Roguelike

Паттерны:

Фабрика:

- Предметов
- Мобов
- Можно аналогично для Map
- Можно аналоично для Skill

Спецификация:

- Для проверки уровня здоровья мобов и изменения поведения
- Для изменения характеристик персонажа в зависимости от надетых на него предметов и уровня
- Для повышения уровня после получения определенного количества опыта
- Для определения столкновения убойной части оружия с мобами и персонажем, персонажа с мобами, персонажа с предметами.

MapDrawer

Отрисовывает карту методом *DrawMap(map : Map)* на консоли.

ActivityDrawer

В двух дополнительных потоках отрисовывается активность мобов и персонажа методами *DrawMobsActivity(mobs : Map[*])* и *DrawCharacterActivity(character : Character),* соответственно.

Kernel

Сначала настраивается управление персонажем. Для этого настраивается экземпляр класса *ShortcutContoller* следующим образом.
CommandKernel.SetUpUI() -- строковому представлению нажатия клавиш на устройстве управления (для геймпадов же тоже коды какие-то сопоставляются для их обработки?) сопоставляется команда, обрабатывающая ввод.

Вызывается метод *Kernel.GoToNextLevel(),* котоорый организует переход на новый уровень, вызывая метод создания карты
Generator.CreateMap(kindOfDesign : Enum) с передачей типа для дизайна уровня, чтобы генератор понимал как ему создавать карту.

Потоки:

- Жизнь мобов
- Обработка ввода игрока с клавиатуры

После того, как *Generator* оживил всех мобов и запустил поток для жизнедеятельности мобов (например, движение, атака, смена стратегии поведения), в методе *Kernel.HandleUserInput()* создается поток для обработки пользовательского ввода с клавиатуры.
Парсится какая клавиша нажата, по словарю *ShortcutController.shortcuts* представлению клавиши сопоставляется функция обработки (function() {executeCommand(new Attack(executor, character}), и эта функция отрабатывает.

ShortcutController

Класс “хелпер” для обработки нажатой клавиши.

shortcuts : Dictionary<Enum, CharacterCommand> -- словарь сопоставление представлению клавиши функции обработки клавиши.
OnKeyPress(key : Enum, handler : CharacterCommand) -- заполняет словарь.
GetHandler(key : Enum) : CharacterCommand -- возвращает команду для обработки кнопки.

Map

Содержит свойства карты, которые настраивает *Generator.* Вообще, также можно сделать абстрактную фабрику для карт, чтобы создавать конкретные карты ада, роботизированные и т.п. Реализация UML представления этого дела аналогична абстрактным фабрикам мобов и предметов.

Generator

В методе *CreateMap(kindOfDesign : Enum)* в зависимости от типа дизайна уровня *Generator* создаёт карту, метод возвращает кортеж (Mob[*], Character), который будет использоваться классом *ActivityDrawer.* Тип *MobFactory* изменяется через *Generator.SetMobFactory(mobFactory : MobFactory)* (тип фабрики в *Generator* можно менять по мере генерации карты, чтобы наполнить её разнородными мобами).
Аналогично в *Generator* можно менять тип *Generator.CharacterItemFactory* с помощью *Generator. SetCharacterItemFactory(characterItemFactory : CharacterItemFactory).*
Методами *Generator.SpawnItemsOnMap(), Generator.SpawnCharacter()* и *Generator.SpawnMobsOnMap()* генератор размещает на карте предметы, персонажа и мобов, соответственно.
Метод *Generator.GenerateMap()* генерирует карту, настраивая экземпляр класса *Map* случайным образом согласно выбранному Вами алгоритму генерации карты.
Метод *Generator.LoadMapFromFile()* загружает готовую карту, парся файл и создавая экземпляр класса *Map.*

Далее вызывается отрисовка карты методом *MapDrawer.DrawMap(map : Map).*

Затем *Generator.EnlivenAllMobs()* запускает активность всех мобов (вообще, можно оживлять не всех, а в комнате. Но это уже не MVP (=) после создания всей карты.

После этого поток управления приложением передается в *Kernel*.

<<enumeration>> LevelDesign

Перечисление возможных дизайнов уровней. На этом основывается генерация карты.

<<interface>>ICollision

Контракт по которому элементы мира (Item, Mob, Character) определяют столкновение с собой.

Паттерн DDD Фабрика для Item

<<Interface>> CharacterItemFactory

Контракт для паттерна DDD “Фабрика” по созданию предметов: нагрудных, головных и т.п. Абстрагирует создания конкретных нагрудных, головных и других предметов от *Generator’a.*

<<abstract>>Item

Класс для игровых предметов.

Item.coordinates -- координаты объекта на карте.
DetectCollision() -- реализует контракт *<<interface>>ICollision.* Этот метод определяет, когда у *Item’a* произошло столкновение с элементами мира*.*
GetHealthBonus() -- возвращает число, на которое увеличивать количество здоровья персонажа.
GetAttackBonus() -- возвращает число, на которое увеличивать количество аттаки персонажа.

<<abstract>>ChestItem

Класс для нагрудных предметов. Наследуется от *<<abstract>>Item*

<<abstract>>HeadItem

Класс для головных предметов. Наследуется от *<<abstract>>Item*

RegularItemFactory

RegularItemFactory реализует контракт *<<Interface>> CharacterItemFactory*.

RegularItemFactory.CreateHeadItem() -- создание предметов типа *RegularHeadItem*
RegularItemFactory.CreateChestItem() -- создание предметов типа *RegularChestItem.*

RegularHeadItem

Конкретные головные предметы обычного (RegularHead) типа, создаваемые в *RegularItemFactory* и в *RareItemFactory.* Наследуется от *<<abstract>>HeadItem.*

RegularChestItem

Конкретные нагрудный предметы обычного (RegularChset) типа, создаваемые в *RegularItemFactory* и в *RareItemFactory*. Наследуется от <<*abstract*>>*ChestItem.*

RareItemFactory

RareItemFactory реализует контракт <<*Interface*>> *CharacterItemFactory*.

RareItemFactory.CreateHeadItem() -- создание предметов типа *RareHeadItem*
RareItemFactory.CreateChestItem() -- создание предметов типа *RareChestItem.*

RareHeadItem

Конкретные головные предметы обычного (RareHead) типа, создаваемые в *RareItemFactory* и в *RareItemFactory.* Наследуется от <<*abstract*>>*HeadItem.*

RareChestItem

Конкретные нагрудный предметы обычного (RareChset) типа, создаваемые в *RareItemFactory* и в *RareItemFactory*. Наследуется от <<*abstract*>>*ChestItem.*

Класс CharacterCommand

Использование паттерна *Команда.* Класс *CharacterCommand* является абстрактным. Он описывает общий для всех команд интерфейс. В нем содержится абстрактный метод *CharacterCommand.Execute*, который каждая конкретная команда определяет по-своему. В классе есть поля executer и character.

Класс Attack

Класс *Attack* расширяет класс *CharacterCommand.* Метод *Attack.Execute* переопределяется как команда атаки персонажа.

Класс Move

Класс *Move* расширяет класс *CharacterCommand.* Метод *Move.Execute* переопределяется как команда движения персонажа.

Класс OpenInventory

Класс *OpenInventory* расширяет класс *CharacterCommand.* Метод *OpenInventory.Execute* переопределяется как команда открытия инвентаря персонажа.

Класс EquipItem

Класс *EquipItem* расширяет класс *CharacterCommand.* Метод *EquipItem.Execute* переопределяется как команда персонажа надеть на себя предмет.

Класс Executer

Класс *Executer* содержит непосредственные операции в игре. Он играет роль получателя. Команды делегируют ему свои действия. Содержит поля map, character, mobs. Они отражают текущее состояние карты, текущего персонажа и мобов, которые в данный момент находятся на карте.

- Метод *Executer.ReduceHealthMob(count: int)* наносит урон мобу.
- Метод *Executer.PutOnItem(count: int)* вызывается при надевании на персонажа какого-либо предмета.
- Метод *Executer.MoveCharacter()* передвигает персонажа

Класс CommandKernel

Класс *CommandKernel* играет роль отправителя. В данном классе создаются объекты конкретных команд, в них передаются все необходимые параметры. У него есть метод *CommandKernel.SetUpUi(),* который привзязывает команды к элементам интерфейса. Метод *CommandKernel.ExecuteCommand()* запускает нужную команду.

Класс StateMob

Использование паттерна *Состояние.* Класс *StateMob* является абстрактным и описывает общий интерфейс для всех конкретных состояний. В конструктор класса передается интерфейс IMob, чтобы состояние могло обращаться к его данным и методам. Это сделано для динамического переключения между состояниями. Каждое из конкретных состояний реализует метод *StateMob.Behave()* по-своему.

Класс AggressiveState

Определяет агрессивное состояние моба. Если Моб находится в агрессивном состоянии, то он нападает. Метод *AggressiveState.Behave()* реализует атаку моба.

Класс PassiveState

Определяет пассивное состояние моба. Если Моб находится в пассивном состоянии, то он бездействует. Метод *PassiveState.Behave()* реализует бездействие моба.

Класс CowardlyState

Определяет трусливое состояние моба. Если Моб находится в трусливом состоянии, то он убегает. Метод *CowardlyState.Behave()* реализует побег моба.

Класс Mob

Абстрактный класс Mob реализует общий для всех мобов интерфейс. Метод *Mob.ReduceHealthAfterHitting()* уменьшает здоровье моба после удара персонажем. *Mob.RegenerateHealth()* увеличивает здоровье моба со временем. Метод *Mob.Attack()* отвечает за атаку моба. Метод *Mob.ChangeState()* позволяет изменять состояние моба (например, с трусливого на пассивный с увеличением здоровья). Метод *Mob.Behave()* делегирует работу активному состоянию. *Mob.EnlivenAfterCreation()* активирует моба после создания.

Интерфейс MobFactory

Интерфейс объявляет методы создания абстрактных мобов (разных семейств - адского и роботизированного).

Классы RoboticMobFactory и HellMobFactory

Это конкретные фабрики для роботизированных и адских мобов. Они относятся каждая к своей реализации моба и реализуют методы абстрактной фабрики.

Классы MeleeMob и ArcherMob

Это абстрактные классы, определяющие семейства мобов. Они описывают общий интерфейс для мобов ближнего и дальнего боя соответственно.

Классы RoboticArcherMob, RoboticMeleeMob, HellMeleeMob, HellArcherMob

Данные классы являются реализациями конкретных видов мобов.

Класс Character

Данный класс является реализацией главного персонажа. У него есть координаты, здоровье, экспа, сила и уровень. Метод *Character.IncreaseLevel()* позволяет увеличивать уровень персонажа. *Character.ReduceHealth()* уменьшает здоровье персонажа. Метод *IncreaseExperience()* увеличивает экспу персонажа.
DetectCollision() реализует контракт *<<interface>>ICollision*. Этот метод определяет, когда у *Character'a* произошло столкновение с элементами мира.