

# Ваши комментарии по домашке и наши ответы:

**Зеленые классы** -- измененные, добавленные классы для упрощения поиска на диаграмме.

- "- ChangeState(state: StateMob) вызывается в мобе в зависимости от количества здоровья" --- более в духе паттерна "Состояние" было бы доверить самим состояниям вызывать этот метод, потому что вообще говоря только они знают про условия перехода между состояниями. Иначе у вас "Стратегия" получилась

- Соответственно, рассуждения на второй половине первой страницы описывают StateMob не как конечный автомат, управляющий поведением моба, а как просто вынесенный в отдельный класс switch, который выдаёт мобу очередную стратегию. В общем, посмотрите ещё раз на паттерн "Состояние" и мои комментарии, вы что-то не так понимаете, либо у вас описано очень неаккуратно.

- Состояние определяет какую стратегию будет использовать моб. В паттерне "Состояние" состояние могут менять как контекст, так и сами состояния.
- При создании моба на фабрике мобу задается состояние по умолчанию: Фабрикой вызывается метод Mob.InitializeState(state : DefaultState), в который передаётся экземпляр DefaultState; затем экземпляр state конфигурируется методом MobState.SetContext(mob : Mob), принимающий созданного моба.
- Затем в метод DefaultState.ChooseStrategy в зависимости от типа моба, MeleeMob или ArcherMob, передаётся стратегия, AggressiveStrategy или PassiveStrategy соответственно.
  - (InitializeState)
    - this.state = state;
    - this.state.SetContext(this);
    - this.state.ChooseStrategy(strategy : Strategy);
- Методом Mob.EnlivenAfterCreation [напоминание: метод Kernel.EnlivenAllMobs(mobs : Mob[\*]) запускает активность всех мобов (вообще, можно оживать не всех, а в комнате. Но это уже не MVP (= ) после создания всей карты] вызывается Mob.LetsBehave, в котором вызывается метод MobState.LetsBehave(), в котором вызывается метод Strategy.Behave(mob : Mob), выполняющий реализацию конкретной стратегии моба:
  - LetsBehave()
    - this.state.LetsBehave()
  - Отдельный метод Mob.EnlivenAfterCreation нужен для того, чтобы в нём можно было бы реализовать, например, анимацию рождения, либо звуковое сопровождение при рождении моба (это к вопросу: зачем отдельный публичный EnlivenAfterCreation, если можно было бы сделать публичный LetsBehave согласно структуре паттерна "Состояние")
- При понижении здоровья до критического уровня у моба методом -Mob.ReduceHealthAfterHitting() моб вызывает метод +HandleCriticalHP(), в

котором вызывается метод `Mob.ChangeState(state: MobState)`, с аргументом `panicState : PanicState`, сконфигурированным методом `panicState.SetContext(this)`, где `this : Mob`, а также методом `panicState.ChooseStrategy(strategy : CowardlyStrategy)`.

- `ChangeState(state : MobState)`
  - `this.state = state;`
  - `Mob.LetsBehave()`
- Вопрос к Вашему комментарию “Соответственно, рассуждения на второй половине первой страницы описывают `StateMob` не как конечный автомат, управляющий поведением моба, а как просто вынесенный в отдельный класс `switch`, который выдаёт мобу очередную стратегию”:

информирование контекстом об уровне здоровья -- это тоже **switch**: если здоровье критическое, то состояние должно переключиться в паническое, если оптимальное -- в состояние по умолчанию. Этот свич (пусть даже неявно реализованный через методы `HandleCriticalHP` и `HandleOptimalHP`) -- это уже издержки игровой логики?

- В стратегии `CowardlyStrategy` регенерируется здоровье через метод `Mob.RegenerateHealth()`. При регенерации здоровья до оптимального уровня вызывается метод `MobState.HandleOptimalHP`, в котором вызывается метод `Mob.ChangeState(state: MobState)`, с аргументом `defaultState : DefaultState`, сконфигурированным методом `defaultState.SetContext(this)`, где `this : Mob`, а также методом `defaultState.ChooseStrategy(strategy : DefaultStrategy)`.

- "Боевая система на операционном, т.к. она отвечает за инициализацию атаки" --- всё, что в этом месте написано, описывает обычную функциональность уровня предметной области, которая не зависит от конкретного интерфейса или даже от конкретного приложения. Атака по идее должна одинаково обсчитываться и в случае просто десктопного приложения, и в распределённом случае, когда есть сервер и клиенты, и всё считается на сервере. `ICollision`, если он участвует в расчётах, тоже на уровне предметной области должен быть. Операционный уровень свой для каждого приложения, модель предметной области на всех одна.

- Получается, перемещение персонажа тоже должно быть на уровне предметной области => `CharacterController` перешел из операционного уровня.
- Отрисовка инвентаря -- на операционном, а вот сам инвентарь -- на уровне предметной области.

- Что произошло со стрелками в иерархии наследования `Item`-ов? `RareChestItem` наследуется от `HeadItem`, что? :) И там вообще какой-то хаос.

- Проблема в визуализации стрелочек :(
- Проведена коррекция.

- Что-то на уровне предметной области никак не моделируется инвентарь и то, что вещи можно надеть и снять. Ведь некоторые вещи могут быть подняты, но не надеты. Если вы предлагаете моделировать это и считать эффект от вещей на операционном уровне, буду вынужден с вами не согласиться.

- Частично логика предметов уже была реализована на уровне предметной области. А вот сущности инвентаря, действительно, не было, что нехорошо :) Добавлен инвентарь и его обработка.
- В команде OpenInventory вызываются:
  - InventoryDrawer.DrawInventory(inventory : Inventory) -- отрисовывает инвентарь персонажа
  - Inventory.OpenInventory() -- обрабатывает открытый инвентарь, чтобы можно было надеть и снять предметы.
- Inventory class:
  - Inventory.EquipItem(item : Item) -- обрабатывает надевание предмета. Вызывает метод +ChangeCharacterPower(isEquipping: true, item: Item). Меняет хар-ки преса. То же самое со скилами: изменяется их наличие при надевании предмета.
  - Inventory.RemoveItem(item : Item) -- обрабатывает снятие предмета. Вызывает метод +ChangeCharacterPower(isEquipping: false, item: Item). Возвращает измененные хар-ки обратно к состоянию (не про паттерн :) ) до надевания данного предмета. То же самое со скилами: изменяется их наличие при надевании предмета.
  - Inventory.OpenInventory() -- обрабатывает открытый инвентарь, чтобы можно было надеть и снять предметы.

В общем, пока 8.5 из 10, можно ещё поправить :)

- Сделано :)