Roguelike

При загрузке игры в зависимости от настроек карта либо загружается из файла в классе *FileLoader* либо генерируется классом Generator: создаются мобы (по паттерну абстрактная фабрика), предметы (по паттерну абстрактная фабрика), персонаж и размещаются на карте. Карта отрисовывается. Начинают отрисовываться действия мобов и персонажа. Обработка нажатия клавиш игрока происходит с помощью паттерна “Команда” и благодаря конфигурированию класса *ShortcutController*. Работа с персонажем происходит в классе Character. Класс *UpgradeSystem* отвечает за улучшение характеристик персонажа с повышением уровня. На карте находятся предметы (абстрактный класс Item), которые могут давать новые скилы или улучшать характеристики персонажа. Противниками главного героя являются мобы (абстрактный класс Mob). В игре существует 4 вида мобов: Адские мобы ближнего и дальнего боя и роботизированные собы ближнего и дальнего боя. Мобы могут находиться в двух состояниях: паническом и состоянии по умолчанию. Оно задается при создании моба. В зависимости от состояния выбирается, на какую стратегию переключится моб.
Когда персонаж появляется на карте, он может выполнять команды:

- движение
- открыть инвентарь
- надеть на себя одежду
- атака оружием
- использование скилла
- сменить оружие

Реализация интерфейса ICollision классами Mob, Character, Weapon, Item позволяет определить столкновение объектов и задать поведение при этом столкновении (поднять предмет, нанести урон, препятствовать передвижению).

Задача игрока - перейти на новый уровень, для этого необходимо набрать определенное количество экспы. Чем выше уровень тем большее количество опыта надо набрать.
Если персонаж умирает, игра заканчивается, ни его местоположение, ни накопленные предметы не сохраняются.

Паттерны:

Фабрика:

- Предметов
- Мобов

Спецификация:

- Для проверки уровня здоровья мобов и изменения поведения
- Для повышения уровня после получения определенного количества опыта
- Для определения столкновения убойной части оружия с мобами и персонажем, персонажа с мобами, персонажа с предметами.

Операционный уровень

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(),* который 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.

После создания карты метод *Generator.CreateMap(kindOfDesign : Enum)* возвращает (*character : Character, map : Map*),

метод *MapDrawer.DrawMap(map : Map)* отрисовывает карту,
а метод *Kernel.EnlivenAllMobs(mobs : Mob[*])* запускает активность всех мобов (вообще, можно оживлять не всех, а в комнате. Но это уже не MVP (=) после создания всей карты.

Потоки:

- Жизнь мобов
- Обработка ввода игрока с клавиатуры

После того, как оживились все мобов и запустился 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.

Паттерн команда

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 привязывает команды к элементам интерфейса, конфигурируя класс ShortcutController. Метод *CommandKernel.ExecuteCommand(c: Command)* запускает нужную команду.

В конструктор передаётся: character : Character, shortcutController : ShortcutController. Также нужно проинициализировать fightingSystem : FightingSystem, characterController : CharacterController.

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 characterController, character, fightingSystem.

Класс AttackByWeapon

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

Класс UseSkill

Класс *UseSkill* расширяет класс *CharacterCommand.* Проверяет имеется ли нажатый скилл у персонажа и если имеется, то вызывает метод атаки скилом у боевой системы

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 ChangeWeapon

Класс *ChangeWeapon* расширяет класс *CharacterCommand.* Метод *EquipItem.Execute* переопределяется как команда персонажа поменять оружие для атаки.

CharatcerController

Класс отвечает за логику команд: надевание предметов, передвижение персонажа, открытия инвентаря. Он играет роль получателя. Команды делегируют ему свои действия.

FightingSystem

По паттерну “Стратегия” отвечает за изменение используемого оружия персонажа для атаки, атаку конкретным оружием, использование скиллов. Он играет роль получателя. Команды делегируют ему свои действия.

<<abstract>>Weapon

Абстрактный класс оружия персонажа, используемый в паттерне стратегия. Каждое оружие имеет свои характеристики. Конкретные классы оружия реализуют нанесение урона и определение коллизии объектов.

RangeWeapon

Класс оружие дальнего боя. Реализует Weapon.Damage().

MeleeWeapon

Класс оружие ближнего боя. Реализует Weapon.Damage().

<<interface>>ICollision

Контракт по которому элементы мира (Item, Mob, Character, Weapon) определяют столкновение с собой. Для предметов -- персонаж получает предмет, для противников и оружия -- нанесения урона, для ограничения передвижения из-за препятствий.

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.

<<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.*

RegularItemFactory

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

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

RareItemFactory

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

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

Уровень предметной области

<<abstract>>MapCreator

Абстрактный класс для создателей карт: Generator, MapLoader. Метод CreateMap переопределяют создатели. Предоставляет методы для установки типа конкретной фабрики мобов и предметов.

Map

Содержит свойства карты, которые настраивает *Generator.* Вообще, сверх ТЗ, также можно сделать абстрактную фабрику для карт, чтобы создавать
конкретные карты ада, роботизированные и т.п.

Generator

В методе *CreateMap(kindOfDesign : Enum)* в зависимости от типа дизайна уровня *Generator* создаёт карту, метод возвращает кортеж (Character, Map), 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.

После этого поток управления приложением передается в *Kernel*.

MapLoader

Метод *MapLoader.LoadMapFromFile()* загружает готовую карту, парся файл и создавая экземпляр класса *Map.*

<<enumeration>> LevelDesign

Перечисление возможных дизайнов уровней. На этом основывается какую карту нужно создавать.

<<abstract>>Item

Класс для игровых предметов. Содержит список атрибутов (например, +5 к здоровью, +10 к силе и -5 к количеству экспы, которое осталось набрать для перехода на следующий ровень) и скилов, которые дает данный предмет.

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.
DetectCollision() -- при столкновении с персонажем, предмет добавляется персонажу

<<abstract>>ChestItem

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

<<abstract>>HeadItem

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

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.*

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 StateMob

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

Класс DefaultState

Определяет действие моба по умолчанию.

Класс PanicState

В паникующее состояние моб переходит при достижении здоровья критического уровня.

Интерфейс Strategy

Стратегия отвечает за действие моба и реализует общий для всех стратегий метод *Strategy.Behave()*, содержимое которого зависит от конкретной стратегии.

Класс AggressiveStrategy

Определяет агрессивную стратегию моба. Если Моб находится в агрессивном состоянии, то он нападает. Метод *AggressiveStrategy.Behave()* реализует атаку моба.

Класс PassiveStrategy

Определяет пассивную стратегию моба. Если Моб пассивен, то он бездействует. Метод *PassiveStrategy.Behave()* реализует бездействие моба.

Класс CowardlyStrategy

Определяет трусливую стратегию моба. Если Моб находится в трусливом состоянии, то он убегает. Метод *CowardlyStrategy.Behave()* реализует побег моба.

Класс Mob

Абстрактный класс Mob реализует общий для всех мобов интерфейс. При его создании определяется стратегия (пассивная, агрессивная, трусливая) по умолчанию для данного моба. Метод *Mob.ReduceHealthAfterHitting() (вызывается через DetectCollision())* 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 состояние по умолчанию с увеличением здоровья). Метод *Mob.Behave()* делегирует работу активному состоянию. *Mob.EnlivenAfterCreation()* активирует моба после создания.

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() (вызывается в DetectCollision())* 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.

items добавляют характеристики и скилы.

IncreaseExperience() вызывается из класса моба при его смерти.

IncreaseCharacterPower(item: Item) -- изменяет характеристики персонажа и добавляет скилы в зависимости от надетого предмета

Die() -- завершает прохождение игры, персонаж умирает безвозвратно.

Класс UpgradeSystem

Данный класс отвечает за характеристики и скилы, которые присущи каждому уровню. attributeList и slillList это списки которые хранят соответствие номера уровня и атрибуты и скилы соответственно, которые имеет персонаж на каждом уровне. Метод GetAttribute(level) вызывается при повышении уровня или при инициализации игрока, он по номеру уровня определяет характеристики, которые имеет персонаж на данном уровне. Аналогично работает getSkills(level).

Класс Attribute

Содержит такие характеристики, как здоровье, силу и количество экспы которое нужно набрать для перехода на следующий уровень.

Interface ISkill

Определяет метод использования скила *ISkill.UseSkill()*, который каждый из конкретных скилов определяет по-своему.

Классы SuperAtackSkill, ClimbTreeSkill, InvisibleSkill

Реализуют интерфейс *ISkill* и переопределяют метод *ISkill.UseSkill()* как методы супер атаки, способности лазить по деревьям, способности становиться невидимым.

<<interface>> MobSkill

Определяет контракт для действия скилов мобов.

<<interface>> ArcherSkill

Контракт для мобов дальнего боя

<<interface>> MeleeSkill

Контракт для мобов ближнего боя

FireBall

Конкретный класс скилла моба. Производится дальнобойная атака огненным шаром

Exploser

Конкретный класс скилла моба. Производится взрыв моба с нанесением урона ближайшим противникам.