated. The possibilities are endless with this system. Having this abstraction gives us a lot of power and it'll allow us to create a very dynamic game that is fun to play. They are managed by the EventManager, who dispatches their effects to the Entities that they target.

7.1 Event

7.1.1 Overview

An event is an abstract class that encapsulates an action that can be performed with a Timer. One-time use events can be parameterized with a duration of zero. Events are managed by the EventManager, which handles removing expired events from its event queue. Also, in certain contexts, an Event won't know its trigger upon construction. In these cases, the Trigger reads in the Target for the Event, and sets the Events target in that context. In using an item (i.e. Potion) from the inventory, the Event may be constructed with its target; In the event of stepping on a TakeDamage Area-Effect (Trigger), it will set the Entity to the Triggers Event Target. Events are then passed along to the EventManager, who is responsible for dispatching and discontinuing their consequences. ALSO, maybe if one doesn't set the Target for an event throw a UntargetedEventException. (just a thought)

7.1.2 Responsibilities

1. Performs an action with a Timeout
2. Expire after its duration has passed

7.1.3 Collaborators

1. Entity
2. EventManager

* UML class diagram {{abstract}}

- duration: long (ms)
- onBegin(): void //called by the event manager
- onExpire(): void //called by the event manager
- hasExpired(): boolean
- perform(): void

7.1.4 Implementor

Josh

7.1.5 Tester

Josh

7.2 UnsourcedEvent

7.2.1 Overview

An UnsourcedEvent is a type of event that is targeted to affect one Entity. That Entity is the target of the Event, and is the sole receiver of the consequence of the UnsourcedEvent.

7.2.2 Responsibilities

1. Performs an action on a target entity when perform() is invoked.
2. Expire after its duration has passed

7.2.3 Collaborators

1. Entity
2. EventManager

* UML class diagram {{abstract}}

- target: Entity
- Event(double duration, Entity target)
- onBegin(): void //called by the event manager
- onExpire(): void //called by the event manager
- hasExpired(): boolean
- perform(): void

7.2.4 Implementor

Josh

7.2.5 Tester

Josh

7.3 StatisticModifierEvent

7.3.1 Overview

A StatisticModifierEvent is a type of UnsourcedEvent that modifies the Statistics of the Target Entity. An example would be getting a Strength Bonus, or Drinking a Potion.

7.3.2 Responsibilities

1. Modifies Statistics of a target entity when perform() is invoked.

7.3.3 Collaborators

1. Entity
2. EventManager

* UML class diagram extends UnsourcedEvent

- statistics: EntityStatistics
- StatisticModifierEvent(Entity, EntityStatistic, duration)
- onBegin()
- perform() // MixedInstance :'(

7.3.4 Implementor

Josh

7.3.5 Tester

Josh

7.4 BehaviorModifierEvent

7.4.1 Overview

A BehaviorModifierEvent is a type of Event, in which a new State/Behavior is added to the Targeted Entity. For example, adding a 'Frozen' behavior to an Entity once hit with an iceball.

7.4.2 Responsibilities

1. Adding / Modifying a Behavior of an Entity

7.4.3 Collaborators

1. Entity
2. EventManager

* UML class diagram extends UnsourcedEvent

- newBehavior: Behavior
- StatisticModifierEvent(Entity, Behavior, duration)
- onBegin()
- perform()

7.4.4 Implementor

Josh

7.4.5 Tester

Josh

7.5 SkillModifierEvent

7.5.1 Overview

A SkillModifierEvent modifies the Skill set of a targeted Entity. For example, consider the Bargain skill. Once you use Bargain skill, it will reduce the Bargain skill of the targeted Entity, thus reducing the prices of the purchased Items.

7.5.2 Responsibilities

1. Modifies the SkillPoint Levels of a targeted Entity.

7.5.3 Collaborators

1. Entity
2. EventManager

* UML class diagram extends UnsourcedEvent

- target: Entity
- Event(double duration, SkillCollection skills, Entity)
- onBegin(): void //called by the event manager
- onExpire(): void //called by the event manager
- hasExpired(): boolean
- perform(): void
- setTarget(Entity): void

7.5.4 Implementor

Josh

7.5.5 Tester

Josh

7.6 PrintEvent

7.6.1 Overview

PrintEvent prints words to a menu or console. For example, it may print out the stats of the Entity we are observing.

7.6.2 Responsibilities

1. Print out things to a Dialog / Console.

7.6.3 Collaborators

1. Entity
2. EventManager

* UML class diagram extends UnsourcedEvent

- target: Entity
- duration: long (ms)
- Event(double duration, Entity,TextHandle)
- onBegin(): void //called by the event manager
- onExpire(): void //called by the event manager
- hasExpired(): boolean
- perform(): void

7.6.4 Implementor

Josh

7.6.5 Tester

Josh

7.7 SourcedEvent

7.7.1 Overview

SourcedEvent is a special Event, that affects two entities: A 'source' entity, and a 'target' entity. Consider an example PickPocket. You need to get Items / Money from one Entity, and place them into your Inventory / Bank.

7.7.2 Responsibilities

1. Represent an event that affects two entities.
2. its perform() will invoke methods on two Entities

7.7.3 Collaborators

1. Entity
2. EventManager

* UML class diagram extends Event

- target: Entity
- destination: Entity
- Event(double duration, Entity target, Entity src)
- onBegin(): void //called by the event manager
- onExpire(): void //called by the event manager
- hasExpired(): boolean
- perform(): void

7.7.4 Implementor

Josh

7.7.5 Tester

Josh

7.8 EventManager

7.8.1 Overview

The EventManager keeps track of current Events. At each game tick, it calls an Event's perform method and checks to see if it has expired. Note that the perform method is called first, to ensure One-Time-Use events run. If it has expired, it removes the Event from the e and allows it to be garbage collected. The EventManager will receive incoming events from spawned by triggers, abilities, and other entities.

7.8.2 Responsibilities

1. Maintain a list of events that are currently active
2. Accept new events
3. Call each event's perform method at each game tick
4. Remove any events once they expire

7.8.3 Collaborators

1. Events
2. StructuredMap

* UML class diagram

- eventList: List<Event>
- update(): void
- addEvent(event: Event): void
- save(): StructuredMap
- load(structuredMap): void

7.8.4 Implementor

Josh

7.8.5 Tester

Josh

8 Subsystem Items

8.0.1 Overview

These in general are “things” in the game. Items can be touched and used. Items in the game fall under a few separate categories. OneShot items are ones that are activated on touch and then cease to exist, these will not be in our item hierarchy they will instead be handled as one time triggers. Obstacle items are items that block the pathway of a player. Interactive items are items that perform some action when they are touched. Takeable items fall into two categories: items that can be “used” and items that are equipment (their “use” is to be equipped).

8.1 Item

8.1.1 Overview

A thing in the game.

8.1.2 Responsibilities

1. Respond to an entity's touch
2. Have a use
3. Can act as an obstacle when it is on a map tile

8.1.3 Collaborators

1. Entity
2. ItemView
3. StructuredMap

8.1.4 UML class diagram

{{ Abstract }}

- itemView: ItemView
- touch(Entity) : void
- use(Entity) : void
- getInfo(): String
- save(): StructuredMap
- load(structuredMap): void

8.1.5 Implementor

Joe

8.1.6 Tester

Joe

8.2 TakeableItem

8.2.1 Overview

An Item that can be picked up by an Entity.

8.2.2 Responsibilities

Define an item that can be held by an Entity.

8.2.3 Collaborators

1. Entity
2. StructuredMap

* UML class diagram Extends Item

- touch(Entity) : void
- use(Entity) : void
- getInfo(): String
- save(): StructuredMap
- load(structuredMap): void

8.2.4 Implementor

Joe

8.2.5 Tester

Joe

8.3 ConsumableItem

8.3.1 Overview

An Item that can be picked up by an Entity, then, from the Inventory Menu, can be used to initiate an Event. Examples of Events that can be activated would be a 'HealEvent', which would then target the Avatar, and heal Damage. This ConsumableItem described could possibly be a Potion, or a HealStone.

8.3.2 Responsibilities

Define an item that can be held by an Entity. Be Usable from the Inventory Screen. Send an Event to the EventManager.

8.3.3 Collaborators

1. Event
2. Entity
3. StructuredMap

* UML class diagram Extends TakeableItem

- touch(Entity) : void
- use(Entity) : void
- getInfo(): String
- save(): StructuredMap
- load(structuredMap): void

8.3.4 Implementor

Joe

8.3.5 Tester

Joe

8.4 InteractiveItem

8.4.1 Overview

An item that can be interacted with by an entity, it will perform an action when a prerequisite is met on an entity touching it. The scope of InteractiveItems in this game are limited to Door Items, which require a special type of TakeableItem to be present in the Avatar's Inventory. The InteractiveItems will block Entity's movement until the requirement is met.

8.4.2 Responsibilities

Define an item that can be interacted with.

8.4.3 Collaborators

1. Entity
2. TakeableItem
3. StructuredMap

8.4.4 UML class diagram

Extends Item

- requiredItem: TakeableItem
- touch(Entity) : void
- use(Entity) : void
- getInfo(): String
- save(): StructuredMap
- load(structuredMap): void

8.4.5 Implementor

Joe

8.4.6 Tester

Joe

8.5 EquipableItem

8.5.1 Overview

Items that can be equipped.

8.5.2 Responsibilities

1. Maintain the equipment's statistics.
2. Knows which slot it gets equipped to

8.5.3 Collaborators

1. EquipmentManager
2. Statistics
3. StructuredMap

8.5.4 UML class diagram

Extends Item

- getStats();
- getInfo(): String
- save(): StructuredMap
- load(structuredMap): void

8.5.5 Implementor

Joe

8.5.6 Tester

Joe

8.6 ChestPiece

8.6.1 Overview

A piece of armor worn on the chest.

8.6.2 Responsibilities

1. Maintain's the ChestPiece's statistics
2. Knows it should be equipped to the Armor slot

8.6.3 Collaborators

1. EquipmentManager
2. Statistics
3. StructuredMap

8.6.4 UML class diagram

Extends Equippable

- getStats();
- save(): StructuredMap
- load(structuredMap): void

8.6.5 Implementor

Joe

8.6.6 Tester

Joe

8.7 Boots

8.7.1 Overview

A piece of armor worn in the Boots slot.

8.7.2 Responsibilities

1. Maintain's the Boots' statistics
2. Knows it should be equipped to the Boots slot

8.7.3 Collaborators

1. EquipmentManager
2. Statistics
3. StructuredMap

8.7.4 UML class diagram

Extends Equippable

- getStats();
- save(): StructuredMap
- load(structuredMap): void

8.7.5 Implementor

Joe

8.7.6 Tester

Joe

8.8 Gloves

8.8.1 Overview

A piece of armor worn in the Gloves slot.

8.8.2 Responsibilities

1. Maintain's the Gloves' statistics
2. Knows it should be equipped to the Gloves slot

8.8.3 Collaborators

1. EquipmentManager
2. Statistics
3. StructuredMap

8.8.4 UML class diagram

Extends Equippable

- getStats();
- save(): StructuredMap
- load(structuredMap): void

8.8.5 Implementor

Joe

8.8.6 Tester

Joe

8.9 Leggings

8.9.1 Overview

A piece of armor worn in the Leggings slot.

8.9.2 Responsibilities

1. Maintain's the Leggings' statistics
2. Knows it should be equipped to the Leggings slot

8.9.3 Collaborators

1. EquipmentManger
2. Statistics
3. StructuredMap

8.9.4 UML class diagram

Extends Equippable

- getStats();
- save(): StructuredMap
- load(structuredMap): void

8.9.5 Implementor

Joe

8.9.6 Tester

Joe

8.10 Helmet

8.10.1 Overview

A piece of armor worn in the Helmet slot.

8.10.2 Responsibilities

1. Maintain's the helmets' statistics
2. Knows it should be equipped to the helmet slot

8.10.3 Collaborators

1. EquipmentManger
2. Statistics
3. StructuredMap

8.10.4 UML class diagram

Extends Equippable

- getStats();
- save(): StructuredMap
- load(structuredMap): void

8.10.5 Implementor

Joe

8.10.6 Tester

Joe

8.11 Weapon {abstract}

8.11.1 Overview

A piece of equipment worn in a weapon slot. Used for attacks. Overrides the use function in equipables.

8.11.2 Responsibilities

1. Maintain's the Weapon's statistics
2. Knows it should be equipped to the Weapon's slot

8.11.3 Collaborators

1. EquipmentManager
2. StructuredMap

8.11.4 UML class diagram

- getStats();
- getInfo():
- getAttack(); /?? or just attack()? or. . . ? (discuss)
- use(): void
- save(): StructuredMap
- load(structuredMap): void

8.11.5 Implementor

Joe

8.11.6 Tester

Joe

8.12 TwoHandedWeapon

8.12.1 Overview

A weapon requiring two hands. Specific to the Smasher occupation. A chainsaw would be an example of a TwoHandedWeapon

8.12.2 Responsibilities

1. Maintain's the TwoHandedWeapon's statistics
2. Knows it should be equipped to the TwoHandedWeapon's slot

8.12.3 Collaborators

1. EquipmentManager
2. StructuredMap

8.12.4 UML class diagram

Extends Weapon(I presume?)

- getStats();
- getAttack(); //??? discuss.
- use(): void
- save(): StructuredMap
- load(structuredMap): void

8.12.5 Implementor

Joe

8.12.6 Tester

Joe

8.13 OneHandedWeapon

8.13.1 Overview

A weapon requiring one hand. A Sword would be an example of a OneHandedWeapon

8.13.2 Responsibilities

1. Maintain's the OneHandedWeapon's statistics
2. Knows it should be equipped to the OneHandedWeapon's slot

8.13.3 Collaborators

1. EquipmentManager
2. StructuredMap

8.13.4 UML class diagram

Extends Weapon

- getStats();
- getAttack(); //??? discuss.
- use(): void
- save(): StructuredMap
- load(structuredMap): void

8.13.5 Implementor

Joe

8.13.6 Tester

Joe

8.14 BrawlingWeapon

8.14.1 Overview

A fast "weapon" (or lack thereof..?–discuss: Perhaps a smasher with no weapon automatically equips one of these?) requiring two hands. Brass Knuckles would be an example of this.

8.14.2 Responsibilities

1. Maintain's the BrawlingWeapon's statistics
2. Knows it should be equipped to the BrawlingWeapon's slot

8.14.3 Collaborators

1. EquipmentManager
2. StructuredMap

8.14.4 UML class diagram

extends Weapon

- getStats();
- getAttack(); //??? discuss.
- use(): void
- save(): StructuredMap
- load(structuredMap): void

8.14.5 Implementor

Joe

8.14.6 Tester

Joe

8.15 WeaponVisitor

8.15.1 Overview

Usedf to get the actual type of the weapon that the Entity uses. If the Weapon is of the specific type, it will use the Skills of the Entity, Brawling, THW, SignleWeapon, Staff, and Bow(Range) to decide the effectiveness of the weapon. The skill bonus is added to the offensive rating of the entity that is attacking.

8.15.2 Responsibilities

1. gets the Skill specific adder for a specific Weapon.

8.15.3 Collaborators

1. EquipmentManager

8.15.4 UML class diagram

- accept(BrawlingWeapon)
- accept(StaffWeapon)
- accept(TwoHandedWeapon)
- accept(SingleWeapon)
- accept(Bow)
- getSkillBonus():int
- save(): StructuredMap
- load(structuredMap): void

8.15.5 Implementor

Joe

8.15.6 Tester

Joe

9 Subsystem Projectiles

9.0.1 Overview

Projectiles are moving things that cause an effect when the collide with either an entity or are blocked by an impassable tile such as a mountain.

9.1 Projectile

9.1.1 Overview

Its a Projectile. It travels in a straight line from where it started it is simply a moving trigger. It keeps track of it's own time-out, determined by its speed. The timeout is used to make sure that advance only works after a certain time after the projectile last moved.

9.1.2 Responsibilities

1. Move along a trajectory
2. Contain a trigger
3. Collide with obstacles

9.1.3 Collaborators

1. Trigger
2. GameMap
3. StructuredMap

9.1.4 UML class diagram

- direction: Angle
- location: Location
- speed: double
- timeOut: long (ms) // time when advance should work again. (1/speed)
- trigger: Trigger
- hasExpired(): boolean
- advance(): void
- save(): StructuredMap
- load(structuredMap): void

9.1.5 Implementor

Jacob

9.1.6 Tester

Jacob

9.2 Conical Projectile

9.2.1 Overview

Its a Projectile. It travels in a 60° arc from where it started and it is simply a moving trigger. It keeps track of it's own time-out, which is determined by its speed. The timeout is used to make sure that advance only works after a certain time after the projectile last moved.

9.2.2 Responsibilities

1. Move along a conical trajectory by spawning other projectiles
2. Contain a trigger
3. Collide with obstacles
4. Signal when it has expired

9.2.3 Collaborators

1. Trigger
2. GameMap
3. StructuredMap

9.2.4 UML class diagram

{{ Extends Projectile }}

- direction: Angle
- location: Location
- speed: double
- timeOut: long (ms) // time when advance should work again. (1/speed)
- trigger: Trigger
- hasExpired(): boolean

/advance(): void

- save(): StructuredMap
- load(structuredMap): void

9.2.5 Implementor

Jacob

9.2.6 Tester

Jacob

9.3 Projectile Manager

9.3.1 Overview

This is another manager that is in charge of making sure that every projectile is advanced on every game tick. It removes every projectile as soon as it returns hasExpired().

9.3.2 Responsibilities

1. Advance every projectile on a game tick
2. Keep track of every projectile on the map
3. Remove projectiles as soon as they expire or are triggered.

9.3.3 Collaborators

1. Trigger
2. GameMap

9.3.4 UML class diagram

- projectiles: List<Projectile>
- addProjectile(projectile)
- update(): void // Advances all projectiles

9.3.5 Implementor

Jacob

9.3.6 Tester

Jacob

9.4 Angle

9.4.1 Overview

An enum that specifies a number of different directions. Holds the backing angle in degrees. Note: 0 degrees is right and an increasing angle goes counterclockwise

9.4.2 Responsibilities

1. Represent a possible direction in a Human-Readable format (UP, Down-Left, . . .)

9.4.3 Collaborators

None

9.4.4 UML class diagram

- theta : int (0 to 360)
- getAngle() : int
- sin(): double
- cos(): double

[UP, DOWN, UP_{RIGHT}, UP_{LEFT}, $\text{DOWN}_{\text{RIGHT}}$, $\text{DOWN}_{\text{LEFT}}$]

- save(): StructuredMap
- load(structuredMap): void

9.4.5 Implementor

Jacob

9.4.6 Tester

Jacob

10 Inventory Subsystem

10.0.1 Overview

The Equipment subsystem is used to equip and unequip items from a entity. It uses a Observer pattern to communicate with the stats.

10.1 ItemManager

10.1.1 Overview

This inventory manager will be contained inside of all entities and will encapsulate the entity's inventory and equipped inventory and will provide a nice interface for the entity to use.

10.1.2 Responsibilities

1. adding items
2. removing items
3. equipping items
4. unequipping items.

10.1.3 Collaborators

1. EquipmentInventory
2. Inventory
3. StructuredMap

10.1.4 UML class diagram

- unequip(equippable: Equippable): boolean
- equip(equippable: Equippable): boolean
- addItem(item: Item): boolean
- removeItem(item: Item): boolean
- save(): StructuredMap
- load(structuredMap): void

10.1.5 Implementor

Joe

10.1.6 Tester

Joe

10.2 Inventory

10.2.1 Overview

The inventory is used by the avatar to pick up items and use items. There will be a limited amount of items that an entity can pick up.

10.2.2 Responsibilities

1. adding items
2. removing items
3. dropping items.

10.2.3 Collaborators

1. InvenotrySlot
2. Entity
3. ItemManager
4. StructuredMap

10.2.4 UML class diagram

- slots: InvenotrySlot[][]
- addItem(item: TakeableItem): boolean
- removeItem(item: TakeableItem): boolean
- getItems(); TakeableItem[][]
- hasItem(TakeableItem): boolean
- save(): StructuredMap
- load(structuredMap): void

10.2.5 Implementor

Joe

10.2.6 Tester

Joe

10.3 InventorySlot

10.3.1 Overview

The InventorySlot is used by the avatar to pick up items and use items.

10.3.2 Responsibilities

1. adding items
2. removing items
3. dropping items.

10.3.3 Collaborators

1. InvenotrySlot
2. Entity
3. ItemManager
4. StructuredMap

10.3.4 UML class diagram

- addItem(item: TakeableItem): boolean
- removeItem(): TakeableItem
- getItem();
- hasItem(): boolean
- save(): StructuredMap
- load(structuredMap): void

10.3.5 Implementor

Joe

10.3.6 Tester

Joe

10.4 EquipmentSlot<K extends Equipable>

The slots are the equipment slots for the Equipments. Each is its own specific class, they are not derived from anything. Each is a Observer and can update the stats of the Entity that they are used by.

10.4.1 Overview

Contains a Armor equipable item. Is a Aggregate and can only equip this type of item. Every un/equip updates the stats appropriately.

10.4.2 Responsibilities

1. equip
2. unequip
3. getStatBonus
4. update the stats subjects.

10.4.3 Collaborators

1. Equipable
2. StructuredMap

10.4.4 UML class diagram

- equip(<K extends Equipable>)
- unequip(): <K extends Equipable>
- hasEquipment(): boolean
- save(): StructuredMap
- load(structuredMap): void

10.4.5 Implementor

Joe

10.4.6 Tester

Joe

10.5 DoubleEquipmentSlot

This is a Composite of the Shield and the Weapon Slot used by the Equipment. It will generate a Shield and a Weapon Slot in one.

10.5.1 Overview

Contains a Armor equipable item. Is a Aggregate and can only equip this type of item. Every un/equip updates the stats appropriately.

10.5.2 Responsibilities

1. equip a Weapon (OneHanded, TwoHanded, Brawling)
2. equip Shield
3. unequip Shield
4. uneqip Weapon (OneHanded, TwoHanded, Brawling)
5. getStatBonus
6. update stats

10.5.3 Collaborators

1. Shield
2. Weapon
3. TwoHanded
4. Equipment
5. StructuredMap

10.5.4 UML class diagram

Extends EquipmentSlot

- unequip(); TwoHandedWeapon
- has(): boolean
- unequipShield(): Shield

- unequipWeapon(): Weapon
- unequipTHW(): TwoHandedWeapon
- save(): StructuredMap
- load(structuredMap): void

10.5.5 Implementor

Joe

10.5.6 Tester

Joe

10.6 EquipmentManager

10.6.1 Overview

This EquipmentManager contains all the equipment slots that an entity can hold. It also manages the responsibility of maintaining that only one type of a piece of Equipable can be equipped at a time.

10.6.2 Responsibilities

1. Contain all the different equipment slots for all the different equipment types
2. Ensures that only one type of each item can be equipped

10.6.3 Collaborators

1. EquipmentSlot
2. Equipable
3. StructuredMap

10.6.4 UML class diagram

- equipHelmet(Helmet): void
- equipChestPiece(ChestPiece): void
- equipLeggings(Leggings): void
- equipBoots(Boots): void
- equipGloves(Gloves): void
- equipShield(Shield): void
- equipWeapon(Weapon): void
- unequipHelmet(item: Equipable): void
- unequipChestPiece(ChestPiece): void
- unequipLeggings(Leggings): void
- unequipBoots(Boots): void

- unequipGloves(Gloves): void
- unequipShield(Shield): void
- unequipWeapon(Weapon): void
- save(): StructuredMap
- load(structuredMap): void

10.6.5 Implementor

Joe

10.6.6 Tester

Joe

10.7 TradingManager

10.7.1 Overview

Allows an Avatar to trade with the NPC. It allows a transaction between the Avatar and the NPC with one party spending Gold and the other sending a TakeableItem.

10.7.2 Responsibilities

1. buy
2. sell
3. getPrice

10.7.3 Collaborators

1. NPC
2. barter
3. Avatar
4. Invenotry

10.7.4 UML class diagram

- buy(Item,Avatar)
- sell(Item,Avatar)
- getInfo(item)

10.7.5 Implementor

Joe

10.7.6 Tester

Joe

11 SubSystem Behaviors

11.0.1 Overview

Behaviors are Entity states, these can be applied by items, spells, other Behaviors, engagements, and other things.

11.1 AvatarControllerMachine

11.1.1 Overview

The Avatar ControllerMachine allows the avatar to change its control sets. It also allows the Avatar to change its Behavior state, allowing it to be controlled by NPC behaviors.

11.1.2 Responsibilities

1. sets the Controllers for the Avatar
2. can clear all the COntrollers of the Avatar, and allow it to perform NPC Behaviors.

11.1.3 Collaborators

1. Entities
2. Behavior

11.1.4 UML class diagram

- setControllers()

11.1.5 Implementor

Jacob

11.1.6 Tester

Jacob

11.2 AvatarControllerState

11.2.1 Overview

A Avatar controller is a interface for how a Avatar Controller State can change by resetting the Controllers through this controller State.

11.2.2 Responsibilities

1. Changes the Avatar Controllers

11.2.3 Collaborators

1. Avatar
2. ControllerManager

11.2.4 UML class diagram

- setControllers()

11.2.5 Implementor

Jacob

11.2.6 Tester

Jacob

11.3 NormalController

11.3.1 Overview

Allows the Avatar to have its Controllers.

11.3.2 Responsibilities

1. Changes the Avatar Controllers

11.3.3 Collaborators

1. Avatar
2. ControllerManager

11.3.4 UML class diagram

- setControllers()

11.3.5 Implementor

Jacob

11.3.6 Tester

Jacob

11.4 DefaultState

11.4.1 Overview

The NPC's normal movement or standing behavior.

11.4.2 Responsibilities

1. Performs the normal behavior of the NPC

11.4.3 Collaborators

1. Entities
2. Behavior

11.4.4 UML class diagram

{{interface}}

- perform()

11.4.5 Implementor

Jacob

11.4.6 Tester

Jacob

11.5 Stand

11.5.1 Overview

The NPC's behavior default behavior is Freeze.

11.5.2 Responsibilities

1. Freezes the NPC. They cannot move, will not attack, and cannot use spells.

11.5.3 Collaborators

1. Entities
2. Behavior

11.5.4 UML class diagram

{{implements DefaultState}}

- perform()

11.5.5 Implementor

Jacob

11.5.6 Tester

Jacob

11.6 Patrol

11.6.1 Overview

The NPC's behavior default behavior is to Patrol in a movement pattern.

11.6.2 Responsibilities

1. The NPC will move in a pattern.

11.6.3 Collaborators

1. Entities
2. Behavior

11.6.4 UML class diagram

{{implements DefaultState}}

- perform()

11.6.5 Implementor

Jacob

11.6.6 Tester

Jacob

11.7 Coward

11.7.1 Overview

If the ObserveState identifies a

11.7.2 Responsibilities

1. The NPC will move in a pattern.

11.7.3 Collaborators

1. Entities
2. Behavior

11.7.4 UML class diagram

{{implements DefaultState}}

- perform()

11.7.5 Implementor

Jacob

11.7.6 Tester

Jacob

11.8 InteractState

11.8.1 Overview

The NPC's normal interact behavior.

11.8.2 Responsibilities

1. Performs the normal interact behavior of the NPC.

11.8.3 Collaborators

1. Entities
2. Behavior

11.8.4 UML class diagram

{{interface}}

- interact(Entity)

11.8.5 Implementor

Jacob

11.8.6 Tester

Jacob

11.9 Barter

11.9.1 Overview

The NPC's interact behavior will perform a Bartering with the Avatar.

11.9.2 Responsibilities

1. Performs the normal interact behavior of the NPC.

11.9.3 Collaborators

1. Entities
2. Behavior

11.9.4 UML class diagram

{{implements Interact}}

- interact(Entity)

11.9.5 Implementor

Jacob

11.9.6 Tester

Jacob

11.10 Mount

11.10.1 Overview

The NPC's interact behavior will allow the Avatar to Mount the Mount.

11.10.2 Responsibilities

1. Performs the normal interact behavior of the NPC.

11.10.3 Collaborators

1. Entities
2. Behavior

11.10.4 UML class diagram

{{implements Interact}}

- interact(Entity)

11.10.5 Implementor

Jacob

11.10.6 Tester

Jacob

11.11 Talk

11.11.1 Overview

The NPC's interact behavior will allow the Avatar to Talk to it.

11.11.2 Responsibilities

1. Talks

11.11.3 Collaborators

1. Entities
2. Behavior

11.11.4 UML class diagram

{{implements Interact}}

- interact(Entity)

11.11.5 Implementor

Jacob

11.11.6 Tester

Jacob

11.12 Attack

11.12.1 Overview

The NPC's interact behavior will allow the NPC to attack on interactions.

11.12.2 Responsibilities

1. Attack

11.12.3 Collaborators

1. Entities
2. Behavior

11.12.4 UML class diagram

{{implements Interact}}

- interact(Entity)

11.12.5 Implementor

Jacob

11.12.6 Tester

Jacob

11.13 ObserveState

11.13.1 Overview

The NPC's observing.

11.13.2 Responsibilities

1. Performs the normal behavior of the NPC

11.13.3 Collaborators

1. Entities
2. Behavior

11.13.4 UML class diagram

{{interface}}

- observe()

11.13.5 Implementor

Jacob

11.13.6 Tester

Jacob

11.14 SightTracking

11.14.1 Overview

The NPC's normal movement or standing behavior.

11.14.2 Responsibilities

1. Allows the NPC to notice other Entities, primarly the Avatar.

11.14.3 Collaborators

1. Entities
2. Behavior

11.14.4 UML class diagram

{{implements ObserverState}}

- observe()

11.14.5 Implementor

Jacob

11.14.6 Tester

Jacob

11.15 NPCBehaviorable

11.15.1 Overview

The NPCbehavior class belongs to NPCs and defines 3 states: default, interact, and observe that defines their actions based on an event. It also has a behavior that gets pushed to the specific entity that this behavior belongs to on attack. This new attack behavior defines a new set of states that define different actions for the entity to take.

11.15.2 Responsibilities

1. To be performed.
2. To interact
3. To Observe
4. onAttack

11.15.3 Collaborators

1. NPC
2. DefaultState
3. InteractState
4. ObserveState

11.15.4 UML class diagram

{{interface}}

- perform()
- interact(Entity)
- observe()
- onAttack()
- onExit()
- onEnter()

11.15.5 Implementor

Jacob

11.15.6 Tester

Jacob

11.16 Behavior

11.16.1 Overview

A NPC specific behavior is very complex, as it defines the AI of a NPC. They can be as complex as you want to make them. Given these infinite possibilities, we will make a behavior have 3 states interact, default, and observe and the given onAttack Behavior change and the onObserve behavior change. So basically a Behavior is comprised of 3 states and 2 possible Behaviors that can be activated within. The two other behaviors will allow the NPC to change state at any time.

11.16.2 Responsibilities

1. Performs the default state
2. observes from the observe state
3. Interacts from the interact State
4. Changes Behavior on Observeing
5. Canges Behavior on Attack

11.16.3 Collaborators

1. Entities
2. NPCStateMachine

11.16.4 UML class diagram

{{implements Behaviorable}}

- perform()
- interact(Entity)
- observe()

11.16.5 Implementor

Jacob

11.16.6 Tester

Jacob

11.17 NPCStateMachine

11.17.1 Overview

A Preferred behavior is passed to the State to perform that behavior. The entity will continue to perform the peek behavior that it has, until the behavior is timed out or if the state pushes another behavior onto the top. Used by all Entities to control their state. Manages the killing of behaviors, and the activation of behaviors

11.17.2 Responsibilities

1. push states on
2. perform a state
3. revert a state
4. kill all states

11.17.3 Collaborators

1. Entities
2. Abilities
3. Effects

11.17.4 UML class diagram

- perform()
- interact(Entity)
- observe()
- push(Behavior)
- pop():Behavior

11.17.5 Implementor

Jacob

11.17.6 Tester

Jacob

12 Subsystem Light

12.0.1 Overview

The light subsystem implements and manages the fog of war. This system will have a LightMap that maintains a visibility level for every tile and is responsible for raising the visibility level of a location when a light source is present and lowering the visibility level of unseen locations. This LightMap will be managed by the LightManager which will be the class that coordinates the whole operation on every game tick.

12.1 LightManager

12.1.1 Overview

The light manager coordinates the LightMap. It maintains all the light sources currently registered and tells the light map to illuminate based on the light sources that the manager has registered.

12.1.2 Responsibilities

1. Keep track of all the light sources currently on the map
2. Tell the lightmap to dim all its lights every game update
3. Tell the game map to illuminate an area around a lightsource

12.1.3 Collaborators

1. Light Map
2. LightSource
3. StructuredMap

12.1.4 UML class diagram

- lightSources: List<LightSources>
- lightMap: LightMap
- update()
- save(): StructuredMap
- load(structuredMap): void

12.1.5 Implementor

Matt

12.1.6 Tester

Matt

12.2 LightMap

12.2.1 Overview

The light map maintains the visibility levels of all the locations on the map and maintains if the tile has been visited before. This will be used by the view to determine when to draw an entity on a map and also when to gray out the tile. The LightManager will be in charge of dimming all the lights on the whole map when instructed and illuminating (increasing the visibility level of a tile). This allows the lights to slowly dim when the area is not illuminated.

12.2.2 Responsibilities

1. Keep track of the visibility and isVisited attributes for every location on the map
2. Dim all the visibilities on command
3. Raise the visibilities of all the locations that are illuminated by a given light source

12.2.3 Collaborators

1. Light Source
2. StructuredMap

12.2.4 UML class diagram

- visiblities: Visiblities[][]
- dimLights(): void
- illuminate(lightSource): void
- save(): StructuredMap
- load(structuredMap): void

12.2.5 Implementor

Matt

12.2.6 Tester

Matt

12.3 LightSource

12.3.1 Overview

The LightSource defines a geometry that is illuminated by a lightsource.

12.3.2 Responsibilities

1. A single point of light on the map

12.3.3 Collaborators

1. StructuredMap

12.3.4 UML class diagram

{{abstract}}

- visiblitie: Visiblitie
- dimLight(): void
- save(): StructuredMap
- load(structuredMap): void

12.3.5 Implementor

Matt

12.3.6 Tester

Matt

12.4 Static Light Source

12.4.1 Overview

A source of light that does not move.

12.4.2 Responsibilities

12.4.3 Collaborators

1. StructuredMap

12.4.4 UML class diagram

{{extends Light source}}

12.4.5 Implementor

Matt

12.4.6 Tester

Matt

12.5 Dynamic Light Source

12.5.1 Overview

A source of light that does move

12.5.2 Responsibilities

12.5.3 Collaborators

1. StructuredMap

12.5.4 UML class diagram

{{extends Light source}}

- move(. . .):void

12.5.5 Implementor

Matt

12.5.6 Tester

Matt

13 Subsystem Statistics

13.0.1 Overview

Statistics are used to represent the power and skill of an entity or item.

13.1 Statistics

13.1.1 Overview

Represent the power and skill of various entities and items.

13.1.2 Responsibilities

1. Hold values for each of the different properties that are being tracked.
2. Get the derived properties at a current instant in time

13.1.3 Collaborators

1. Equipped Inventory
2. Equippable Items
3. StructureMap

13.1.4 UML class diagram

- Strength
- Agility
- etc...
- save(): StructuredMap
- load(structuredMap): void

13.1.5 Implementor

JR

13.1.6 Tester

JR

13.2 EntityStatistics

13.2.1 Overview

Represent the power and skills of an entity. This subclass contains extra information relevant to entities, like current Health and current Mana

13.2.2 Responsibilities

1. Hold values for each of the different entity specific properties that are being tracked.
2. Get the derived properties at a current instant in time.
3. Keep track of the extra information (currentHealth, currrentMana)

13.2.3 Collaborators

1. Equipped Inventory
2. Equippable Items
3. StructuredMap

13.2.4 UML class diagram

Extends Statistics

- CurrentHealth
- CurrentMana
- Experience
- Lives Left
- getOffensiveRating(): int
- getDefensiveRating(): int
- getArmorRating(): int
- save(): StructuredMap
- load(structuredMap): void

13.2.5 Implementor

JR

13.2.6 Tester

JR

14 Subsystem State Machinery

14.0.1 Overview

Much of the game is driven by a number of state machines. These state machines allow states to be pushed, popped, and swapped. There are 3 subcategories of states: GameStates, which handle transitions between menus, NPCStates, which determine entity behavior, and AvatarStates which modify an Avatar's behavior.

14.1 StateMachine<T extends State>

14.1.1 Overview

The StateMachine is a simple container class that allows states to be pushed, popped, and swapped. It calls template methods on each state for when states are first entered, paused, resumed, and exited–modelled much like Activities in android.

14.1.2 Responsibilities

1. Contain a stack of states
2. Provide ways for states to transition
3. Call the appropriate methods on a state as they are pushed/popped/swapped.

14.1.3 Collaborators

1. State

14.1.4 UML class diagram

- push(T): void
- swap(T): void
- pop(): void

14.1.5 Implementor

JR

14.1.6 Tester

JR

14.2 State

14.2.1 Overview

State is an interface for providing uniform functionality to specific kinds of states.

14.2.2 Responsibilities

1. Provide hooks for State implementors to perform appropriate behavior on state transitions.
2. Handle StateTransitions

14.2.3 Collaborators

1. StateMachine

14.2.4 UML class diagram

- onEnter(): void
- onPause(): void
- onResume(): void
- onExit(): void

14.2.5 Implementor

JR

14.2.6 Tester

JR

14.3 SubSubSystem GameStates

1. Overview Menu transitions & popup dialogues are handled by game state transitions. These are all held by a StateMachine<GameState> within the model.

14.3.1 GameState

(a) Overview A game state is associated with the current view layout & controller. As game states are transitioned, we are generally moving from menu to menu.

(b) Responsibilities

i. Present a new view layout (or overlay, in the case of inventory/shops/etc)

ii. Setup control logic for the current view

(c) Collaborators

i. StateMachine<GameState>

(d) UML class diagram implements State

- onEnter(): void //initialize and attach a view layout object
- onPause(): void
- onResume(): void
- onExit(): void

(e) Implementor JR

(f) Tester JR

14.3.2 MainMenuState

(a) Overview The game state associated with the Main Menu.

(b) Responsibilities

  i. Register the MainMenuLayout in the view.

  ii. Instantiate and register the MainMenuController.

(c) Collaborators

  i. StateMachine<GameState>

(d) UML class diagram extends GameState

(e) Implementor JR

(f) Tester JR

14.3.3 CharacterSelectionMenuState

(a) Overview The game state associated with a new game in which a player selects a new character. Can be reached only from the Main Menu.

(b) Responsibilities

  i. Register the CharacterSelectionMenuLayout in the view.

  ii. Instantiate and register the CharacterSelectionMenuController.

(c) Collaborators

  i. StateMachine<GameState>

(d) UML class diagram extends GameState

(e) Implementor JR

(f) Tester JR

14.3.4 LoadGameMenuState

(a) Overview The game state associated with the "Load Game" menu. Can be reached from the Main Menu and from the Pause Menu.

(b) Responsibilities

  i. Register the LoadGameMenuLayout in the view.

  ii. Instantiate and register the LoadGameMenuController.

(c) Collaborators

  i. StateMachine<GameState>

(d) UML class diagram extends GameState

(e) Implementor JR

(f) Tester JR

14.3.5 SaveGameMenuState

(a) Overview The game state associated with the "Save Game" menu. Can only be reached from the Pause Menu.

(b) Responsibilities

   i. Register the SaveGameMenuLayout in the view.

   ii. Instantiate and register the LoadGameMenuController.

(c) Collaborators

   i. StateMachine<GameState>

(d) UML class diagram extends GameState

(e) Implementor JR

(f) Tester JR

14.3.6 OptionsMenuState

(a) Overview The game state associated with an open options menu. Options can be accessed from the Pause Menu.

(b) Responsibilities

  i. Register the OptionsMenuLayout in the view.

  ii. Instantiate and register the PauseMenuController.

(c) Collaborators

  i. StateMachine<GameState>

(d) UML class diagram extends GameState

(e) Implementor JR

(f) Tester JR

14.3.7 GameplayState

(a) Overview The game state associated with actual gameplay. Can be entered only after loading a game which occurs after the LoadGameMenu state or after the CharacterSelectionMenu state (which also loads a preconfigured game save).

(b) Responsibilities

i. Register the GameplayLayout in the view.

ii. Instantiate and register the GameplayController.

iii. Accepts Views from the Model, and pushes them up to the GameplayLayout

(c) Collaborators

i. StateMachine<GameState>

(d) UML class diagram extends GameState

- add/registerItemView(itemView: ItemView)
- add/registerEntityView(entityView: EntityView)
- add/registerLightView(lightView: LightView)
- add/registerTileView(TileView: TileView)

(e) Implementor JR

(f) Tester JR

14.3.8 PauseMenuState

(a) Overview The game state associated with an open Pause Menu. Acts as a pop-up, rather than completely replacing the previous view. Provides access to other menus–Load Game, Save Game, Options, Main Menu, and a Resume Gameplay button that takes you back to gameplay.

(b) Responsibilities

i. Register the PauseMenuLayout in the view.

ii. Instantiate and register the PauseMenuController.

(c) Collaborators

i. StateMachine<GameState>

(d) UML class diagram extends GameState

- onEnter(): void //overrides normal view behavior to overlay instead
- onExit(): void //removes popup layer

(e) Implementor JR

(f) Tester JR

14.3.9 InventoryMenuState

(a) Overview The game state associated with an open Inventory Menu. Acts as a pop-up, rather than completely replacing the previous view.

(b) Responsibilities

i. Register the InventoryMenuLayout in the view.

ii. Instantiate and register the InventoryMenuController.

(c) Collaborators

i. StateMachine<GameState>

(d) UML class diagram extends GameState

- onEnter(): void //overrides normal view behavior to overlay instead
- onExit(): void //removes popup layer

(e) Implementor JR

(f) Tester JR

14.3.10 TradeMenuState

(a) Overview The game state associated with an open trading menu. Acts as a pop-up, rather than completely replacing the previous view.

(b) Responsibilities

i. Register the TradeMenuLayout in the view.

ii. Instantiate and register the TradeMenuController.

(c) Collaborators

i. StateMachine<GameState>

(d) UML class diagram extends GameState

- onEnter(): void //overrides normal view behavior to overlay instead
- onExit(): void //removes popup layer

(e) Implementor JR

(f) Tester JR

14.3.11 SkillsMenuState

(a) Overview The game state associated with an open skills menu. Acts as a pop-up, rather than completely replacing the previous view.

(b) Responsibilities

i. Register the SkillsMenuLayout in the view.

ii. Instantiate and register the SkillsMenuController.

(c) Collaborators

i. StateMachine<GameState>

(d) UML class diagram extends GameState

- onEnter(): void //overrides normal view behavior to overlay instead
- onExit(): void //removes popup layer

(e) Implementor JR

(f) Tester JR

15 SubSystem Loading / Saving

15.0.1 Overview

This system covers utility classes for loading and saving the state of the game. We use a JSON parser to produce a helper object of type StructuredMap. The StructuredMap's role is to provided typed access to saved data from the JSON.

15.1 StructuredMap

15.1.1 Overview

StructuredMaps are simply hashmaps of String to a structured map type that provide typed access to the different objects they hold.

15.1.2 Responsibilities

(a) Act as a DAO.

15.1.3 Collaborators

None.

15.1.4 UML class diagram

- put(): void
- getString(key : String): String
- getDouble(key : String): Double
- getInteger(key : String): Integer
- getBoolean(key : String): Boolean
- getStructuredMap(key : String): StructuredMap
- getStructuredMapArray(key : String): StructuredMap[]
- getIntArray(key : String): int[]
- getDoubleArray(key : String): double[]
- getKeys(): Set<String>

15.1.5 Implementor

Daniel

15.1.6 Tester

Daniel

15.2 JSONParser

15.2.1 Overview

This is a utility class that parses JSON. It converts a JSONToken stream into a StructuredMap.

15.2.2 Responsibilities

(a) Parse JSON and return a StructuredMap

15.2.3 Collaborators

(a) JSONScanner
(b) JSONToken

15.2.4 UML class diagram

- getStructuredMapFromJSON(String : json): StructuredMap

15.2.5 Implementor

Daniel

15.2.6 Tester

Daniel

15.3 JSONScanner

15.3.1 Overview

This class simply tokenizes JSON.

15.3.2 Responsibilities

(a) Tokenize JSON.

15.3.3 Collaborators

(a) JSONScanner
(b) JSONToken

15.3.4 UML class diagram

- getTokens(String : json): Queue<JSONToken>

15.3.5 Implementor

Daniel

15.3.6 Tester

Daniel

15.4 JSONToken

15.4.1 Overview

This is an enum of all valid JSON tokens.

15.4.2 Responsibilities

(a) Represent JSON tokens.

15.4.3 Collaborators

None.

15.4.4 UML class diagram

[LBRACE, RBRACE, LBRACKET, RBRACKET, COLON, COMMA, TRUE, FALSE, NULL, STRING, DOUBLE, INTEGER]

15.4.5 Implementor

Daniel

15.4.6 Tester

Daniel

15.5 JSONFormatter

15.5.1 Overview

This class formats a StructuredMap into a String. Produces proper, tabbed, printable, JSON.

15.5.2 Responsibilities

(a) Convert a StructuredMap into a String.

15.5.3 Collaborators

None.

15.5.4 UML class diagram

- getStringRepresentation(StructuredMap): String

15.5.5 Implementor

Daniel

15.5.6 Tester

Daniel

16 SubSystem Dialog

16.0.1 Overview

This is the Dialog that we can Log to the system to show on the DialogLayout. Use this to talk to the Avatar.

17 Subsystem View

17.0.1 Overview:

The view is what the user interacts with. It will show a visible representation of a portion of model on the screen and will provide a panel for the controller to attach to and listen for key presses. This subsystem will consist of individual views that will be swapped out at runtime depending on the state of the game.

17.1 Dialog

17.1.1 Overview

This is the model that communicates and updates the ViewDialog. Takes input, and output. It is a Singleton.

17.1.2 Responsibilities

(a) write

(b) read

17.1.3 Collaborators

(a) Everything can be

17.1.4 UML class diagram

- print(String)

17.1.5 Implementor

Daniel

17.1.6 Tester

Daniel

18 Subsystem Views

18.0.1 Overview

the Views comprise of the visual representation of the model. If the view was taken out and replaced with another it would work just fine, as long as it adhears to the models update methods to cahnge the view. The View is connected with Controllers that allow the user tho interact with the Views and pass messages to the model to change the state of the game.

18.1 Subsystem View Layouts

18.1.1 Overview

Layouts are the top level containers associated with a particular menu / GameState. Layouts extend JPanel, and thus can display different views. Some Layouts (such as Layouts with menus), can be decorated with JComponents (or ViewComponents). Menu Layouts use the Mouse to advance to the next State and Layout.

18.2 MainMenuLayout

18.2.1 Overview

Constitutes the entirety of the view for the MainMenu. Uses MouseEvents / JButtons / ViewComponents to navigate.

18.2.2 Responsibilities

(a) Contain all the components in the MainMenu

18.2.3 Collaborators

(a) TextLabel (Main Menu)
(b) MenuButton (New Game, Load Game, Exit Game)
(c) Calculate the size & location of each contained component

18.2.4 UML class diagram

extends JPanel

18.2.5 Implementor

Daniel

18.2.6 Tester

Daniel

18.3 PauseMenuLayout

18.3.1 Overview

A layout. Game Paused. Handles the game paused view.

18.3.2 Responsibilities

(a) Contain all the components in the Pause Menu

18.3.3 Collaborators

(a) TextLabel (Main Menu)
(b) MenuButton (Options, Save State, Return to Main Menu)
(c) Calculate the size & location of each contained component

18.3.4 UML class diagram

extends JPanel

18.3.5 Implementor

Daniel

18.3.6 Tester

Daniel

18.4 CharacterSelectionLayout

18.4.1 Overview

Constitutes the entirety of the view for the new character selection screen.

18.4.2 Responsibilities

(a) Contain all the components in the CharacterSelectionMenu.

(b) Calculate the size & location of each contained component

18.4.3 Collaborators

(a) TextLabel (Select a Character)

(b) CharacterButton (Smasher, Summoner, Sneak)

(c) MenuButton (Back)

18.4.4 UML class diagram

extends JPanel

18.4.5 Implementor

Daniel

18.4.6 Tester

Daniel

18.5 LoadMenuLayout

18.5.1 Overview

Constitutes the entirety of the view for the LoadMenu.

18.5.2 Responsibilities

(a) Contain all the components in the LoadMenu
(b) Calculate the size & location of each contained component

18.5.3 Collaborators

(a) TextLabel (Select a Game to Load)
(b) SaveSlotButton (1-5)
(c) MenuButton (Back, Load)

18.5.4 UML class diagram

extends JPanel

18.5.5 Implementor

Daniel

18.5.6 Tester

Daniel

18.6 InventoryMenuLayout

18.6.1 Overview

Constitutes the entirety of the view for the Inventory Menu.

18.6.2 Responsibilities

(a) Contain all the components in the Inventory Menu
(b) Calculate the size & location of each contained component

18.6.3 Collaborators

(a) StatisticsView
(b) InventoryView
(c) EquipmentView
(d) MenuButton (Back)

18.6.4 UML class diagram

extends JPanel

18.6.5 Implementor

Daniel

18.6.6 Tester

Daniel

18.7 SkillsMenuLayout

18.7.1 Overview

Constitutes the entirety of the view for the Skills Menu.

18.7.2 Responsibilities

(a) Contain all the components in the Skills Menu
(b) Calculate the size & location of each contained component

18.7.3 Collaborators

(a) SkillBarView (One for each skill)
(b) MenuButton (Back)
(c) PlusButton (One for each skill)
(d) TextLabel (Skills)

18.7.4 UML class diagram

extends JPanel

18.7.5 Implementor

Daniel

18.7.6 Tester

Daniel

18.8 SaveMenuLayout

18.8.1 Overview

Constitutes the entirety of the view for the LoadMenu.

18.8.2 Responsibilities

(a) Contain all the components in the LoadMenu
(b) Calculate the size & location of each contained component

18.8.3 Collaborators

(a) TextLabel (Select a Game to Load)
(b) SaveSlotButton (1-5)
(c) MenuButton (Back, Save)

18.8.4 UML class diagram

extends JPanel

18.8.5 Implementor

Daniel

18.8.6 Tester

Daniel

18.9 GameplayLayout

18.9.1 Overview

Constitutes the entirety of the view for the Gameplay view. The Gameplay layout appropriately layers the different views that comprise it. It will first draw the GameMapView, then the LightMapView, then the EntityView and then the HUDView. It only updates things in the LightMapView.

18.9.2 Responsibilities

(a) Contain all the components in the Gameplay view

(b) Calculate the size & location of each contained component

18.9.3 Collaborators

(a) GameMapView

(b) LightMapView

(c) EntityView

(d) HUDView

18.9.4 UML class diagram

extends JPanel

- add/registerItemView(itemView: ItemView)
- add/registerEntityView(entityView: EntityView)
- add/registerLightView(lightView: LightView)
- add/registerTileView(TileView: TileView)

18.9.5 Implementor

Daniel

18.9.6 Tester

Daniel

18.10 AbilityLayout

18.10.1 Overview

Displays the Current Abilities of the Avatar in any state. The abilities in this Layout shade and show the respective time left on the Ability around the cooldown of the Ability view.

18.10.2 Responsibilities

(a) Contains the View of the Abilities for the User to interact with.

18.10.3 Collaborators

(a) GameView

18.10.4 UML class diagram

extends JPanel

- add(Ability)

18.10.5 Implementor

Daniel

18.10.6 Tester

Daniel

18.11 DialogLayout

18.11.1 Overview

Displays the information that the user inputs and the surrounds write to the Dialog in the Model.

18.11.2 Responsibilities

(a) Responsible for displaying the Dialog Model writes.

18.11.3 Collaborators

(a) DialogLog

18.11.4 UML class diagram

extends JPanel

- add(String)

18.11.5 Implementor

Daniel

18.11.6 Tester

Daniel

18.12 TradingView

18.12.1 Overview

Displays the trading of a Barter with the Avatar. This view shows the items that an Avatar can buy or sell to the Barter.

18.12.2 Responsibilities

(a) Responsible for the trading and selling of the items.

18.12.3 Collaborators

(a) DialogLog

18.12.4 UML class diagram

extends JPanel

- addBarter(item)
- addCustomer(item)

18.12.5 Implementor

Daniel

18.12.6 Tester

Daniel

18.13 OptionAndControlsLayout

18.13.1 Overview

Displays a the list of options and Controllers for the User to adjust to their key bindings.

18.13.2 Responsibilities

(a) Display each Controller and Options.

18.13.3 Collaborators

(a) Controllers

18.13.4 UML class diagram

- set(String[]);

18.13.5 Implementor

Daniel

18.13.6 Tester

Daniel

19 Subsystem View Components

19.0.1 Overview

Reusable view components used across various other views & layouts.

19.1 TextLabel

19.1.1 Overview

Provide a view for displaying text that is properly themed with the rest of the game.

19.1.2 Responsibilities

(a) Display text with a consistent theme

19.1.3 Collaborators

None.

19.1.4 UML class diagram

19.1.5 Implementor

Daniel

19.1.6 Tester

Daniel

19.2 MenuButton

19.2.1 Overview

A text-based button. Provides a consistent look and feel across the game.

19.2.2 Responsibilities

(a) Display menu buttons with a consistent theme

19.2.3 Collaborators

None.

19.2.4 UML class diagram

19.2.5 Implementor

Daniel

19.2.6 Tester

Daniel

19.3 IncrementButton

19.3.1 Overview

A graphical button used for incrementing skills, volume, etc. To be paired with slider-type components.

19.3.2 Responsibilities

(a) Act as a button with a plus sign on it.

19.3.3 Collaborators

None.

19.3.4 UML class diagram

19.3.5 Implementor

Daniel

19.3.6 Tester

Daniel

19.4 DecrementButton

19.4.1 Overview

A graphical button used for decrementing volume, etc. To be paired with slider-type components.

19.4.2 Responsibilities

(a) Act as a button with a minus sign on it.

19.4.3 Collaborators

None.

19.4.4 UML class diagram

19.4.5 Implementor

Daniel

19.4.6 Tester

Daniel

19.5 StatBar

19.5.1 Overview

A graphical element for displaying a bar that is filled proportionally to the maximum possible value. (ex: health bar, mana bar)

19.5.2 Responsibilities

(a) Display a stat proportional to its maximum value as a bar.

19.5.3 Collaborators

None.

19.5.4 UML class diagram

19.5.5 Implementor

Daniel

19.5.6 Tester

Daniel

20 SubSystem Basic Views

20.0.1 Overview

Something which things that are visible in the model contain, which they can use to tell the view to update, following MVC where the model tells the view to update, and the model is pushed by the controller. Subclassed into EntityView, and so forth. While the model contains references to this, this is distinctly separate from the model, as the model mustn't know how to render itself. This is the interface which the basic things in the model uses to push to the view.

20.1 View

20.1.1 Overview

Something which the model calls to update itself to the viewable display of the User. It knows how to draw the model of which it represents, and it writes it to the Graphics. Once done, it does not need to return anything; The graphics has been written to, and the remaining views can continue to write to that Graphics. Once the Graphics has completed writing to all the views, the graphics is used in the paintComponent(Graphics g) method of the Layout of which it is being managed by, then finally presented to the User.

20.1.2 Responsibilities

(a) Knows how to draw the piece of the model that calls it.

20.1.3 Collaborators

(a) (a model thing)

20.1.4 UML class diagram

{{abstract}}

- render(g: Graphics)

20.1.5 Implementor

Daniel

20.1.6 Tester

Daniel

20.2 EntityView

20.2.1 Overview

Knows how to draw render the Entity. Entity sends this messages to indicate what kind of Entity image to display. An entity in the Frozen state will send this a message called displayFrozenEntity() that will then request that the Entity View draws a different model.

20.2.2 Responsibilities

(a) Knows how to draw the piece of the model that calls it.

20.2.3 Collaborators

(a) Entity

20.2.4 UML class diagram

extends View

- displayFrozenEntity(gameX: int, gameY: int) // &c.

20.2.5 Implementor

Daniel

20.2.6 Tester

Daniel

20.3 TileView

20.3.1 Overview

It renders a specific tile.

20.3.2 Responsibilities

Knows how to draw the piece of the model that calls it.

20.3.3 Collaborators

Tile

20.3.4 UML class diagram

extends View +displayLavaTile(gameX: int, gameY: int) // etc.

20.3.5 Implementor

Daniel

20.3.6 Tester

Daniel

20.4 ItemView

20.4.1 Overview

Knows how to draw a specific item.

20.4.2 Responsibilities

Knows how to draw the piece of the model that calls it.

20.4.3 Collaborators

Item

20.4.4 UML class diagram

extends View

20.4.5 Implementor

Daniel

20.4.6 Tester

Daniel

20.5 Decal

20.5.1 Overview

Some triggers have an associated 'Decal' with their display. Example images of decals are (skull & crossbones, heart, and red cross). These are present for the ViewTriggerDecorator to display the appropriate graphic.

20.5.2 Responsibilities

(a) Maintains a TriggerView
(b) Forwards behavioral requests to the Trigger it wraps

20.5.3 Collaborators

(a) TriggerView
(b) StructuredMap

20.5.4 UML class diagram

- image: Image // doesn't have to be an Image, could be any graphic?
- getImage(): Image

20.5.5 Implementor

Daniel

20.5.6 Tester

Daniel

20.6 TriggerView

20.6.1 Overview

Represents a specific trigger in the view.

20.6.2 Responsibilities

(a) Knows how to draw the piece of the model that calls it.

20.6.3 Collaborators

(a) ViewableTriggerDecorator

20.6.4 UML class diagram

extends View

- update()
- getImage():img

20.6.5 Implementor

Daniel

20.6.6 Tester

Daniel

20.7 InventoryView

20.7.1 Overview

Contains multiple specialized buttons that render an Item view. Presents the Inventory of a Avatar.

20.7.2 Responsibilities

(a) Display an entitity's inventory.

(b) Have buttons for each item

20.7.3 Collaborators

(a) Inventory

(b) Button

(c) TextLabel (Inventory)

20.7.4 UML class diagram

20.7.5 Implementor

Daniel

20.7.6 Tester

Daniel

20.8 GameMapView

20.8.1 Overview

GameMapView is the view that is responsible for combining all the hexagonal tiles, the Items, and the Entities.

20.8.2 Responsibilities

(a) Display the GameMap model in the Graphics

20.8.3 Collaborators

(a) TileView
(b) ItemView
(c) EntityView
(d) LightView

20.8.4 UML class diagram

- add/registerItemView(itemView: ItemView)
- add/registerEntityView(entityView: EntityView)
- add/registerLightView(lightView: LightView)
- add/registerTileView(TileView: TileView)

20.8.5 Implementor

Daniel

20.8.6 Tester

Daniel

20.9 EquipmentView

20.9.1 Overview

Contains the views for the slots that the Equipment that the Avatar uses.

20.9.2 Responsibilities

(a) Shows each piece of equipment in an EquipmentManager.
(b) Provides buttons for each element in the EquipmentManager that can be pressed.

20.9.3 Collaborators

(a) EquipmentManager
(b) ItemView
(c) TextLabel (Equipment)

20.9.4 UML class diagram

- updateArmorSlot(Armor);
- updateHelmetSlot(Helmet);
- updateWeaponSlot(Weapon);
- updateBootsSlot(Boots);
- updateGlovesSlot(Gloves);
- updateLeggingsSlot(Leggings);
- updateProjectileSlot(Projectile);
- updateShield(Shield);

20.9.5 Implementor

Daniel

20.9.6 Tester

Daniel

21 Subsystem Camera

21.0.1 Overview

This subsystem represents the portion of the model that the user is currently able to see. This may be centered on the location of the avatar, or it may be detached from the avatar and moved around freely. This subsystem is only active when the game is in the gameplay state.

21.1 Camera

21.1.1 Overview

Represents the open view of the player.

21.1.2 Responsibilities

(a) Limits rendering views to the appropriate viewing area.
(b) Forwards rendering requests to all non-HUD view components.

21.1.3 Collaborators

(a) GameMapView
(b) EntityView

21.1.4 UML class diagram

- update(Location): void

21.1.5 Implementor

Daniel

21.1.6 Tester

Daniel

22 Subsystem Stats and other Views

22.0.1 Overview

This is the subsystem of views that the player can see his stats and other things.

22.1 StatsView

22.1.1 Overview

Represents the view of the avatars Stats.

22.1.2 Responsibilities

(a) Limits rendering views to the appropriate viewing area.
(b) Forwards rendering requests to all non-HUD view components.

22.1.3 Collaborators

(a) GameMapView
(b) EntityView

22.1.4 UML class diagram

- updateStrength()
- ... other stats

22.1.5 Implementor

Daniel

22.1.6 Tester

Daniel

23 Subsystem Controller

23.0.1 SubSystem Overview

Controllers are to represent the 'C' in the MVC Pattern. They are to listen and interpret input, and send it along to the Model. Each State in the game has an associated Controller to set the action listeners and commands that the user will call on to change the state of the model. The controller allows is the only form of mediary for the User to communicate with the model.

23.0.2 Diagram

23.1 MainMenuController

23.1.1 Overview

The controller associated with the Main Menu.

23.1.2 Responsibilities

(a) Handle control logic for the Main Menu.

23.1.3 Collaborators

(a) MainMenuLayout

23.1.4 UML class diagram

- setLayout(JPanel) : void
- newGame(): void
- loadGame(): void
- exitGame(): void
- Model

23.1.5 Implementor

Kyle

23.1.6 Tester

Kyle

23.2 CharacterSelectionMenuController

23.2.1 Overview

The controller associated with the Character Selection Menu.

23.2.2 Responsibilities

(a) Handle control logic for the Character Selection screen.

23.2.3 Collaborators

(a) CharacterSelectionLayout

23.2.4 UML class diagram

- setLayout(JPanel) : void
- selectSmasher(): void
- selectSummoner(): void
- selectSneak(): void
- goBack(): void
- Model

23.2.5 Implementor

Kyle

23.2.6 Tester

Kyle

23.3 LoadMenuController

23.3.1 Overview

The controller associated with the Load Menu.

23.3.2 Responsibilities

(a) Handle control logic for the Load screen.

23.3.3 Collaborators

(a) LoadMenuLayout

23.3.4 UML class diagram

- setLayout(JPanel) : void
- selectSlot(int): void
- loadGame(): void
- goBack(): void
- Model

23.3.5 Implementor

Kyle

23.3.6 Tester

Kyle

23.4 SaveMenuController

23.4.1 Overview

The controller associated with the Save Menu.

23.4.2 Responsibilities

(a) Handle control logic for the Save screen.

23.4.3 Collaborators

(a) SaveMenuLayout

23.4.4 UML class diagram

- setLayout(JPanel) : void
- selectSlot(int): void
- saveGame(): void
- goBack(): void
- Model

23.4.5 Implementor

Kyle

23.4.6 Tester

Kyle

23.5 PauseMenuController

23.5.1 Overview

The controller associated with the Pause Menu.

23.5.2 Responsibilities

(a) Handle control logic for the Pause screen.

23.5.3 Collaborators

(a) PauseMenuLayout

23.5.4 UML class diagram

- setLayout(JPanel) : void
- selectOptions(): void
- selectSaveGame(): void
- selectLoadGame(): void
- selectResume(): void
- selectMainMenu(): void
- Model

23.5.5 Implementor

Kyle

23.5.6 Tester

Kyle

23.6 SkillsMenuController

23.6.1 Overview

The controller associated with the Skills Menu.

23.6.2 Responsibilities

(a) Handle control logic for the Skills screen.

23.6.3 Collaborators

(a) SkillsMenuLayout

23.6.4 UML class diagram

- setLayout(JPanel) : void
- incrementSkill(int): void
- goBack(): void
- Model

23.6.5 Implementor

Kyle

23.6.6 Tester

Kyle

23.7 InventoryMenuController

23.7.1 Overview

The controller associated with the Inventory Menu.

23.7.2 Responsibilities

(a) Handle control logic for the Inventory screen.

23.7.3 Collaborators

(a) InventoryMenuLayout

23.7.4 UML class diagram

- setLayout(JPanel) : void
- selectItem(int): void
- goBack(): void
- Model

23.7.5 Implementor

Kyle

23.7.6 Tester

Kyle

23.8 GameplayController

23.8.1 Overview

The controller for the main gameplay.

23.8.2 Responsibilities

(a) Encapsulate the individual controllers that make up the control of the game.

23.8.3 Collaborators

(a) GameplayLayout
(b) MountController
(c) AvatarController
(d) CameraController
(e) Model

23.8.4 UML class diagram

- getMountController(): MountController
- getAvatarController(): AvatarController
- getCameraController(): CameraController

23.8.5 Implementor

Kyle

23.8.6 Tester

Kyle

23.9 MountController

23.9.1 Overview

Controller for handling the Avatar when it is on a mount. Associates keys with movement.

23.9.2 Responsibilities

(a) Controls the mount.

23.9.3 Collaborators

(a) Mount

23.9.4 UML class diagram

- setKeyBindings()

23.9.5 Implementor

Kyle

23.9.6 Tester

Kyle

23.10 CameraController

23.10.1 Overview

The controller for moving the camera. The keys for camera movement are the same as regular movement plus a meta-key modifier.

23.10.2 Responsibilities

(a) Controls the camera.

23.10.3 Collaborators

(a) Camera

23.10.4 UML class diagram

- setKeyBindings()

23.10.5 Implementor

Kyle

23.10.6 Tester

Kyle

23.11 TradeMenuController

23.11.1 Overview

The controller for the state associated with trading with an NPC.

23.11.2 Responsibilities

(a) Handle the control of trade.

23.11.3 Collaborators

(a) Inventory
(b) Model

23.11.4 UML class diagram

- sellItem(int): void
- buyItem(int): void
- goBack(): void

23.11.5 Implementor

Kyle

23.11.6 Tester

Kyle

23.12 EntityController

23.12.1 Overview

Controls Entities.

23.12.2 Responsibilities

(a) Controls the Entity

23.12.3 Collaborators

(a) Entity

23.12.4 UML class diagram

- setKeyBindings()

23.12.5 Implementor

Kyle

23.12.6 Tester

Kyle

23.13 Listener

23.13.1 Overview

When the key given is pressed, calls the ability it contains.

23.13.2 Responsibilities

- Moving shit when the key it's registered to is pressed.

23.13.3 Collaborators

- JPanel
- Ability

23.13.4 UML class diagram

- Listener(KeyStroke, Ability) //Constructor
- addAsBinding(JPanel)

23.13.5 Implementor

Kyle

23.13.6 Tester

Kyle

23.14 DialogController

23.14.1 Overview

When the User enters text into the DialogView, it will be passed here to the Dialog, so that when on Enter, it will appear on the View.

23.14.2 Responsibilities

- Moving text to the ModelDialog

23.14.3 Collaborators

- JTextField
- Dialog (Model)

23.14.4 UML class diagram

- getText();

23.14.5 Implementor

Kyle

23.14.6 Tester

Kyle

24 Subsystem KeyPreferences

24.0.1 Overview

The key preferences system holds a users preferences that map keys to each conceptual action. These preferences will map a certain key to walk up, another key to use ability 2, and so on.

24.1 KeyPreferences

24.1.1 Overview

The key preferences system holds a users preferences that map keys to each conceptual action. These preferences will map a certain key to walk up, another key to use ability 2, and so on.

24.1.2 Responsibilities

(a) Map a key to a certain action to be taken on the current unit

(b) Provide an interface to change which keys map to which action.

24.1.3 Collaborators

(a) StructuredMap

24.1.4 UML class diagram

- getUpKey():KeyStroke
- getDownKey():KeyStroke
- . . .
- getMoveDownRightKey():KeyStroke
- getAbility1Key():KeyStroke
- . . . .
- getAbility9Key():KeyStroke
- getAttackKey():KeyStroke
- . . .
- getMenuMoveUpKey():KeyStroke
- getMenuMoveDownKey():KeyStroke
- getMenuMoveLeftKey():KeyStroke
- getMenuMoveRightKey():KeyStroke
- getMenuSelectKey():KeyStroke
- save() : StructuredMap
- load(StructuredMap):

24.1.5 Implementor

Kyle

24.1.6 Tester

Kyle

25 Subsystem RunGame

25.0.1 Subsystem Overview

This subystem will contain the entry point of the program and will be responsible for loading all the view, model, and controller and run a loop that periodically calls update on the model.

25.1 RunGame

25.1.1 Overview

This class will be responsible for creating a gameObject, which is a runnable, and throwing it into a thread.

25.1.2 Responsibilities

(a) Start the gameObject

25.1.3 Collaborators

(a) GameObject

25.1.4 UML class diagram

- gameObject: GameObject
- main(): void

25.1.5 Implementor

JR

25.1.6 Tester

JR

25.2 GameObject

25.2.1 Overview

This class will be responsible for containing the model object, the controller, and the view. It will periodically call update on the model to signal a "gameTick"

25.2.2 Responsibilities

(a) Start the gameObject

(b) Periodically call update on the Model

25.2.3 Collaborators

(a) GameObject

25.2.4 UML class diagram

{{ Implements Runnable }}

- model: Model
- view: View
- controller:
- run(): void

25.2.5 Implementor

JR

25.2.6 Tester

JR