Министерство Образования и Исследований Республики Молдова

Технический Университет Молдовы

Факультет Вычислительной Техники, Информатики и Микроэлектроники

Департамент Программной Инженерии и Автоматики

**КУРСОВАЯ РАБОТА**

по предмету

методы и механизмы проектирования программного обеспечения

на тему:

**Реализация системы экипирования в онлайн шутере**

**Implementation of the equipment system in an online shooter**

**Implementarea sistemului de echipamente într-un shooter online**

Студент: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Коваль Роман

подпись

Проверил: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ унив. ассистент Булдумак Олег

подпись

Кишинэy, 2022

СОДЕРЖАНИЕ

[ВВЕДЕНИЕ 3](#_Toc105377360)

[1 АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ 4](#_Toc105377361)

[1.1 Значимость темы 4](#_Toc105377362)

[1.2 Схожие системы и их специфика 5](#_Toc105377363)

[1.3 Цели создания системы 7](#_Toc105377364)

[1.4 Требования к системе и ее спецификация 8](#_Toc105377365)

[2 МОДЕЛИРОВАНИЕ СИСТЕМЫ 9](#_Toc105377366)

[2.1 Структура системы 9](#_Toc105377367)

[3 РАЗРАБОТКА СИСТЕМЫ 10](#_Toc105377368)

[3.1 Модуль генерирования сущностей 10](#_Toc105377369)

[3.2 Модуль состояний доступа 12](#_Toc105377370)

[3.3 Модуль генерирования ресурсов 14](#_Toc105377371)

[3.4 Модуль модификаций 15](#_Toc105377372)

[3.5 Модуль информирования 16](#_Toc105377373)

[3.6 Программный интерфейс системы 18](#_Toc105377374)

[ЗАКЛЮЧЕНИЕ 21](#_Toc105377375)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 22](#_Toc105377376)

[ПРИЛОЖЕНИЕ А: Скриншоты приложения 23](#_Toc105377377)

[ПРИЛОЖЕНИЕ Б: Ссылка на репозиторий проекта 24](#_Toc105377378)

# ВВЕДЕНИЕ

Шу́тер (от англ. shooter) — жанр компьютерных игр. На момент зарождения жанра за рубежом укрепилось слово «шутер», как вариант описания игрового процесса и перевод для слова shooter.

Обычно, в играх такого жанра игрок находится в трёхмерном пространстве и имеет некоторую свободу передвижения. Уровни, как правило, являют собой ограниченный лабиринт, в котором расположены враги, союзники и нейтрально настроенные объекты. Действия большинства шутеров разворачивается в анизотропном пространстве (помещения имеют очевидные пол и потолок, в них действует гравитация), хотя существуют и исключения, в которых пространство изотропно.

Для данной работы, в качестве основы взяты принципы многопользовательского шутера, для которого применимы все ранее перечисленные особенности, но он существенно отличается по своей сути и принципам игровых действий.

Массовый многопользовательский шутер от первого лица (от англ. Massively multiplayer online first-person shooter, или MMOFPS) — жанр компьютерных игр, сочетающий в себе особенности шутеров от первого лица и массовых многопользовательских онлайн-игр, в том числе и в формате браузерной игры. В этих играх большое количество пользователей взаимодействует друг с другом внутри виртуального мира. Можно сказать, что MMOFPS — это онлайн-игра в реальном времени, происходящая внутри крупной игровой области, с большим количеством одновременно участвующих игроков в виде шутера от первого лица. В этих играх представлены крупномасштабные, иногда командные сражения.

Ввиду быстротечного, стратегического характера игр в этом жанре, игроки должны рассчитывать в первую очередь на свои знания и физическую координацию, что придает большее значение индивидуальному мастерству игроков, нежели их обобщенным характеристикам, поскольку никакое количество игровых бонусов или подобного не способно восполнить отсутствие у игрока умения прицеливаться или тактического мышления.

Во многих MMOFPS функционируют экономические системы. Игровая валюта и предметы обычно зарабатываются в процессе игры и имеют для игроков определенную ценность. Подобные виртуальные экономики могут быть исследованы (если осуществляется журналирование игровых данных), что имеет значение в экономических исследованиях. Более значительным является то, что эти «виртуальные» экономики могут оказывать влияние на экономики реального мира.

# 1 АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ

Выбор темы должен опираться на разные аспекты, которые связаны с этой предметной областью. Такими аспектами выступают: значимость темы, описание и наличие плюсов этой реализации перед схожими системами, цели сотворения системы и требования к системе. Любой из этих качеств должен быть пристально рассмотрен перед созданием системы, так как процесс изучения перечисленного выше обязательно скажется на качестве выполнения данной работы.

Так, например, для разработки компьютерной игры в нынешних реалиях в команду могут одновременно входить: продюсер, издатель, геймдизайнер, художник, программист, звукорежиссер, тестировщик.

## Значимость темы

В начале 1980-х годов после появления первых домашних компьютеров и игровых приставок один программист мог управлять почти всеми задачами, связанными с разработкой игры. Разработка современных игр предполагает наличие широкого круга навыков и персонала поддержки. Для работы над одним проектом требуются целые команды, в состав которых обычно входят представители ряда специализаций.

Разработкой компьютерных игр может заниматься как один человек, так и фирма (коллектив разработчиков). Коммерческие игры создаются командами разработчиков, нанятыми одной фирмой. Фирмы могут специализироваться на производстве игр для персональных компьютеров, игровых приставок или планшетных компьютеров. Разработка может финансироваться другой, более крупной фирмой — издателем. Фирма-издатель по окончании разработки занимается распространением игры и берёт на себя связанные с этим затраты. Противоположным подходом является такая разработка, когда фирма самостоятельно (без участия издателей) распространяет копии игр, например, средствами цифровой дистрибуции.

Разработка наиболее крупнобюджетных игр может стоить десятки миллионов долларов США, причём в последние 2 десятилетия эти бюджеты непрерывно росли, как и численность команд разработчиков и сроки разработки. Так, в конце девяностых игру для консоли PlayStation для конечного покупателя — могла сделать команда из 10 человек за год, для PlayStation 2 (первая половина 2000-х годов) необходима была команда из 30-50 человек и два года разработки, к 2012 году речь шла уже о командах из свыше чем 100 разработчиков и срок порядка трёх лет. По утверждению Алекса Мура, геймдизайнера из компании Sumo Digital, если бы цена игры для конечного потребителя росла в той же пропорции, игры в 2012 году стоили бы по 1800 долларов США; иными словами, чтобы окупить возросшие бюджеты при сохранении тех же цен в магазинах, компании-издатели должны продавать намного больше копий игр.

Крупнобюджетная игра для двух платформ — Xbox 360 и PlayStation 3 — обходилась в 2012 году в среднем в 20 миллионов долларов, и для того, чтобы она окупилась, нужно было продать около двух миллионов копий.

Таким образом, разработка разнообразных механизмов, реализующих различную логику и поведение в играх может стать отличным опытом для любого разработчика и несомненно улучшить его умения моделирования, логистики и технические навыки. В будущем эти навыки, возможно, помогут ему стать частью большой команды разработчиков компьютерных, или мобильных игр, а может быть и даст толчок к чему-то самостоятельному, что заимеет огромный успех на просторах игрового пространства.

## 1.2 Схожие системы и их специфика

В настоящее время, практически в каждом онлайн шутере присутствует, своего рода, система экипирования. Эта система предоставляет пользователю выбор экипировки, необходимой его персонажу по мере игры. Как правило, в соревновательных шутерах, такая система представляет собой некий “магазин”, в котором игрок может получить необходимую экипировку заплатив за это внутриигровую валюту, полученную в виде регулярных начислений в течении матча, а также в качестве бонусов за хорошие игровые показатели или результаты его команды.

В качестве примера реализации данной системы можно привести популярные в нынешнее время онлайн шутеры.

В первом шутере, на рисунке 1.1, система экипирования представляет собой интерактивное меню, вызываемое пользователем по нажатию клавиши B на клавиатуре. Далее в меню пользователь может использовать для навигации как мышь, так и клавиатуру. Кроме того, в самом меню есть множество подсказок и дополнительных функций, обеспечивающих улучшение удобства пользования для игрока. Интерфейс такого меню интуитивно понятен и не вызывает трудностей в пользовании, однако концепция игра не предусматривает продажу оружия. Так, если пользователь необдуманно выбрал какую-либо экипировку и приобрел её, вернуть валюту за неё будет невозможно и раунд придется играть с ней.

Рисунок 1.1 – Интерактивное меню экипирования в Сounter Strike

Во втором случае, представлена система экипирования которая реализована в шутере под названием Valorant, интерфейс которой представлен на рисунке 1.2.

Главным отличием этой системы от предыдущей является способность возвращать экипировку в отведенное количество времени после её покупки, а также более узкий выбор экипирования. Однако здесь это нельзя назвать недостатком, так как концепция данной игры предполагает, что пользователь так же может приобрести не только экипировку, но и специальные заряды для необходимых в процессе игры умений, которые уникальны для каждого персонажа.

Меню здесь имеет совершенно другой вид, но дизайн и конструкция так же интуитивно понятны и не вызывают никаких трудностей в пользовании даже для новых игроков.

Возможно, её преимуществом можно назвать то, что здесь весь арсенал представлен пользователю сразу и нет необходимости преодолевать несколько вкладок меню, для того чтобы добраться до необходимого предмета экипировки. Однако и здесь не все так просто – вероятнее всего такой подход допускается ввиду не очень большого количества экипировки, в отличии от первого варианта, где предметов экипировки для персонажа куда больше.

Рисунок 1.2 – Интерактивное меню экипирования в Valorant

## 1.3 Цели создания системы

Главной целью данной работы является улучшение навыков в области разработки различных логических систем для разнообразных целей. В данном случае – подобие движка для системы экипирования в игре. Выполнив поставленные задачи, разработчик оттачивает существующие навыки и приобретает новые умения в области моделирования и построения логистических систем для взаимодействия человека с системой. Кроме того, задачи включают в себя создание чего-то нового, без сомнений, похожего на существующие и зарекомендовавшие себя системы, но с различными обновлениями и изменениями в угоду личных предпочтений разработчика, на основе его опыта взаимодействия с данными системами.

## 1.4 Требования к системе и ее спецификация

Система не должна целиком повторять существующие аналоги и представлять пользователю новые, расширенные функции. Системы должна быть понятна клиенту и в отношении пользователя не нести никакой нагрузки. То есть, пользователь должен как можно меньше взаимодействовать с логикой системы, а лишь использовать интуитивно понятные, настроенные операции.

Система должна быть открыта к расширению настолько, насколько это возможно. То есть, подразумевается использование множества модулей, расширение которых, ни в коем случае не нарушит работу и целостность существующих связей в данной системе.

Система должна реализовывать поставленные логистические задачи, которые должны соответствовать тем задачам, которые реализуют системы такого рода в контексте компьютерных игр.

Система должна придерживаться основных принципов программирования и принципов распределения ответственности и соблюдать четкий, или наиболее определенный баланс между функционалом, оформлением, читабельностью, доступностью и другими аспектами проектирования систем.

# 2 МОДЕЛИРОВАНИЕ СИСТЕМЫ

Для упрощения рабочего процесса и улучшения общего понимания концепции системы необходимо проработать модель системы и взаимодействие её компонентов, начиная с аппаратных средств и заканчивая программными модулями. Далее рассматривается некоторые базовые принципы построения данного проекта и очень обобщенные взаимодействия модулей в контексте программных средств, используемых в разработке.

## 2.1 Структура системы

Согласно спецификации, система предлагает пользователю набор инструментов, которые можно использовать для реализации механизма экипирования в шутере. Таким образом, система условно разделена на три уровня, как показано на рисунке 2.1.

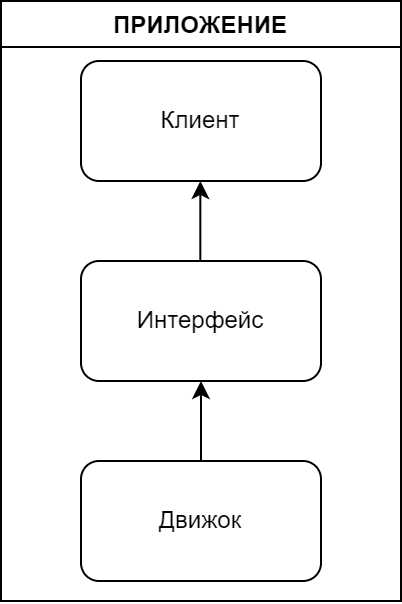


Рисунок 2.1 – Структура приложения

На схеме, которая представлена выше, движок реализует всю логику программы, создание сущностей и их взаимодействие. Интерфейс принимает в себя модули движка и содержит в себе всю логику, с помощью которой наш клиент сможет взаимодействовать с ними. Ну и последний элемент системы – сам клиент, который использует предоставленные ему инструменты.

# 3 РАЗРАБОТКА СИСТЕМЫ

На этом этапе рассматриваются шаги разработки и подробно описываются модули системы и их взаимодействие. В качестве среды разработки выбрал Visual Studio Code. Языком разработки будет TypeScript, позволяющий использовать строгую структурную типизацию в JavaScript. Для того, чтобы использовать JavaScript в качестве языка общего назначения, используется программная платформа NodeJS, которая добавляет возможность взаимодействовать через свой API с устройствами ввода-вывода, а также добавлять сторонние библиотеки, написанные на разных языках, и другое.

## 3.1 Модуль генерирования сущностей

Основная составляющая любого магазина, кроме персонала и оборудования – товар. В системе экипирования всё так же. Главные сущности здесь – элементы экипировки. Каждый отдельный элемент представляет собой класс, с соответствующими полями – характеристиками экипировки, как показано ниже:

class Pistol implements IGuns {

public ammo: number = 12;

public price: number = 250;

public scope: string = '0x';

public suppressor: boolean = false;

}

Кроме того, каждый класс реализует соответствующий интерфейс, а интерфейсы в дальнейшем наследуют от одного общего. Это сделано для удобства и детальнее отражает структуру этой части системы.

В программе присутствуют два типа экипировки – медицина и оружие. Группа элементов каждого отдельного семейства взята в рамки одного типа, для удобства дальнейшего использования данных сущностей. В частности, в инвентаре конкретного пользователя предметы сортируются тоже согласно обобщенным типам, с которыми можно ознакомится ниже:

type Guns = Pistol | Shotgun | SniperRifle | Rifle;

type Medicine = Bandage | FirstAidKit | MedKit;

Построение этого модуля основывается на фабричном методе проектирования, соответственно здесь есть и создатели, которые наследуют логику абстрактного создателя и создают продукт того, или иного семейства, в зависимости от нужны пользователя. Как реализуется конкретный создатель, можно посмотреть в листинге ниже:

abstract class Creator {

public abstract buy(product: string): any;

}

class GunCreator extends Creator {

buy(someGun: string): Guns {

if (someGun === 'pistol') {

return new Pistol();

} else if (someGun === 'rifle') {

return new Rifle();

} else if (someGun === 'sniper') {

return new SniperRifle();

} else if (someGun === 'shotgun') {

return new Shotgun();

}

throw new Error(`'${someGun}' doesn't exist in shop!`);

}

Как видно на листинге выше, конкретный создатель GunCreator наследует логику абстрактного класса Creator и реализует его метод таким образом, как это нужно ему. В данном случае, получая в качестве аргумента определенную строку, метод возвращает объект того или иного класса оружия, а в случае, если переданная строка не подходит не под один предусмотренный вариант, выкидывает ошибку. Стоит так же отметить, что возвращаемый тип в данном методе – Guns, который обобщает все наши классы, которые описывают тот или иной вид оружия.

Таким же образом это работает и для классов, который описывают медицинские предметы экипировки, а более детально все сущности, методы и их взаимодействия представлены на рисунке 3.1.

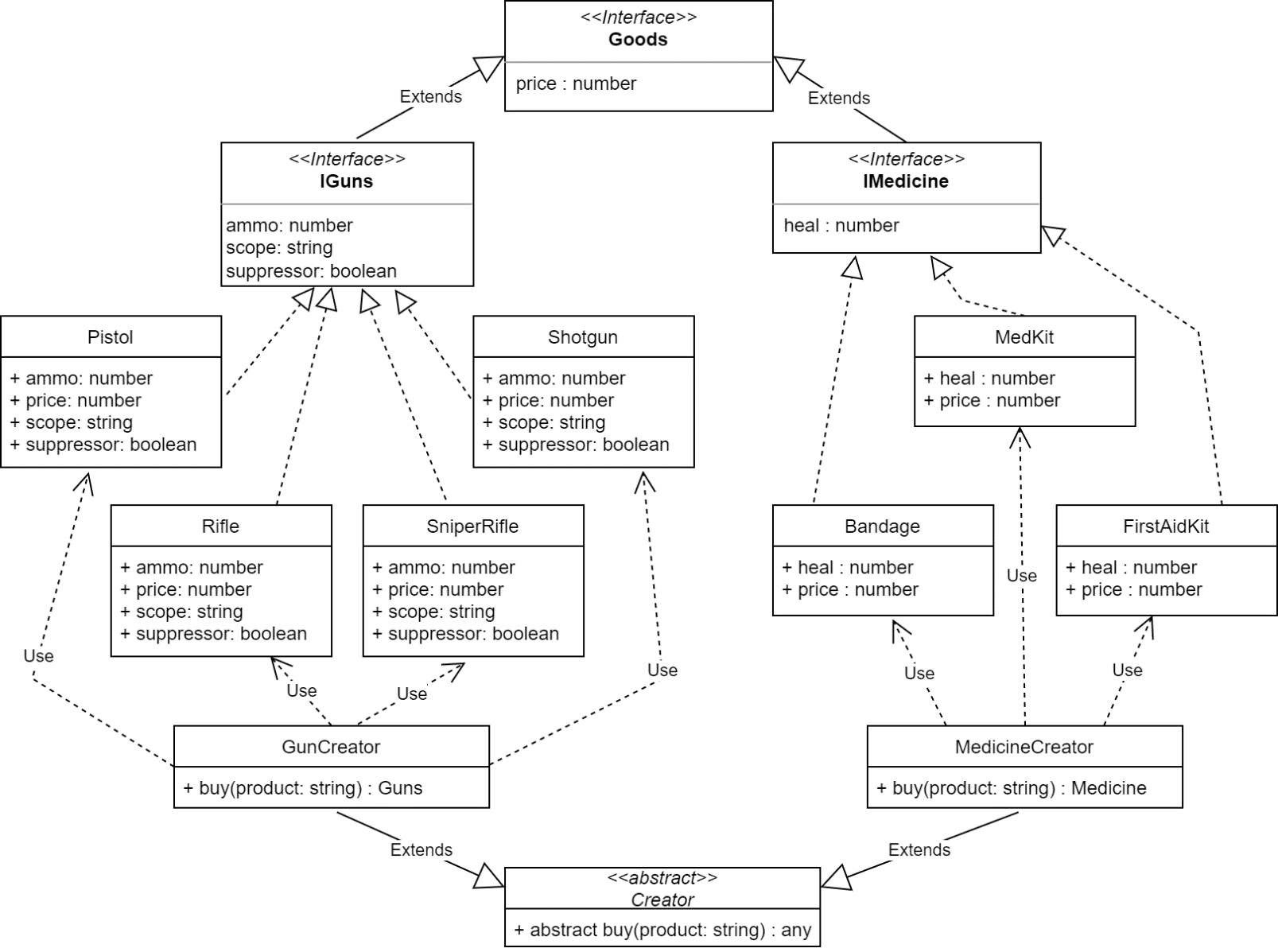


Рисунок 3.1 Диаграмма классов модуля генерирования сущностей

## 3.2 Модуль состояний доступа

Как правило, любому магазину присущ некоторый график, согласно которому этот магазин обслуживает клиентов. Соответственно, посетив магазин вне специально отведенного для этого времени, посетитель скорее всего не получит желаемого, ведь согласно графику, в данный момент, магазин должен быть закрыт и соответственно купить в нём ничего не получится. Данный модуль поможет реализовать подобную логику в системе и будет менять состояние доступа, а проще говоря, открывать и закрывать доступ магазину, в зависимости от того, какая стратегия передана в качестве параметра соответствующего метода.

Данный модуль представляет собой класс доступности магазина, который устанавливает ту или иную стратегию и возвращает соответствующий статус доступности, несколько классов-стратегий и интерфейс, который все эти классы-стратегии обязательно реализуют, чтобы потом без проблем передавать в класс доступности объект любого из них. Как выглядит класс доступности можно увидеть в листинге ниже:

class ShopAvailability {

private stage: Stages;

constructor(stage: Stages) {

this.stage = stage;

}

public setStrategy(stage: Stages): any {

this.stage = stage;

}

get status() {

return this.stage.setStatus();

}

}

В конструктор объектов данного класса передается объект класса-стратегии, который определяет, какой статус доступа будет у магазина. Один из таких классов и интерфейс, который он реализует представлены ниже:

interface Stages {

setStatus(): boolean;

}

class Init implements Stages {

public setStatus(): boolean {

console.log(`Initial stage...`);

return true;

}

}

Подробнее взаимодействие сущностей в данном модуле описывает диаграмма классов на рисунке 3.2.

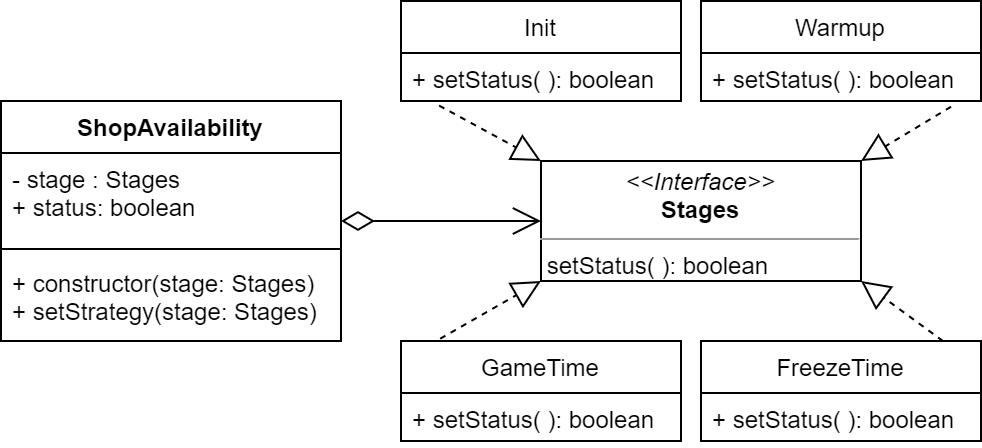


Рисунок 3.2 – Диаграмма классов для модуля состояний доступа

## 3.3 Модуль генерирования ресурсов

Для того, чтобы что-то купить необходимы деньги. В большинстве игр так же присутствуют внутриигровые ценности и покупаются они за внутриигровую валюту, однако, иногда приобрести тот или иной внутриигровой предмет можно и за реальные деньги, но в данном случае речь о другом. Для того, чтобы приобрести тот или иной предмет экипировки во время матча, игроку нужны ресурсы. В данном случае ресурс это и есть внутриигровая валюта, над которой в дальнейшем будут совершаться манипуляции, в зависимости от того, какие действия выполняет игрок. Данный модуль предоставляет глобальную точку доступа к единственному экземпляру класса, в котором и будет описана вся логика для финансовых операций в нашем движке. Ниже представлена часть реализации данного модуля:

class Balance {

private static \_instance: Balance = new Balance();

private \_money: number = 10000;

private constructor() {

if (Balance.\_instance) {

throw new Error(

'Error: Instantiation failed: Use Balance.getInstance() instead of new.'

);

}

Balance.\_instance = this;

}

…

Остальная логика представлена для ознакомления в диаграмме классов на рисунке 3.3.

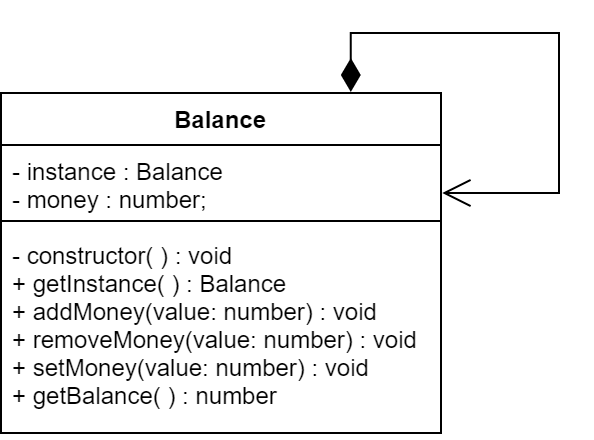


Рисунок 3.3 – Диаграмма классов для модуля генерирования ресурсов

## 3.4 Модуль модификаций

Аналог данной операции из реальной жизни представить сложно, поскольку если модифицировать товар прямо в магазине, предварительно не оплатив его, то скорее всего последствия будут неблагоприятными для того, кто так поступил. В данном случае, модуль нужен для того, чтобы изменять нужный игроку предмет экипировки в различных вариациях. Достигается это следующим образом – в конструктор объектов базового класса декоратора поступает объект какого-либо класса, который охвачен типом Guns. После чего, конкретные декораторы, которые наследуют от базового изменяют объект таким образом, как этого захотел игрок. Пример реализации конкретного декоратора представлен ниже:

class AddSuppressor extends Decorator implements Upgrade {

public upgrade(): Guns {

this.object.suppressor = true;

return this.object;

}

Как видно на листинге выше, конкретный декоратор не только наследует логику базового класса, но и реализует интерфейс Update – это нужно для того, чтобы в будущем, в программном интерфейсе мы могли полиморфно использовать методы модификации, в зависимости от их входных данных. Базовый класс-декоратор, интерфейс и остальные конкретные декораторы представлены на рисунке 3.4.

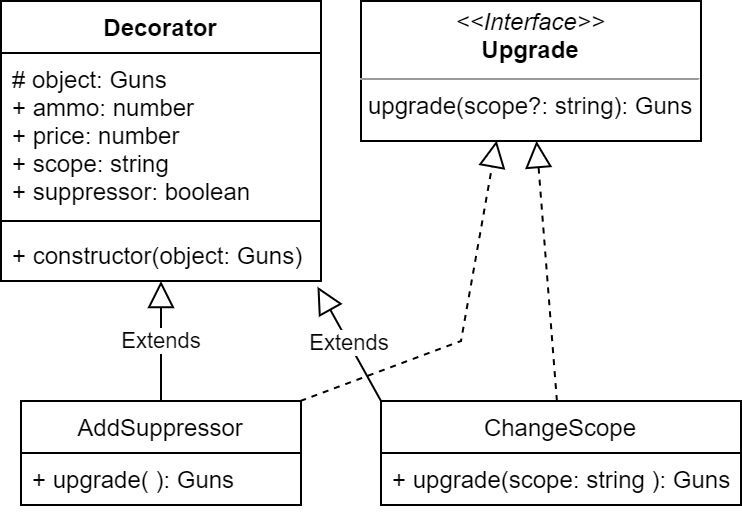


Рисунок 3.4 – Диаграмма классов для модуля модификации

## 3.5 Модуль информирования

В любом уважающем себя магазине есть информационный щит для того, чтоб донести какую-то информацию посетителям, которые пользуются услугами этого заведения. В данном случае, модуль будет передавать пользователям авторскую и техническую информацию используя шаблонный метод проектирования.

Таким образом модуль содержит абстрактный класс Information, который реализует общий для всех наследников метод showDate, абстрактный метод showAdditional, который будет реализован в наследниках по-разному и метод showDefault с пустой реализацией, который можно будет переопределить в наследниках при необходимости. Подробнее с реализацией можно ознакомиться в листинге ниже:

abstract class Information {

protected showDate(): void {

console.log(`\n\n-> Session requested on ${new Date().toLocaleString()}`);

}

protected abstract showAdditional(): string;

protected showDefault(): void {}

public showInfo(): void {

this.showDate();

console.log(this.showAdditional());

console.log(this.showDefault());

}

}

Как видно на листинге выше, с помощью метода showInfo мы в итоге выводим всю необходимую информацию, в зависимости от того, для экземпляра какого класса-наследника мы его вызываем. Один из таких классов так же представлен ниже:

class TechnicalInformation extends Information {

protected showAdditional(): string {

console.log(`\n---- Technical Information ----`);

return 'This project is written in TypeScript. Released on the NodeJS platform.';

}

protected showDefault(): string {

return `By default team balance is ${Balance.getInstance().getBalance()}\nNo users by default!\nBy default app is in init stage.That means u need to run script to start.\n On warmup u can't buy, on freeze time u have 10 seconds to buy, next u have to play, so shop gonna be unavailable.\n\n`;

}

}

Диаграмма классов для данного модуля представлена на рисунке 3.5.

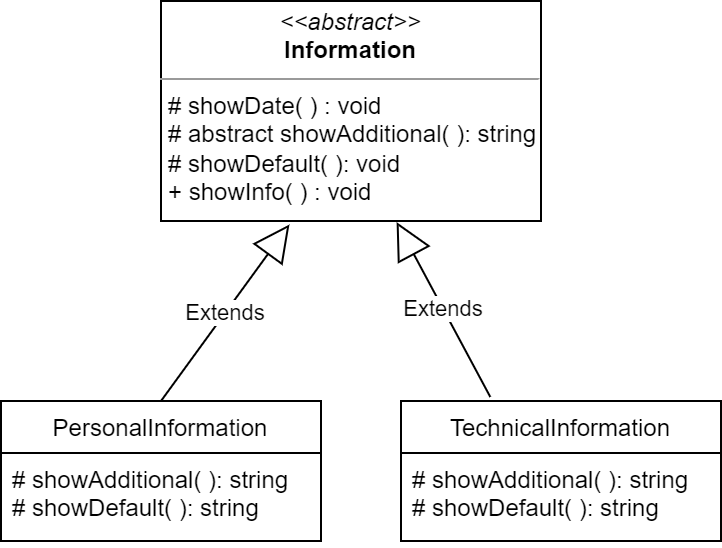


Рисунок 3.5 – Диаграмма классов для модуля информирования

## 3.6 Программный интерфейс системы

Программный интерфейс системы представляет собой фасад, который станет посредником между пользователем и движком. На листинге ниже можно увидеть, какие подсистемы будут участниками этого интерфейса:

class ShopAPI {

private balance = Balance.getInstance();

private canBuy: boolean;

private gameStage: ShopAvailability;

private improver: Upgrade;

private allUsers: Users[];

private creator: Creator;

constructor() {

this.gameStage = new ShopAvailability(new Init());

this.improver = new AddSuppressor(new Pistol());

this.canBuy = this.gameStage.status;

this.allUsers = [];

this.creator = new GunCreator();

}

…

Можно заметить, что большинство сущностей нам уже знакомо из описания предыдущих модулей, неудивительно, ведь фасад мы используем для того, чтоб заставить их взаимодействовать друг с другом напрямую, посредством фасада, при этом не зная о нём. Примером такого взаимодействия может быть один из методов класса ShopAPI, который нужен для модификации предметов экипировки:

public upgradeGun(gun: Guns, type: string, scope?: string) {

if (type === 'supp') {

this.improver = new AddSuppressor(gun);

this.improver.upgrade();

} else if (type === 'scope') {

this.improver = new ChangeScope(gun);

this.improver.upgrade(scope);

} else if (type === 'both') {

this.improver = new AddSuppressor(gun);

this.improver.upgrade(scope);

this.improver = new ChangeScope(gun);

this.improver.upgrade(scope);

}

}

Кроме приведенного выше метода, программный интерфейс содержит в себе еще много различных операций и методов, которые описывают различный функционал. С некоторыми из них фасад взаимодействует напрямую, а с некоторыми лишь косвенно. Так, например, для доступа к инвентарю пользователя, фасад использует публичное поле объекта класса Users, которое в себе хранит объект класса Backpack, поля которого и представляют собой отдельные части инвентаря. Такой подход лучше всего отражают следующие строки кода:

if (this.creator.buy(whatToBuy).price <= this.balance.getBalance()) {

user.backpack.gunsInside.push(this.creator.buy(whatToBuy));

this.balance.removeMoney(this.creator.buy(whatToBuy).price);

О том, с какими модулями и как взаимодействует фасад можно узнать подробнее, ознакомившись с диаграммой на рисунке 3.6.

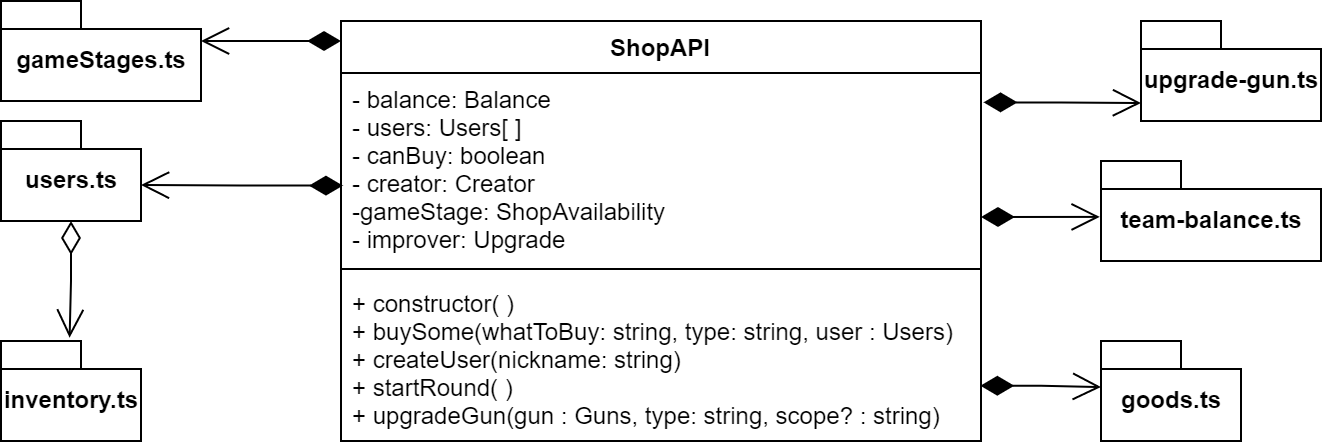


Рисунок 3.6 – Диаграмма классов для программного интерфейса

# ЗАКЛЮЧЕНИЕ

Информационные технологии не стоят на месте и каждый день специалисты в этой области обнаруживают для себя новые знания или по-новому смотрят на то, что уже знают. Область разработки игр так же не стоит на месте, и разработчики постоянно предлагают пользователям своих продуктов всё более совершенные и требовательные продукты. В разработке действительно серьезного продукта, как правило, участвует множество различных отделов той или иной ветки в компании и каждый из них занимается конкретной задачей – от написания движка, до отрисовки уровней и тестирования продукта на уровне разработки.

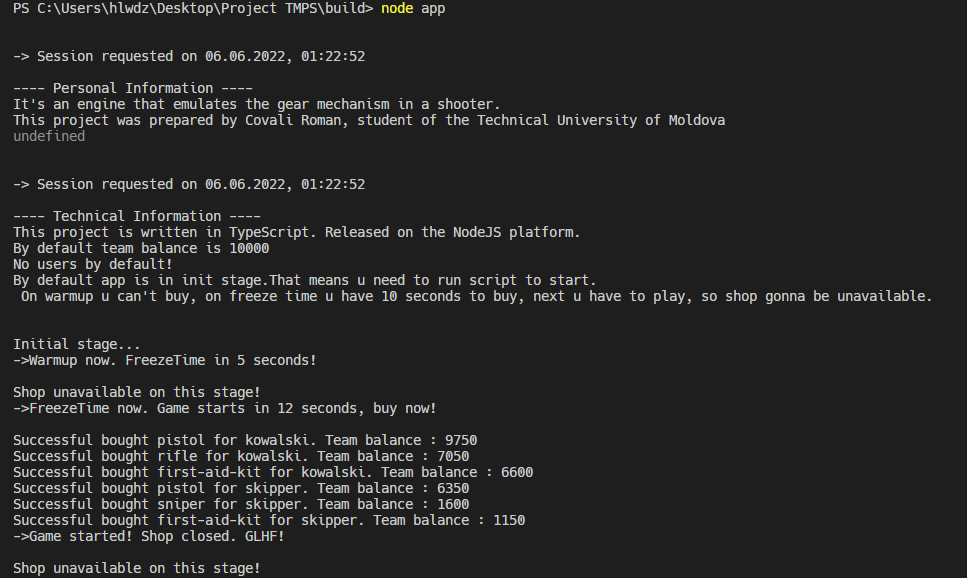
Реализация систем, подобных той, что реализована в этой работе дело не из простых. Конечно, нельзя назвать её конкурентноспособной по сравнению с готовыми решениями ведущих компаний, специализирующихся на этом, однако программа достаточно точно отражает логику и механизм применения таких систем в готовом продукте.

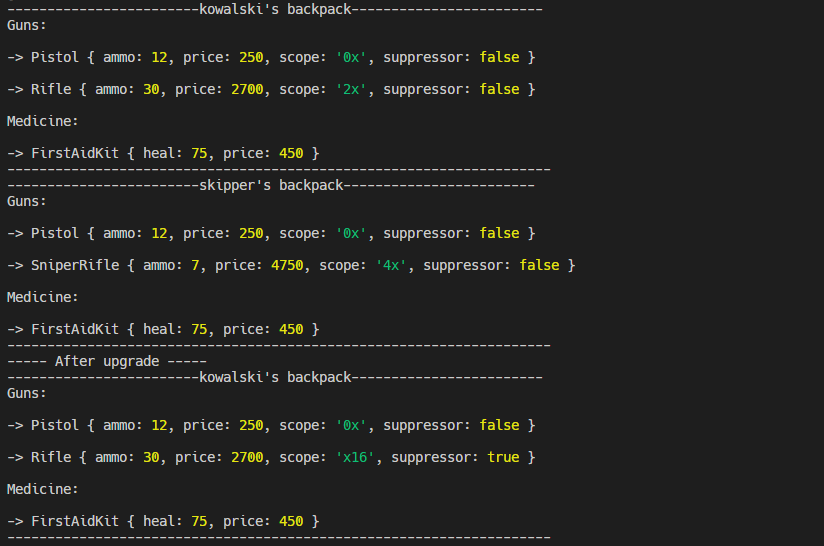
В ходе разработки данного движка по необходимости применялись соответствующие паттерны проектирования в объектно-ориентированном программировании. Где-то, ввиду необходимости, их концепция подвергалась модификациям, а где-то и вовсе было принято решение отказаться от использования того или иного паттерна в виду ухудшения читабельности и нелогичного увеличения объема кода. С другой стороны, на некоторых этапах разработки паттерны идеально подходили под необходимые задачи в том или ином модуле и отлично выполняли свою функцию. В некоторых местах структура кода была построена таким образом, что он сам по себе являлся паттерном, хотя изначально таковой задумки не было. Подытожив слова выше, можно сделать вывод, что любой уважающий себя разработчик должен обладать знаниями в области паттернов проектирования в объектно-ориентированном программировании и по возможности следовать им. Однако, не стоит проявлять фанатичность и использовать их везде, а особенно там, где они точно навредят вашему коду.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. ERIC, ELIZABETH FREEMAN (2011). “Design Patterns” ISBN 978-5-459-00435-9
2. DUSTIN BOSWELL, TREVOR FOUCHER (2012). “The Art of Readable Code” ISBN 978-5-459-01188-3.
3. ERICH GAMMA, RICHARD HELM, JOHN VLISSIDES, RALPH JOHNSON (2021). “Design Patterns” ISBN 978-5-4461-1595-2
4. ALEXANDER SHVETS (2021). “Dive Into Design Patterns”
5. ROBERT NYSTROM (2014). “Game Programming Patterns” ISBN 978-0-9905829-0-8
6. ROBERT C. MARTIN (2018). “Clean Architecture. A Craftsman’s Guide to Software Structure and Design” ISBN 978-0-13-449416-6
7. SCOTT ROGERS (2014). “Level Up! The Guide to Great Video Game Design, 2nd Edition” ISBN 978-1-118-87716-6
8. ADAM FREEMAN (2021). “Essential TypeScript 4: From Beginner to Pro” ISBN 978-1-4842-7010-3
9. MARIO CASCIARO, LUCIANO MAMMINO (2020). “Node.js Design Patterns: Design and implement production-grade Node.js applications using proven patterns and techniques, 3rd Edition” ISBN 978-1-83921-411-0
10. DAVID FLANAGAN (2020). “JavaScript: The Definitive Guide: Master the World's Most-Used Programming Language, 7th Edition” ISBN 978-1-491-95202-3

# ПРИЛОЖЕНИЕ А: Скриншоты приложения





# ПРИЛОЖЕНИЕ Б: Ссылка на репозиторий проекта

https://github.com/hollywoodzzz/Project-TMPS