Министерство образования и науки Российской Федерации
Федеральное государственное автономное
образовательное учреждение высшего образования
«Южно-Уральский государственный университет
(национальный исследовательский университет)»
Высшая школа электроники и компьютерных наук
Кафедра «Электронные вычислительные машины»

ДОПУСТИТЬ К ЗАЩИТЕ
Заведующий кафедрой
_____Г.И. Радченко
«___»_____2018 г.

Проектирование движка для класса игр «Симулятор профессии»

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
К ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЕ
ЮУРГУ-090301.2018.143 ПЗ ВКР

Руководитель работы,
доцент каф. «Электронные
Вычислительные машины»
_____И.Л. Надточий
«___»_____2018 г.

Автор работы
студента группы КЭ-484
_____А.Д. Рогозин
«___»_____2018 г.

Нормоконтролёр, ст. преп. каф.
«Электронные вычислительные
машины»
_____В.В. Лурье
«___»_____2018 г.

Челябинск 2018

АННОТАЦИЯ

Рогозин А.Д. Проектирование движка и инструментария для класса игр «Симулятор профессии». – Челябинск: ЮУрГУ, КЭ-484, 128 с., 28 илл., библиогр. список – 71 наим.

В рамках выпускной квалификационной работы автор знакомился с отраслью разработки игр, благодаря проектированию игрового движка и инструментария.

Целью выпускной квалификационной работы является проектирование движка для класса игр «Симулятор профессии». Это позволит упростить разработку такого класса игр. Данный продукт был создан на основе языка C#, среды разработки Visual Studio, системы управления базами данных SQLite и Entity Framework.

В последнее время появился спрос на мобильные игры, чаще всего встречающиеся под такими названиями «Симулятор студента», «Симулятор хакера», «Симулятор чиновника» и т.п. Суть этих игр в том, что игроку даётся персонаж, которого он должен провести через череду действий к победе.

Персонаж добивается поставленной цели ежедневными действиями на разных локациях, с помощью которых он развивает свои характеристики. Он может гулять по локациям, покупать товары в магазине, отвечать действиями на какие-либо события, получать из-за этого какие-то последствия, влияющие на его состояние.

В рамках выпускной квалификационной работы был создан гибкий движок для класса игр «Симулятор профессии», который можно подключить к другим средам разработки для упрощения разработки такого класса игр. Данная система позволяет программисту не задумываться о логике движка, предоставляя ему удобные для понимания функции, обеспечивающие инкапсуляцию.

Было проведено тестирование работы системы на платформе .Net.

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

В современном мире создание мобильных видеоигр является одним из наиболее крупных сегментов индустрии развлечений, рынок которого растёт с каждым годом.

Следует отметить, что за последние 3 года изменилась структура мирового рынка игр по сегментам. Если в 2013 году большая доля рынка приходилась на игровой ПК-рынок (почти 40%), а 37% занимал сегмент консолей, то в 2016 году самым крупным сегментом становится сегмент мобильных игр, генерируя выручку 36,9 млрд долл., что соответствует 37% всего объема мирового рынка игр. То есть в 2016 году доходы в мобильном сегменте впервые превысили доходы от сегмента игр на персональных компьютерах.

Аналитики Newzoo утверждают, что рынок игр (компьютерных, мобильных и прочих) по итогам текущего года достигнет отметки в 137,8 млрд долларов, что на 13,3% превысит показатель 2017 года.

Самым крупным сегментом остаются мобильные игры. По итогам года ожидается, что этот рынок достигнет отметки в 70,3 млрд долларов, то есть на мобильные игры будет приходиться более половины всего рынка. В годовом выражении этот сегмент вырастет на 25,5%.

Оставшиеся 49% практически пополам разделят консольные и компьютерные игры. Если точнее, первый сегмент займёт 25% рынка (34,6 млрд долларов; рост на 4,1%), а второй – 24% (32,9 млрд долларов; рост на 1,6%).

Что касается более долгосрочных прогнозов: аналитики считают, что к 2021 году игровой рынок вырастет до 180,1 млрд долларов. 59% будут занимать мобильные игры, 19% – компьютерные игры и 22% – видеоигры.

В последнее время появился спрос на мобильные игры, чаще всего встречающиеся под такими названиями «Симулятор студента», «Симулятор хакера», «Симулятор чиновника» и т.п. Суть этих игр в том, что игроку даётся персонаж, которого он должен провести через череду действий к победе.

Персонаж добивается поставленной цели ежедневными действиями на разных локациях, с помощью которых он развивает свои характеристики. Он может гулять по локациям, покупать товары в магазине, отвечать действиями на какие-либо события, получать из-за этого какие-то последствия, влияющие на его состояние.

Как правило, количество действий в день у персонажа ограничено. За день персонаж может потратить все свои действия, после чего он не сможет ничего сделать. Каждое утро количество действий восстанавливается.

В процессе разработки игры у разработчика остро встаёт вопрос: как сильно будет зависеть игра от движка. Есть три варианта развития событий:

1) разработчик напишет с нуля или допишет уже существующее решение, позволяющее создать игру в удобной среде разработки;

2) разработчик так сильно свяжет движок с игрой, что не позволит отдельно использовать его для создания новой игры;

3) разработчик воспользуется уже готовым движком для своей цели, слегка видоизменив его под свои нужды.

К сожалению, среда разработки, которая была бы заточена под разработку такого рода игр, не была найдена в сети Интернет, хотя есть спрос на такого рода игры среди покупателей. Из этого следует, что следует заняться реализацией такого движка, с помощью которого можно было бы упростить работу создателям такого класса игр.

Целью выпускной квалификационной работы является проектирование движка для класса игр «Симулятор профессии». Данный продукт будет создан на основе языка C#, среды разработки Visual Studio, СУБД SQLite и Entity Framework.

1 Введение в терминологию

Игровой движок (англ. game engine) – программное обеспечение, которое пригодно для повторного использования и расширения, и тем самым может быть рассмотрено как основание для разработки множества различных игр без существенных изменений.

Библиотека – сборник подпрограмм или объектов, используемых для разработки программного обеспечения.

Плагин – независимо программный модуль, который динамически подключается к основной программе и предназначается для расширения её возможностей.

Среда разработки – комплекс программных средств, который используется программистами для разработки программного обеспечения и предоставляющий удобные инструменты для разработки.

Модуль – см. плагин.

Система управления базами данных (СУБД) – комплекс программ, которые обеспечивают создание и манипуляцию над данными в базе данных.

БД – совокупность сущностей, связей и атрибутов предметной области, сгруппированные особым образом, чтобы можно было провести эффективный поиск по ним с помощью вычислительного устройства.

Инкапсуляция – упаковка данных и функций в единый компонент, предоставляющий пользователя внешний интерфейс и скрывающий внутренний интерфейс.

ORM (объектно-реляционное преобразование) – технология программирования, связывающая реляционные таблицы базы данных с концепциями объектно-ориентированных языков программирования.

Кроссплатформенность – способность программного обеспечения работать более чем на одной аппаратной платформе и (или) операционной системе.

Инструментарий – см. API.

Язык программирования – формальная знаковая система, которая предназначена для записи компьютерных программ.

API (программный интерфейс приложения, интерфейс прикладного программирования) – набор готовых классов, функций, который предоставляется библиотекой, сервисом или операционной системой для использования во внешних программных продуктах.

Абстракция — это модель некоего объекта или явления реального мира, откидывающая незначительные детали, не играющие существенной роли в данном контексте.

Паттерн проектирования — это часто встречаемое решение определённой проблемы при проектировании архитектуры программ.

2 Предметная область, постановка задачи и анализ существующих решений

2.1 Игровая индустрия в целом

Индустрия компьютерных игр или индустрия интерактивных развлечений – сектор экономики, связанный с разработкой, продвижением и продажей компьютерных игр.

Индустрия компьютерных игр начала свой путь в середине 20 века как движение энтузиастов и за несколько десятилетий выросла из небольшого рынка в популярную индустрию с годовой прибылью в 9,5 миллиардов долларов в США в 2007 году и 11,7 миллиардов в 2008 году (согласно ежегодным отчётам ESA). На рынке работают как крупные игроки, так и небольшие фирмы, а также независимые разработчики и сообщества. [1] В современном мире создание видеоигр является одним из наиболее крупных сегментов индустрии развлечений.

По степени влияния на потребителей и вовлеченности их в интерактивное окружение, предлагаемое видеоиграми, этот сегмент уже давно выделяется среди других видов развлечений.

Разработку игр невозможно рассматривать обособленно от индустрии компьютерных игр в целом. Непосредственно создание игр – это только часть комплексной системы, обеспечивающей полный жизненный цикл производства, распространения и потребления таких сложных продуктов, как компьютерные игры.

В структуре современной разработки видеоигр можно выделить следующие уровни: платформы, игровые движки, процесс разработки видеоигр. [2]

2.2 Игровые платформы

Игровая платформа – аппаратно-программные системы, позволяющие запускать интерактивные игровые приложения.

Игровые платформы можно разделить на несколько типов:

- персональный компьютер;
- игровая приставка;
- смартфон;
- аркадный автомат.

Персональный компьютер – техническое средство индивидуального пользования, ориентированная на решение различных задач. Одной из таких задач может быть воспроизведение видеоигр. [3]

Игровая приставка (игровая консоль) – специализированное электронное устройство, предназначенное для видеоигр. [4] Для таких устройств, в отличие от персональных компьютеров, запуск и воспроизведение видеоигр является основной задачей. Рынок игровых приставок развился из сравнительно простых электронных телевизионных игровых систем, таких как Pong, превратившись в наши дни в мощные многофункциональные игровые системы.

Смартфон – мобильный телефон, дополненный функциональностью карманного персонального компьютера. [5] Хотя в мобильных телефонах практически всегда были дополнительные функции (калькулятор, календарь), со временем выпускались все более и более интеллектуальные модели, для подчеркивания возросшей функциональности и вычислительной мощности таких моделей ввели термин «смартфон». Установка дополнительных приложений позволяет значительно улучшить функциональность смартфонов по сравнению с обычными мобильными телефонами. К этой же категории можно отнести и планшеты, построенные на тех же технологиях, что и в смартфонах. В настоящее время у смартфонов есть большой рынок мобильных игр.

Аркадный автомат – это стойка с аппаратурой для запуска игр, обычно аркадного жанра. [6] Многие известные компьютерные игры впервые появились на игровых автоматах и лишь через некоторое время были перенесены на приставки и домашние компьютеры. Игровой процесс данного жанра достаточно

прост, а каждый автомат делался под конкретную игру, соответствуя ей даже внешним видом.

2.3 Игровой движок

Игровой движок – программная прослойка между платформой и кодом игры. Использование готового игрового движка позволяет существенно упростить разработку новых игр, удешевить их производство и существенно сократить время до их релиза. Также современные игровые движки могут обеспечивать кроссплатформенность создаваемых продуктов. Они предоставляют разработчикам инструменты для создания большинства компонентов игры, которые в дальнейшем можно будет объединить в единое целое, представляющее игру.

Движок игры затрагивает все компоненты игры, начиная от отображения графического интерфейса, физики, звукового оформления, написания кода, создания ИИ и заканчивая сетевыми аспектами. А если что-то создать невозможно, следует создать это в специализированной программе и потом импортировать в игру. [7]

2.4 Разработка игр

Большое количество компаний и независимых команд занимаются созданием компьютерных игр. В разработке участвуют специалисты разных профессий: программисты, игровые дизайнеры, художники и др. [8]

К разработке крупных коммерческих игровых продуктов привлекаются большие профессиональные команды численностью более 100 специалистов. И стоить подобные проекты будут намного дороже, чем разработка проекта небольшой студией.

Однако вполне успешные игровые проекты могут воплощаться и небольшими командами энтузиастов. Этому способствует присутствие на рынке большого количества открытых и распространенных платформ, качественные

движки, площадки по привлечению дополнительных ресурсов для разработки и доступные каналы распространения.

2.5 Игровая индустрия в России

Россия в виду своей специфики долгое время отставала от развитых стран по распространению цифровых технологий. Также серьезную проблему представляло пиратство. Индустрия разработки видеоигр начала формироваться только в конце 1990-х годов. И лишь распространение MMO игр в середине 2000-х дало толчок к расширению игрового рынка в России и серьезному росту доходов игровой индустрии. Инвестиции и совместные с крупными западными компаниями игровые проекты стали появляться и в России. [2]

2.6 Будущее игровой индустрии

В предыдущие пять лет игровая индустрия во всем мире переживает бурный рост (см. рисунки 2.6.1 и 2.6.2). Рост наблюдался во всех сегментах, но основными движущими силами в этот период стали мобильные игры на двух основных платформах: iOS и Android.

Согласно прогнозам J’son and Partners Consulting (см. рис. 2.6.3), объём мирового рынка игр к 2021 году составит почти $130 млрд, демонстрируя средние ежегодные темпы роста в период 2016 - 2021 гг. на уровне 5,4%. [9]

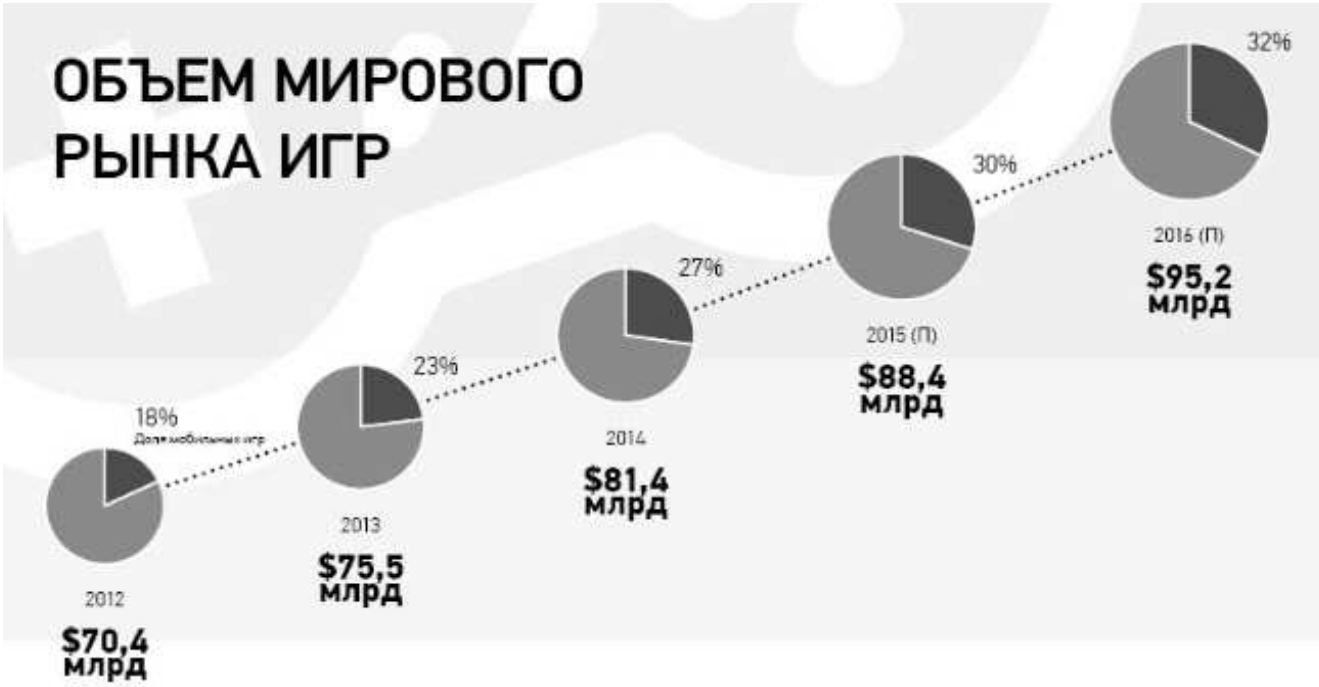

Рисунок 2.6.1 – Динамика мирового рынка видеоигр в 2012-2016 годах

Следует отметить, что за последние 3 года изменилась структура мирового рынка игр по сегментам. Если в 2013 году большая доля рынка приходилась на игровой ПК-рынок (почти 40%), и 37% занимал сегмент консолей, то в 2016 году самым крупным сегментом становится сегмент мобильных игр, генерируя выручку 36,9 млрд долл., что соответствует 37% всего объема мирового рынка игр. То есть в 2016 году доходы в мобильном сегменте впервые превысили доходы от сегмента игр на персональных компьютерах. В рассматриваемый период сегмент мобильных игр демонстрировал колоссальные темпы роста: за 2013-2016 гг. для смартфонов показатель роста составил 28,4%, а для планшетов – 26,9%. [10]

Рисунок 2.6.2 – Объёмы рынка игр в России и мире

Рис. 1. Динамика мирового рынка игр в 2015-2021 гг., млрд долл.

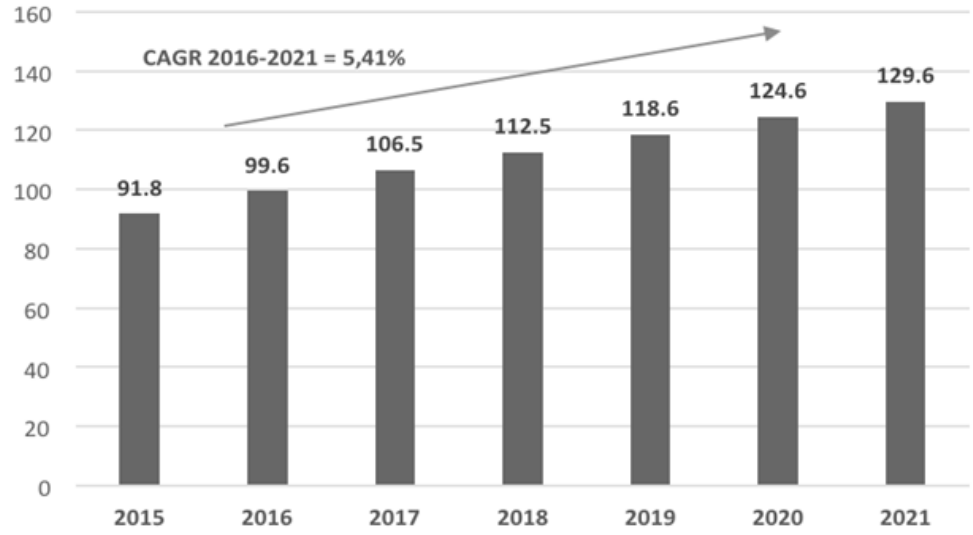

Рисунок 2.6.3 – Динамика мирового рынка в 2015-2021 годах

2.7 Какие перспективы даёт игровая индустрия

В настоящее время устройство на работу в игровую индустрию предоставляет большие возможности и перспективы для роста по следующим причинам:

1. высокий уровень зарплат из-за специфики требований;
2. достаточно большой дефицит хороших специалистов в отрасли;
3. бурный рост индустрии;
4. рост региональных рынков;
5. появление новых технологичных направлений открывают хорошие перспективы для выхода новых компаний на рынок и стимулируют расширение бизнеса в крупных игровых компаниях;
6. при должном упорстве и таланте можно вырасти до международных проектов.

2.8 Индустрия мобильных игр

Индустрия компьютерных игр зародилась в середине 1970-х годов как движение энтузиастов и за несколько десятилетий выросла из небольшого рынка

в популярное течение с годовой прибылью в 9,5 миллиардов долларов в США в 2007 году и 11,7 миллиардов в 2008 году (согласно ежегодным отчётам ESA). На рынке работают как крупные игроки, так и небольшие фирмы, а также независимые разработчики и сообщества.

Рост наблюдается во всех сегментах, но основными движущими силами в этот период стали мобильные игры на двух основных платформах: iOS и Android (см. рис. 2.8.1).

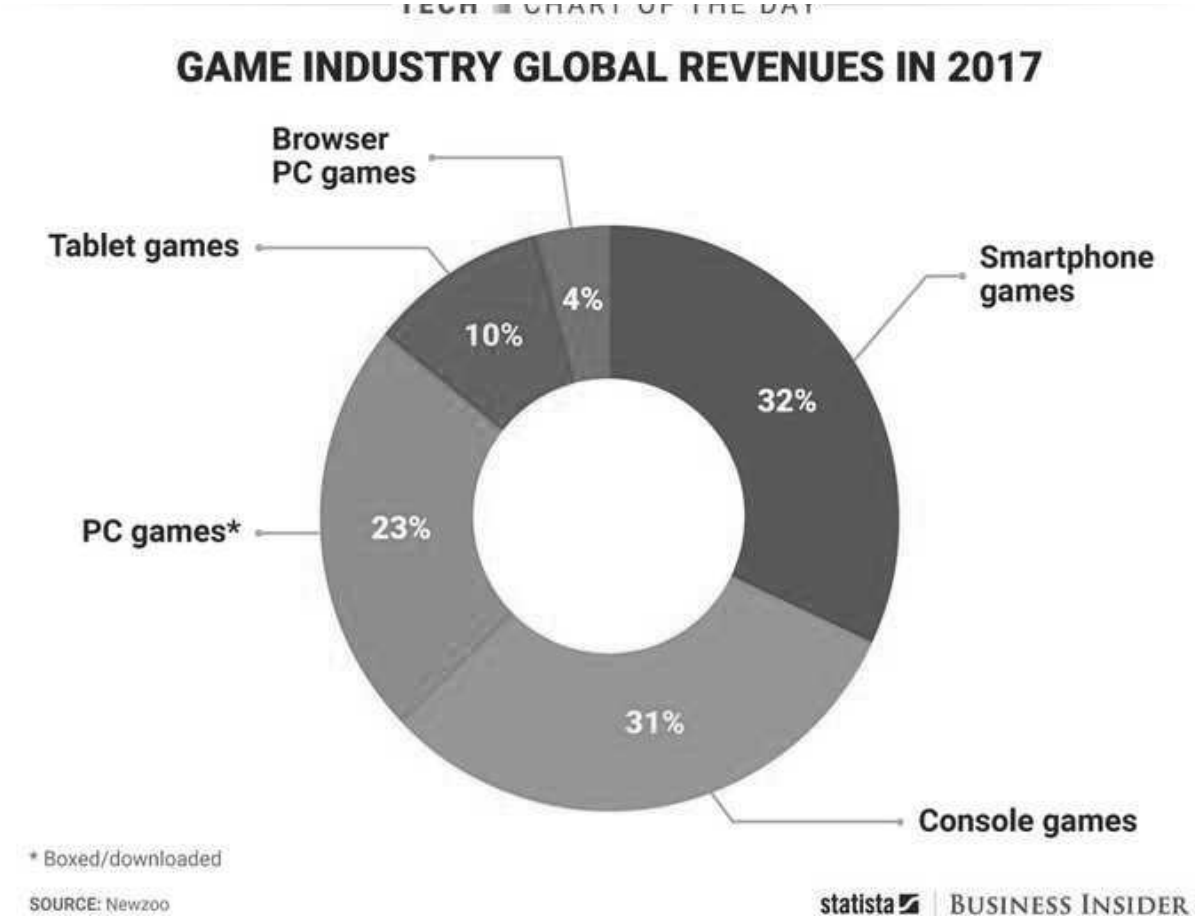

Рисунок 2.8.1 – Рынок игровой индустрии в 2017 году

Аналитики Newzoo утверждают, что рынок игр (компьютерных, мобильных и других) по итогам текущего года достигнет отметки в 137,8 млрд долларов, что на 13,3% превысит показатель 2017 года (см. рис. 2.8.2).

Самым крупным сегментом останутся мобильные игры. Если точнее, по итогам года ожидается, что этот рынок достигнет отметки в 70,3 млрд долларов, то есть на мобильные игры будет приходиться более половины всего рынка. В годовом выражении этот сегмент вырастет на 25,5%. [11]

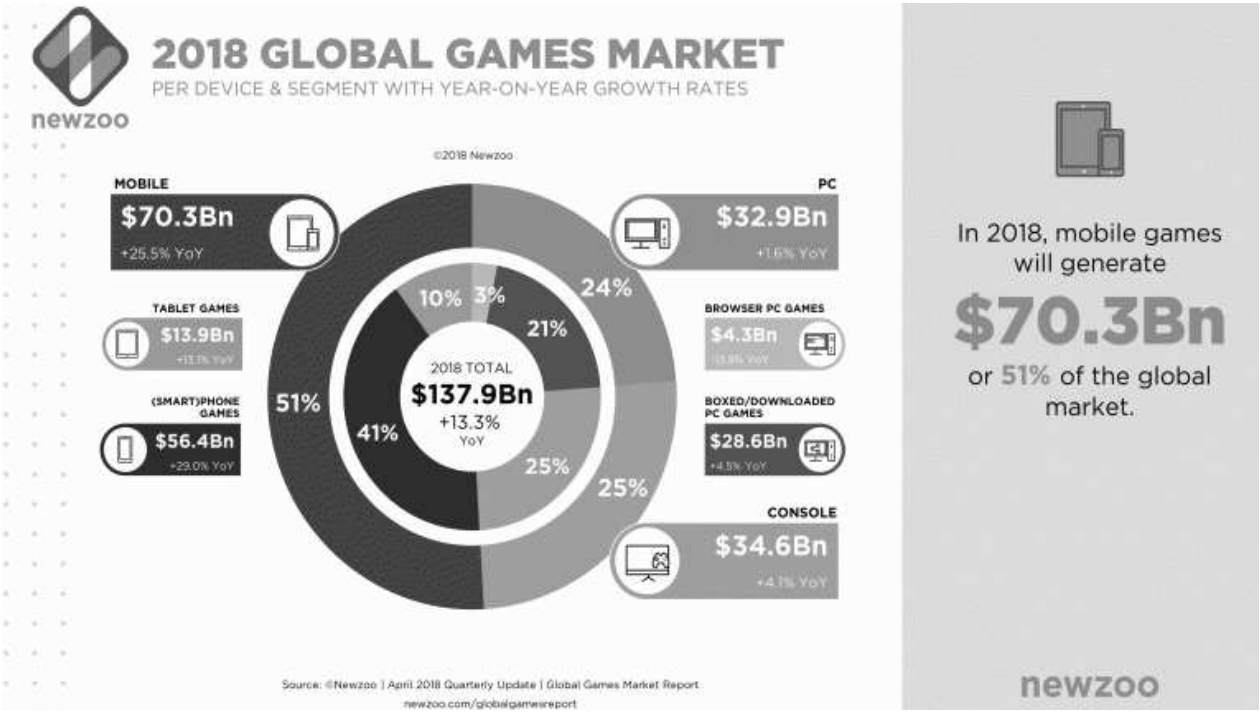

Рисунок 2.8.2 – Рынок игровой индустрии в 2018 году

Оставшиеся 49% пополам разделят консольные и компьютерные игры. Если быть точнее, то первый сегмент займёт 25% рынка (34,6 млрд долларов; рост на 4,1%), а второй – 24% (32,9 млрд долларов; рост на 1,6%).

Что касается более долгосрочных прогнозов, аналитики считают, что к 2021 году игровой рынок вырастет до 180,1 млрд долларов. 59% будут занимать мобильные игры, 19% – компьютерные игры и 22% – видеоигры.

Портал Business Insider со ссылкой на исследовательскую компанию Newzoo сообщает, что по итогам текущего года игры для мобильных принесут 42% от мировой выручки в игровой индустрии. Другие платформы, пользовавшиеся популярностью, принесут значительно меньше.

В последние годы мобильные игры стали настоящим толчком для роста индустрии, во многом благодаря своей доступности для пользователей. Уже не раз отмечалось, что в наше время почти каждый житель цивилизованного мира владеет смартфоном, и почти каждый смартфон способен воспроизводить большинство мобильных игр. Качество таких игр пока не способно конкурировать с крупномасштабными проектами для ПК или консолей, зато они

доступны большему количеству людей. Как следствие, канал распространения у них выше по сравнению с каналом распространения игр для ПК и консолей.

2.9 Движок для класса игр «Симулятор професии»

В последнее время появился спрос на мобильные игры, чаще всего встречающиеся под такими названиями «Симулятор студента», «Симулятор хакера», «Симулятор чиновника» и т.п. Суть этих игр в том, что игроку даётся персонаж, которого он должен провести через череду действий к победе.

Персонаж добивается поставленной цели ежедневными действиями на разных локациях, с помощью которых он развивает свои характеристики. Он может гулять по локациям, покупать товары в магазине, отвечать действиями на какие-либо события, получать из-за этого какие-то последствия, влияющие на его состояние.

Примеры таких игр представлены на рисунках 2.9.1, 2.9.2.

Рисунок 2.9.1 – игра «Симулятор хакера»

Рисунок 2.9.2 – игра «Симулятор строителя»

Как правило, количество действий в день у персонажа ограничено. За день персонаж может потратить все свои действия, после чего он не сможет ничего сделать. Каждое утро количество действий восстанавливается.

У персонажа есть чётко поставленная цель, которой он должен добиться, чтобы он победил в игре. К таким целям можно отнести такие:

- сдать в срок экзамены
- стать миллионером
- и т.п.

Кроме того, он может и проиграть при достижении определённых условий, таких как:

- долго не питался – умер;
- в назначенный срок не закончил проект – не получил гранта;
- и т.п.

Такого вида игры имеют следующие преимущества:

- низкий порог вхождения: сразу понятно, как играть;
- простая цель, которую игрок должен достичь своими действиями;
- для обеспечения сложности существуют условия, из-за которых персонаж может проиграть – поэтому игрок должен их избегать;
- чтобы заинтересовать игрока, по мере продвижения игра может подавать всё новые и новые события, которые открываются по мере прохождения.

Игровой движок – это программный компонент, позволяющий нам создавать и запускать видеоигры. Он предоставляет разработчикам инструменты для создания большинства компонентов игры. [7]

В процессе разработки игры у разработчика остро встаёт вопрос: как сильно будет зависеть игра от движка. В данном случае есть два варианта развития событий:

1. разработчик напишет движок с нуля;
2. разработчик сильно свяжет движок с игрой;
3. разработчик воспользуется уже готовым движком.

Разработчик напишет с нуля или допишет уже существующее решение, позволяющее создать требуемую игру. Процесс разработки новой игры станет легче, так как решение для создания такого класса игр уже было создано. Минусами такого подхода являются то, что при разработке движка следует собрать, систематизировать и проанализировать все требования, предъявляемые к данной предметной области, и то, что на создание движка уходит много времени и ресурсов.

Разработчик может и не создавать отдельно движок, а очень сильно связать игру со средой, в которой он разрабатывает игру. С одной стороны, это может уменьшить время на разработку текущей игры, но с другой стороны, данная практика усложнит расширение функционала и разработку похожей игры в дальнейшем.

Разработчик воспользуется готовым движком для своей цели, слегка видоизменив его под свои нужды. Данный подход сэкономит человеко-часы, так как программисту уже не надо будет разрабатывать движок с нуля.

2.10 Существующие решения

Существует огромное количество игровых движков, которые предоставляют удобные инструменты для разработки игр. Какие-то из них универсальные, в то время как другие могут быть заточены под какой-то определенный жанр. Из известных универсальных игровых движков можно выделить: Unity 3D, Unreal Engine, Game Maker: Studio.

2.11 Анализ существующих решений

2.12.1 Unity 3D

Unity 3D – межплатформенная среда разработки игр. Unity позволяет создавать приложения, работающие под более чем 20 различными операционными системами. [13] На Unity 3D разработано множество игр, в том числе, работающих на консолях, но по уровню графики они почти всегда уступают популярным играм периода, в который выходили. Выпуск Unity состоялся в 2005 году и с того времени идёт постоянное развитие.

Unity 3D имеет очень простой и удобный Drag and Drop интерфейс. Unity 3D поддерживает два языка: C# (наиболее популярный) и Javascipt. [14]

В Unity 4.3 были добавлены 2D спрайты, а также физический движок Box2D, появилась возможность работы с 2D камерой. Для 2D-разработки в Unity есть соответствующие инструменты – sprite creator, sprite editor и sprite packer. [15]

На официальном сайте Unity есть специальный раздел, в котором можно найти статистику по игровым движкам. По этим данным Unity 3D используют более 50% разработчиков видеоигр, 20% принадлежат Unreal Engine, а остальным игровым движкам - 30%. Для разработки небольших 2D или 3D игр Unity 3D подходит по всем параметрам.

Unity развивается очень быстро и с каждым релизом пытается охватить все более и более инновационные функции. Его главные достоинства – простота разработки и кроссплатформенность.

Плюсы Unity 3D:

- легкость в использовании;
- совместимость с любой платформой;
- отличная техподдержка;
- магазин плагинов;
- среда разработки для создания как 2D, так и 3D игр;
- новые инструменты выходят с каждым обновлением.

Минусы Unity 3D:

- ограниченный набор инструментов;
- процесс создания игры занимает много времени;
- некоторые функции требуют покупки;
- не подходит для крупных AAA проектов.

2.12.2 Unreal Engine 4

Unreal Engine 4 – это набор инструментов для разработки игр, имеющий широкие возможности: от создания двухмерных игр на мобильные телефоны до AAA-проектов для консолей. Этот движок использовался при разработке таких игр, как ARK: Survival Evolved, Tekken 7 и Kingdom Hearts III. [15]

Для программирования в Unreal Engine 4 используется язык C++. [16] Движок имеет понятный API и приемлемый период компиляции. Unreal Engine 4 имеет очень мощную и проработанную систему визуального программирования – Blueprints (см. рис. 2.12.2.1), с помощью которой можно достичь практически тех же результатов, что с использованием языка C++.

Blueprints – это система визуального программирования Unreal Engine 4. Она является быстрым способом создания прототипов игр. Вместо построчного

написания кода всё можно делать визуально: перетаскивать ноды (узлы), задавать их свойства в интерфейсе и соединять их «провода». [17]

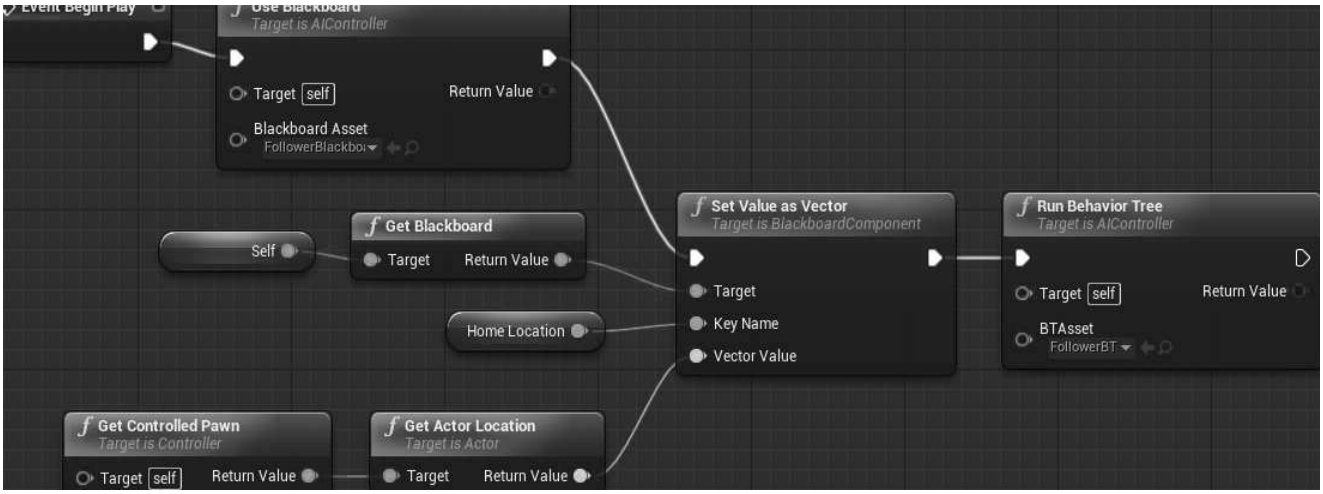

Рисунок 2.12.2.1 – Blueprints

Unreal Engine 4 также поддерживает спрайты в инструменте Paper 2D, но решения, представленные в Unity 3D, имеют лучшую реализацию. [16]

Плюсы Unreal Engine 4:

- полностью бесплатный;
- кроссплатформенность;
- визуальное программирование с помощью Blueprints;
- магазин плагинов;
- отличная техподдержка;
- среда разработки для создания как 2D, так и 3D игр;
- новые инструменты выходят с каждым обновлением.

Минусы Unreal Engine 4:

- если общая прибыль с продукта составляет свыше 3000$, разработчики должны платить 5-процентные отчисления от прибыли с проекта;
- слабые средства для реализации 2D игр;
- сложность некоторых инструментов;
- данная среда разработки требует больше ресурсов для создания игры, чем Unity 3D.

2.12.3 Game Maker: Studio

Game Maker: Studio – это простой игровой движок, позволяющий создать 2D игры для большого числа платформ. [18] GameMaker: Studio является серьёзным развитием своего предшественника, движка Game Maker, и главным отличием является добавление кроссплатформенности, благодаря которой, а также другим существенным доработкам, GameMaker: Studio стал мощным инструментом для профессиональной разработки игр.

Плюсы GameMaker: Studio:

- простое и интуитивное управление;
- собственный язык программирования Game Maker Language (GML), похожий по синтаксису на Java и C#;
- кроссплатформенность;
- магазин плагинов;
- отличная техподдержка;
- среда разработки для создания как 2D, так и 3D игр;
- новые инструменты выходят с каждым обновлением.

Минусы Game Maker: Studio:

- ценовая политика;
- относительно сложно устранять неполадки в игре;
- движок предназначен только для создания 2D игр;
- крайне маленькое сообщество поклонников.

2.12 Вывод

К сожалению, движок, который был бы заточен под разработку такого рода игр, не была найдена в сети Интернет, хотя есть спрос на такого рода игры. Из этого следует, что следует заняться реализацией такого движка, чтобы предоставить разработчикам удобный продукт для их нужд.

Исходя из вышеописанного, необходимо разработать среду разработки или дополнить существующую, с помощью которой можно было бы упростить работу разработчикам такого класса игр.

Вышеупомянутые движки имеют следующие недостатки для реализации графического интерфейса класса игр «Симулятор профессии»:

- избыточность;
- ресурсоёмкость.

Данные движки предназначены для создания широкого диапазона игр. В нашем случае хватит обычного графического интерфейса, реализуемого средой разработки, которая должна удовлетворять следующим требованиям:

- простота реализации;
- возможность создания мобильных приложений для Android и iOS.

Подходящим вариантом явился фреймворк Xamarin, который основан на использовании языка C#, XAML и имеет удобные инструменты для создания графического интерфейса.

Преимущества использования нового решения перед найденными:

- движок в виде отдельной библиотеки может быть подключен к любому движку;
- удобство разработки из-за простого API, обеспечивающего инкапсуляцию;
- расширяемость;
- уменьшение времени разработки и потребления ресурсов на создание и редактирование новой игры.

Из достоинств такого решения вытекает и его недостаток – разработка отдельного движка займёт больше времени, нежели разработка игры, которую нельзя отделить от движка.

3 Сбор и анализ требований заказчика

3.1 Бизнес требования

Первым и самым важным этапом в разработке продукта является сбор бизнес требований. [19] Цель этой работы – определить основные требования бизнеса (исходные данные, истинные цели, которым должен служить продукт и проблемы, которые нужно преодолеть).

3.1.1 Назначение

Движок для класса игр «Симулятор профессии» предоставляет удобный API для разработчиков при создании игр «Симулятор профессии».

Объектом автоматизации является бизнес-процесс «Разработка класса электронных игр». На данный момент при разработке игры необходимо разработать собственный движок или дополнить уже существующий, своё решение для поставленной задачи, для чего требуются время и ресурсы.

Модель текущего бизнес-процесса представлена на рисунке 3.1.1.1.

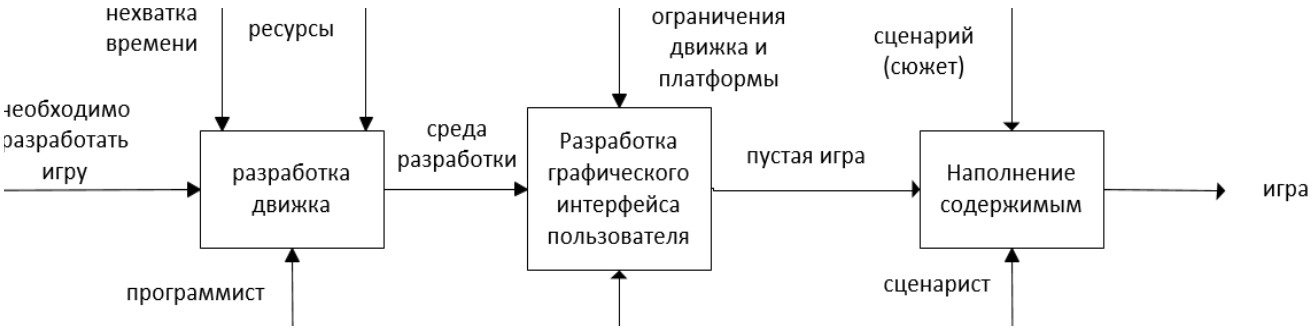

Рисунок 3.1.1.1 – Контекстная диаграмма, модель IDEF0 AS-IS верхний уровень

3.1.2 Стимулы

Данная часть посвящена основным причинам, которые стимулируют принятие решения о создании этого продукта. [19] К стимулам, которые могут подтолкнуть разработчиков использовать готовое решение, можно отнести следующие:

- производственная необходимость – компаниям необходим этот продукт для уменьшения времени разработки и получения за счёт этого большей прибыли;
- оптимизация ресурсов – уменьшение затрат на программистов, за счёт использования готового продукта необходимость в разработке нового решения отпадает;
- конвейер – благодаря готовому решению компании могут поставить такие игры на поток, выпуская их чаще (получая при этом прибыль), чем если бы у них не было готового решения для разработки такого класса игр;
- потребность рынка – необходимо создать удобное решение для разработки игр «Симулятор профессии» в ответ на возросший спрос на такого рода игры.

3.1.3 Цели создания

Данная часть посвящена целям, которые преследуют заинтересованные стороны при разработке продукта. [19] Будут описаны основные преимущества, которые предоставит продукт для бизнеса. Целями создания движка для класса игр «Симулятор профессии» являются следующие тезисы:

- уменьшение времени разработки класса игр «симулятор профессии»;
- достижение большего объёма продаж, чем раньше, за счёт конвейерного выпуска игр;
- экономия человеко-часов, которые в данный момент расходуются на создание удобной для себя среды разработки на одну игру;
- увеличение валовой прибыли для существующего бизнеса;
- уменьшение затрат на программистов;
- увеличение количества выпускаемых игр;
- достижение показателя удовлетворения игроков;
- достижение доминирующего положения на рынке;
- разработка надежный и удобного API;

- получение большего количества положительных отзывов в отраслевых журналах и ресурсах СМИ;
- признание продукта лучшим по удобству пользования в опубликованных отзывах и обзорах;
- соответствие определённым стандартам для такого рода игр.

3.1.4 Целевой сегмент рынка

При выборе сегмента рынка следует определить требования, продиктованные каждым из сегментов к продукту. Выбор целевого сегмента должен основываться на собственных возможностях: бюджете проекта, квалификации аналитиков и разработчиков, наличии возможностей для продвижения продукта. [19]

В данном случае речь идёт о микропредприятиях и малом бизнесе. Как правило, численность сотрудников в таких компаниях составляет от 1 до 100 сотрудников.

Российским законодательством принято следующее определение малого бизнеса – это предприятие с численностью от 16 до 100 сотрудников, выручка от деятельности которого или балансовая стоимость его активов не превышают 400 миллионов рублей по результатам отчетности предыдущего года.

Субъекты предпринимательской деятельности с численностью до 15 сотрудников этим же определением малого бизнеса признаются микропредприятиями и их годовой доход или балансовая стоимость активов не должны превышать 60 млн. рублей. [20]

3.2 Суть проекта

3.2.1 Образ продукта

Пришло время определить очертания будущего продукта, который бы удовлетворял всем требованиям, описанными в предыдущих пунктах. [19]

На этом этапе определяются все основные характеристики, которыми должен обладать будущий продукт, чтобы достичь ранее поставленных целей. А на основе экспертной оценки уже могут быть определены ожидаемые сроки и бюджет продукта.

На протяжении всего жизненного цикла разработки «Образ продукта» должен являться дорожной картой, которой нужно пользоваться, чтобы не сбиться с пути. Для достижения этой цели лучше всего подходит следующий набор тезисов:

- «для» — целевая аудитория покупателей;
- «который» — положение о потребностях или возможностях;
- «этот» — имя продукта;
- «является» — категория продукта;
- «который» — ключевое преимущество, основная причина для покупки или использования;
- «наш продукт» — положение об основном отличии и преимуществе нового продукта.

Продукт создаётся для использования в микропредприятиях и малом бизнесе. Как правило, численность людей в таких компаниях составляет от 1 до 100 сотрудников.

Данное решение носит имя «движок симулятора профессии».

Категорией продукта является категория программного обеспечения под названием игровой движок.

Ключевое преимущество в том, что данное решение предоставляет готовый API, к которому можно обращаться благодаря простым командам, наполняя таким образом игру контентом, расширяя её функционал и добавляя интуитивно понятный графический интерфейс.

Наш продукт является уникальным из-за того, что на рынке ещё не предоставлено конкурирующий решений в данном вопросе. Голубые океаны означают все отрасли, которые на сегодня не существуют, это неизвестные

участки рынка. В алых океанах границы отрасли определены и согласованы, а правила конкуренции всем известны. В них задачей компаний является превосходство над соперниками для того, чтобы перетянуть на себя большую часть спроса. Голубые океаны являются нетронутыми участками рынка, в них конкуренция никому не грозит, они дают возможности расти и получать прибыль.

3.2.2 Список возможностей

Теперь, когда известно кому и зачем нужен продукт, следует определить какую функциональность он будет предоставлять для удовлетворения потребностей будущих пользователей. [19]

Целью создания движка симулятора профессии являются уменьшение времени на разработку класса электронных игр, предоставление удобного инструментария для наполнения контентом за счёт:

- собственного движка с внутренней логикой, к которой программист будет общаться посредством простых команд, не задумываясь о внутреннем устройстве движка;
- движок в виде отдельной библиотеки, которую можно подключить к уже имеющимся средствам разработки для создания приятного графического интерфейса;
- возможность запуска на различных платформах;
- уменьшение времени разработки и потребляемых ресурсов такого класса игр.

3.3 Требования к продукту

3.3.1 Профили пользователей

Для того чтобы движок был удобен пользователям и делал «то, что надо», сначала надо определить, кто же им будет пользоваться.

На практике, даже для домашних продуктов очень сложно определить «среднего пользователя», чтобы на основе его потребностей проектировать

продукт. По этой причине перед началом сбора требований должны быть определены основные профили пользователей. [19]

Пользователями данной системы являются:

1) программисты – специалист, занимающийся непосредственной разработкой графического интерфейса, который будет отображать элементы, основываясь на возвращаемых данных движка;

2) сценаристы – авторы сценария для игры, которые понимают механику такого класса игр и создают для не соответствующие тексты;

3) продюсеры – люди, заинтересованные в получении прибыли от своих вложений в создание такого рода игр;

4) оператор инструментария – человек, преобразующий сценарий в понятные для движка понятия посредством команд движка.

3.3.2 Пользовательские истории, сценарии и варианты использования

Настала очередь начать общение с конечными пользователями и узнать, для достижения каких целей будет использоваться будущий продукт. Лучшим методом для достижения этой цели является сбор пользовательских историй. [19]

Пользовательская история – это вариант использования будущего продукта в конкретной ситуации с целью достижения измеримого результата. Каждая пользовательская история должна приносить пользу. [19]

Важно помнить, что пользовательская история не описывает способ работы с конкретным продуктом и использует только термины предметной области. Она должна быть понятной и родной людям, которые знают процесс, автоматизируемый продуктом.

Программист обладает следующими историями:

- использование простых и понятных функций для обращения к API движка;
- создание объекта инкапсуляции;

- функция для получения свойств, локаций, условий, событий, действий, последствий;
- функция для получения доступных локаций, удовлетворяющих условиям;
- функция для получения доступных событий на локации, удовлетворяющих условиям;
- функция для получения доступных действий у события, удовлетворяющих условиям;
- функция, которая возвращает событие, которое ждёт действия;
- функция, которая отменяет событие, которое ожидает ответа;
- функция, которая отправляет ответ в виде действия на событие, которое ждёт действия;
- создание, редактирование, удаление локаций;
- создание, редактирование, удаление событий;
- создание, редактирование, удаление действий;
- создание, редактирование, удаление свойств персонажа;
- создание, редактирование, удаление условий возникновения тех или иных событий, действий и локаций;
- создание, редактирование, удаление действий на события;
- создание, редактирование, удаление последствий от принятых решений.

3.3.3 Пользовательские требования

3.3.3.1 Функциональные требования

Для группы пользователей «программисты» необходимо соблюдать следующие требования:

- документация по функциям движка;
- программист-разработчик не должен знать, что скрывается внутри движка: он обращает к нему с помощью простых функций, которые инкапсулируют всю логику движка.

3.3.3.2 Требования к программному обеспечению

Для реализации поставленной задачи необходимо соблюдать следующие требования:

- 64-разрядная операционная система Windows 7 или выше с пакетом обновления SP1 или выше;
- платформа .NET Framework 4.7;
- установленный Unity3D, Visual Studio 2013 или другая среда разработки, предоставляющая работу с языком программирования C#.

3.3.3.3 Функциональные требования

Определяются применяемой программой для создания графического интерфейса.

Рекомендуемые требования:

- центральный процессор: двухъядерный 1.8 ГГц или выше, поддержка набора инструкций SSE2;
- оперативная память: 4 Гб или выше;
- монитор: поддержка разрешения 720p (1280 на 720); рекомендуется разрешение 1366x768 и выше;
- свободное место на диске размером в 20-50 Гб;
- видеокарта с поддержкой DirectX 10.

3.3.3.4 Требования к информационному обеспечению

В процессе разработки должны быть использованы документации следующих продуктов: Visual Studio 2017, Unity3D, C#, .Net.

3.3.3.5 Требования к лингвистическому обеспечению

Для создания игры на базе движка для класса игр «Симулятор профессии» необходимо использовать язык программирования C# на платформе .NET Framework 4.7. Руководство пользователя должно быть на русском языке.

4 Обоснования выбора технологий

4.1 Планируемая архитектура продукта

Архитектура движка состоит из следующих компонентов:

- СУБД;
- ORM;
- внешний интерфейс в виде API.

Для обращения к БД будет использоваться СУБД. СУБД – система управления базами данных. СУБД позволяет сосредоточиться на работе с данными, абстрагировавшись от их физического размещения. Также она берет на себя заботу эффективного сохранения данных и их выборки. [21]

Для обращения к СУБД будет использоваться ORM. ORM необходим для упрощения процесса преобразования обычных объектов в реляционные таблицы, точнее для упрощения процесса преобразования между двумя несовместимыми состояниями. [22]

Благодаря API движка программист сможет объединить графический интерфейс с движком, ничего не зная о его архитектуре.

Диаграмма компонентов представлена на рисунке 4.1.1.

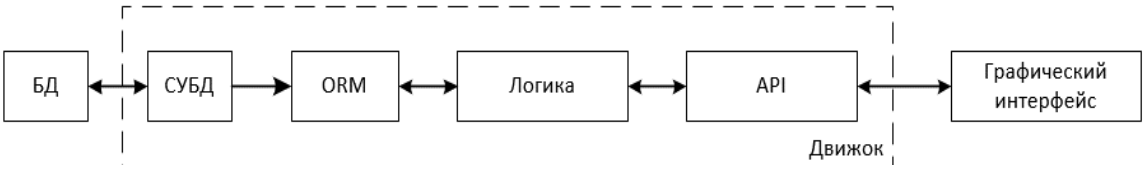

Рисунок 4.1.1 – Диаграмма компонентов

4.2 Язык программирования

4.2.1 Теоретическая часть

4.2.1.1 Высокоуровневые и низкоуровневые языки программирования

Языки программирования можно подразделить на два главных типа: высокоуровневые и низкоуровневые языки программирования. [24]

Язык программирования низкого уровня – язык программирования, близкий по сути к программированию в машинных кодах процессора. [23] Для обозначения машинных команд обычно используют мнемоническое обозначение. Данная практика позволяет запоминать команды не в виде последовательности двоичных нулей и единиц, а в виде сокращений человеческого языка.

Язык программирования высокого уровня – язык программирования, который был разработан для удобства использования программистом. [25] Основная функция этих языков – это абстракция, то есть введение смысловых конструкций, которые кратко описывают структуры данных и операции над ними, описания которых на языке программирования низкого уровня очень длинны и сложны для понимания.

4.2.1.2 Компилируемые и интерпретируемые языки

Код на компилируемом языке при помощи специальной программы под названием компилятор преобразуется в набор команд для данного типа процессора. Далее этот набор записывается в исполняемый модуль, который будет запущен на выполнение как отдельная программа. [26] Иначе говоря, компилятор переводит (компилирует) исходный текст программы с высокоуровневого языка программирования в бинарные коды инструкций процессора.

Если программа написана на интерпретируемом языке, то интерпретатор непосредственно выполняет исходный текст без предварительного перевода по мере выполнения самой программы. При этом программа остаётся на исходном языке и не может быть запущена без интерпретатора.

Компилятор переводит исходный текст на машинный язык сразу и целиком, создавая при этом отдельную исполняемую программу, в то время как интерпретатор выполняет исходный код прямо во время исполнения программы.

4.2.2 Язык C

Язык программирования С был разработан как инструмент для людей, которые практикуются в программировании. В соответствии с этой идеей главной целью автора данного языка было создание удобного языка программирования. [23]

Язык С включает в себя те управляющие конструкции, рекомендованные теорией и практикой программирования. Его концепция основана на нисходящем проектировании, структурном программировании и пошаговой разработке модулей. Язык С достаточно структурирован, чтобы быть способным поддерживать хороший стиль программирования, не связанным с жесткими ограничениями.

4.2.3 Язык C++

Язык С++ был разработан в начале 80-х годов Бьерном Страуструпом с целью избавить программистов от программирования на уже имеющихся языках программирования. [23]

Различие между идеологией языков программирования С и С++ заключается в следующем: программа на языке программирования С отражает способ мышления процессора, а на языке программирования С++ – способ мышления программиста. То есть С++ нацелен на разработку новых типов данных согласно предметной области. Класс в данном случае является ключевым понятием С++. Класс содержит описание данных, требующихся для представления объектов, и набор функций для работы с такими объектами.

Язык программирования С++ является объектно-ориентированным языком, идея объектно-ориентированного программирования которого в последнее время преобладает над традиционным процедурным программированием. Главной целью создания языка программирования было оснащение языка С++ конструкциями, которые позволяют облегчить и ускорить процесс программирования.

Абстракция, инкапсуляция, наследование и полиморфизм являются основными принципами (их ещё называют объектно-ориентированными принципами), которыми обладает язык C++.

4.2.4 Язык Java

Язык программирования Java был разработан как часть проекта создания передового программного обеспечения для различных бытовых приборов. Первым языком, который использовали в этом проекте, был C++, но с ним возникли проблемы. Наилучшим средством борьбы стала смена самого инструмента – языка программирования. Было определено, что необходим платформо-независимый язык программирования, который позволяет создавать программы, которыми можно пользоваться на различных типах процессорах без компиляции под определённые типы процессоров. [23]

Программы на Java транслируются в байт-код, выполняемый на виртуальной java-машине (Java Virtual Machine) – программе, обрабатывающей байт-код и передающей инструкции оборудованию, как в случае с интерпретатором, но с тем отличием, что байт-код, в отличие от кода, представленного текстом, обрабатывается быстрее.

4.2.5 Язык C#

C# (произносится си шарп) – объектно-ориентированный язык программирования, который был разработан группой инженеров компании Microsoft под руководством Андерса Хейлсберга и Скотта Вильтаумота как язык разработки приложений для платформы Microsoft .NET Framework. [27]

Язык программирования C# имеет C-подобный синтаксис, который больше всего похож на языки C++ и Java. Язык имеет статическую типизацию, поддерживает полиморфизм, перегрузку операторов, делегаты, атрибуты, события, свойства, обобщённые типы и методы, итераторы, анонимные функции с поддержкой замыканий, LINQ, исключения, комментарии в формате XML.

Язык C# перенял многое от своих предшественников (языков C++, Pascal, Java) и из-за чего исключает некоторые принципы, зарекомендовавшие себя как проблематичные при разработке программных систем.

4.2.6 Выбор языка программирования

Среди всех перечисленных языков программирования, были выбраны языки C# и Java из-за их особенностей:

- сборка мусора;
- встроенный обработчик событий;
- строгая типизация;
- объектно-ориентированные языки;
- богатые коллекции библиотек;
- пространства имён.

Из-за большей популярности C# был выбран именно он.

Разработка современных приложений все больше тяготеет к созданию программных компонентов в форме автономных и самоописательных пакетов, реализующих отдельные функциональные возможности. Важная особенность таких компонентов – это модель программирования на основе свойств, методов и событий. Каждый компонент имеет атрибуты, предоставляющие декларативные сведения о компоненте, а также встроенные элементы документации. C# предоставляет языковые конструкции, непосредственно поддерживающие такую концепцию работы. Благодаря этому C# отлично подходит для создания и применения программных компонентов.

Ниже представлено несколько функций языка C#, обеспечивающих надежность и устойчивость приложений: [66]

- сборка мусора автоматически освобождает память, занятую уничтоженными и неиспользуемыми объектами;
- обработка исключений дает структурированный и расширяемый способ выявлять и обрабатывать ошибки;

- строгая типизация языка не позволяет обращаться к неинициализированным переменным, выходить за пределы массива или выполнять неконтролируемое приведение типов.

В C# существует единая система типов. Все типы C#, включая типы-примитивы, такие как int и double, наследуют от одного корневого типа object. Таким образом, все типы используют общий набор операций, и значения любого типа можно хранить, передавать и обрабатывать схожим образом. Кроме того, C# поддерживает пользовательские ссылочные типы и типы значений, позволяя как динамически выделять память для объектов, так и хранить упрощенные структуры в стеке.

4.3 Среда разработки

4.3.1 Теоретическая часть

Среда разработки программного обеспечения – это комплекс средств, который используется для разработки программного обеспечения программистами. Как правило, даже самая простая среда разработки имеет редактор текста, интерпретатор или компилятор, отладчик, средство для проверки ошибок. [28]

Обычно среда представляет одну программу, в которой можно производить весь цикл разработки. В состав комплекса также могут входить средства управления проектами, система управления версиями, разнообразные инструменты для упрощения разработки интерфейса пользователя, дополнительные библиотеки и т.п. Современные среды разработки, поддерживающие объектно-ориентированную разработку, могут включать в себя браузер классов, инспектор объектов и диаграмму иерархии классов.

Программист имеет свои приемы разработки, собственные вспомогательные средства и умения. Используя настройки среды разработки, программист может настроить её под себя, чтобы сделать процесс написания кода удобным, а значит и эффективным.

Для создания кода логики движка были выбраны следующие языки: Java и C#. Будем выбирать среды разработки, поддерживающие эти языки.

4.3.2 IntelliJ IDEA

IntelliJ IDEA – интегрированная среда разработки программного обеспечения для многих языков программирования, в частности Java, JavaScript, Python, разработанная компанией JetBrains. [29]

Первая версия данной среды разработки появилась в начале двухтысячных и быстро приобрела популярность как первая среда для Java с широким набором инструментов для рефакторинга, которые позволяли программистам быстро и удобно создавать исходные тексты программ. Дизайн среды был ориентирован на продуктивность работы программистов, позволяя сфокусироваться на задачах архитектуры исходного кода, в то время как среда разработки IntelliJ IDEA отвечала за всё остальное.

4.3.3 Netbeans

NetBeans по праву входит в 5 лучших сред разработки, поддерживающих Java. Данная IDE позволяет разрабатывать мобильные и корпоративные приложения, а также ПО для компьютера. Основное преимущество среды программирования NetBeans – поддержка большого числа технологий (от фиксации ошибок до рефакторинга) и шаблонов без дополнительных настроек. [30]

Последние версии NetBeans IDE поддерживают рефакторинг, профилирование, выделение синтаксических конструкций цветом, автодополнение набираемых конструкций на лету и множество предопределённых шаблонов кода.

Основные характеристики NetBeans IDE:

- рабочая область среды IDE является полностью настраиваемой - существует возможность пользовательской настройки действий, выполняемых с помощью панели, назначения "горячих" клавиш и т.д.;
- производится проверка ошибок во время ввода, отображение вариантов для автозавершения кода и фрагментов документации по требуемому языку программирования;
- редактор NetBeans делает отступы строк, проверяет соответствие скобок и слов, подсвечивает синтаксис исходного кода;
- браузер классов позволяет просматривать иерархию и структуру любого класса Java - отображаются интерфейсы, базовые классы, производные классы и члены классов.

4.3.4 Eclipse

Eclipse является средой разработки с открытым исходным кодом. В отличие от NetBeans, который для создания элементов пользовательского интерфейса использует платформо-независимую библиотеку Swing, Eclipse использует платформо-зависимую библиотеку Standard Widget Toolkit. [31]

Особенности среды разработки Eclipse:

- работает под операционными системами Windows, Linux, Solaris и Mac OS X;
- поддержка множества языков, таких как Java, C, C++, PHP, Perl, Python и других;
- предлагает обширный набор API для создания модулей.

Основным элементом Eclipse является исполняющая среда - Eclipse Runtime, которая отвечает за выполнение расширений и модулей. Она обеспечивает всю базовую функциональность платформы – управление расширениями и обновлениями, взаимодействие с операционной системой, обеспечение работы системы помощи.

Следующим элементом является IDE – она отвечает за управление основными элементами программы, их расположением и настройками, управление проектами, отладку и сборку проектов, поиск по файлам и командную разработку.

4.3.5 Mono

Mono – среда разработки, позволяющая программистам создавать кроссплатформенные приложения. Технология Mono разрабатывается в виде открытой реализации технологии .NET Framework от Microsoft и поддерживает стандарты ECMA-334 (стандарт языка C#) и ECMA-335 (стандарт среды исполнения (Common Language Runtime, CLI). [32]

Mono состоит из следующих компонентов:

- компилятор языка C#;
- среда исполнения Mono состоит из среды исполнения (CLI), компилятора среды исполнения (Just-In-Time, JIT), генератора машинного кода (Ahead-Of-Time, AOT), загрузчика сборок, сборщика мусора, подсистемы управления многопоточностью и компонентов поддержки взаимодействия между сборками и COM;
- базовая библиотека классов;
- библиотека классов Mono.

4.3.6 Visual Studio

Microsoft Visual Studio – среда разработки от компании Microsoft. Данный продукт позволяет разрабатывать как консольные приложения, так и приложения с графическим интерфейсом, веб-сайты, веб-приложения, веб-службы. [33]

Visual Studio включает в себя редактор исходного кода с поддержкой технологии IntelliSense и возможностью простейшего рефакторинга кода. Встроенный отладчик может работать как отладчик уровня исходного кода, так и

отладчик машинного уровня. Остальные инструменты среды разарботки Microsoft Visual Studio включают в себя редактор форм для упрощения создания графического интерфейса приложения, веб-редактор, дизайнер классов и дизайнер схемы базы данных.

Visual Studio позволяет создавать и подключать сторонние дополнения для расширения функциональности, включая добавление поддержки систем контроля версий исходного кода, добавление новых наборов инструментов или инструментов для прочих аспектов процесса разработки программного обеспечения.

4.3.7 Выбор среды разработки

Microsoft Visual Studio – это популярнейшая платформа разработки, где специально для тестировщиков, разработчиков, дизайнеров и руководителей разработки были разработаны инновационные достижения компании Microsoft.

Был сделан выбор в сторону Microsoft Visual Studio из-за следующих характеристик:

- поддержка технологии Intellisense;
- простейший рефакторинг кода;
- встроенный отладчик;
- дизайнер классов;
- редактор форм;
- возможность подключения сторонних фреймворков;
- поддержка контроля версий исходного кода;
- поддержка языка C#.

4.4 Система управления базами данных (СУБД)

4.4.1 Понятие БД и СУБД

Обилие различных данных, которые могут быть помещены в базу данных, ведёт к множеству реализаций того, что можно хранить в БД: личные

данные пользователей, даты, товары, книги, продукты и так далее. В первую очередь это удобно тем, что информацию можно быстро заносить в базу данных и так же быстро ее извлекать при необходимости.

На заре развития информационных технологий все необходимые данные нужно было прописывать в текстовых файлах, что было не очень удобно: избыточность, проблемы с хранением и доступом. Теперь такая необходимость отсутствует – нужная информация может быть запрошена из базы данных при помощи запросов к СУБД. Специальные алгоритмы хранения и поиска информации, которые используются в базах данных, позволяют находить нужные сведения буквально за доли секунд. [35]

СУБД – система управления базами данных. СУБД позволяет сосредоточиться на работе с данными, абстрагировавшись от их физического размещения. Также она берет на себя заботу эффективного сохранения данных и их выборки. [21]

Весь смысл использования базы данных в том, что, когда данных становится много (к примеру, огромное количество таблиц в формате .xls) и их становится невероятно сложно структурировать, а попытки получить полный список тратит очень много ресурсов компьютера, в дело вступают СУБД, которые берут на себя это нелегкое дело, позволяя получать данные из таблиц эффективно. [21]

СУБД освобождает разработчика от задач хранения, модификации и поиска данных. Его дело – указать, какие данные взять, и что с ними сделать. Все остальное сделает сама СУБД.

То есть, упрощённо, БД – это сами данные, представленные в виде совокупности файлов на дисках, с которыми как раз работает СУБД. СУБД – программный продукт, имеющий средства для создания, наполнения, модификации и поиска по базам данных. [36]

4.4.2 Требования к СУБД

Из главных требований к СУБД можно выделить следующие [37]:

- надежность;
- производительность;
- поддержка производителя.

Под надёжностью СУБД подразумевают, что никакие данные не должны пропасть. Сбои должны быть минимизированы и не должны приводить к потерям данных, базы должны быть надёжно защищены от несанкционированного доступа, на режимных объектах могут потребоваться функции шифрования данных, необходимо также обеспечивать регулярное резервное копирование баз данных и возможность восстановления из архива при необходимости.

Под производительностью СУБД подразумевают, что СУБД должна обеспечивать приемлемый уровень производительности для решения возложенных на неё задач.

Поддержка производителем – это уверенность в том, что СУБД будет поддерживаться производителем, и вы не останетесь один на один с проблемой в случае какого-то серьёзного сбоя или сложной ситуации.

4.4.3 Способ доступа к БД

По способу доступа к базам данным можно выделить следующие типы СУБД [38]:

1. клиент-серверные СУБД;
2. файл-серверные СУБД;
3. встраиваемые СУБД.

В клиент-серверных СУБД (Microsoft SQL Server, Oracle, Firebird, PostgreSQL, InterBase, MySQL и др.) вся обработка данных ведётся в одном месте, на сервере, в том же месте, где хранятся данные, то есть доступ к файлам данных имеет только один сервер, одна система – это сама СУБД. Приложения-клиенты при этом посылают запросы на обработку и получение данных из СУБД и получают ответы; приложения-клиенты не имеют непосредственного доступа к файлам данных. Большинство промышленных СУБД являются клиент-серверными.

В файл-серверных СУБД (Paradox, Microsoft Access, FoxPro, dBase и др.), наоборот, приложения имеют общий доступ ко всем файлам базы данных (хранящимся обычно в каком-то разделяемом файловом хранилище) и совместно обрабатывают эти данные. Каждое приложение самостоятельно обрабатывает данные. На данный момент файл-серверная технология считается устаревшей, а её использование в крупных информационных системах – недостатком. Проблема в том, что файл-серверные СУБД не имеют многих преимуществ клиент-серверных, таких как: кэширование данных, параллелизм запросов, высокая производительность и обладают рядом недостатков (сложности с поддержанием целостности базы, восстановлением, блокировками и т.д.), что приводит в свою очередь к пониженной надёжности и производительности.

Встраиваемые СУБД (SQLite, Firebird Embedded, Microsoft SQL Server Compact и др.) входят в состав готового программного продукта, который не требует процедуры самостоятельной установки. Встраиваемые СУБД предназначены для локального хранения данных приложения и не рассчитаны на многопользовательское использование в сети.

4.4.4 Выбор СУБД для поставленной задачи

Исходя из вышеописанных требований к продукту и СУБД, нам необходимо найти такую СУБД, которая удовлетворяла бы следующим характеристикам:

- надёжность;
- поддержка сообщества;
- встраиваемость;
- производительность.

Среди представленных на рынке популярных СУБД, таких как MySQL, PostgreSQL, SQL Server, SQLite, Microsoft Access, Oracle, самым подходящим решением, удовлетворяющим поставленным выше требованиям, оказалась СУБД SQLite.

SQLite – это встраиваемая кроссплатформенная БД, которая поддерживает достаточно полный набор команд SQL.

Слово «встраиваемый» (embedded) означает, что SQLite не использует парадигму клиент-сервер, то есть движок SQLite не является отдельно работающим процессом, с которым взаимодействует программа, а представляет собой библиотеку, с которой работает программа. [41] Такой метод уменьшает расходы, время отклика и упрощает разработку программу.

4.5 Объектно-реляционное отображение

4.5.1 Теоретическая часть

Аббревиатура ORM расшифровывается как «object-relational mapping», что в переводе означает «объектно-реляционное отображение» или «объектно-реляционное преобразование» и означает «технология программирования, которая связывает базы данных с концепциями объектно-ориентированных языков программирования» [42], то есть ORM – это дополнение к БД, которое дает возможность представлять таблицы в виде обычных классов и соответственно обращаться с ними как с простыми классами, не применяя SQL.

ORM необходим для упрощения процесса преобразования обычных объектов в реляционные таблицы, точнее для упрощения процесса преобразования между двумя несовместимыми состояниями. [43]

4.5.2 Парадигма «несоответствия»

На данный момент самыми популярными СУБД являются реляционные СУБД, в то время как для обработки данных самым популярным подходом является объектный подход к проектированию. [43]

Использование ORM решает проблему парадигмы «несоответствия», суть которой в том, что объектные и реляционные модели из-за своей природы не сочетаются друг с другом. Реляционные базы представляют данные в формате таблиц, в то время как объектно-ориентированные языки представляют их как объекты. [44]

Реляционная модель делает фокус на структуре и связях сущностей, а объектная – на поведении и их свойствах. Реляционная модель используется для определения необходимых атрибутов сущности, которые в последующем можно проводить поиск, редактирование. Объектная модель используется для определения свойств сущностей и их поведении.

Какие могут возникнуть проблемы при преобразовании между двумя этими несовместимыми понятиями:

- реляционная модель, как правило, намного детальнее, чем объектная модель, т.е. несколько таблиц могут использоваться для хранения одного объекта;
- для сравнения записей СУБД использует первичные ключи, в то время как объектно-ориентированный язык использует проверку идентичности объектов – первичный ключ;
- реляционная модель не имеет наследования, а объектная имеет;
- принцип доступа к данным в объектно-ориентированных языках отличается от доступа к данным в БД, так как для доступа к данным в объектно-ориентированных языках используются последовательные переходы от родительского объекта к свойствам дочерних элементов и инициализации объектов по необходимости, в то время как реляционная модель использует для этого объединение через JOIN.

4.5.3 Принцип работы ORM

ORM по своей природе выполняет функцию отображения, которое используется для привязки объекта к данным в базе данных. [44] Отображение наглядно показывает, как объект и его свойства связаны с одной или несколькими реляционными таблицами. ORM занимается преобразованием данных между реляционными таблицами и объектами, а также созданием SQL-запросов для манипуляции данными в базе данных из-за изменения объектов.

4.5.4 Преимущества и недостатки использования

Использование ORM избавляет разработчика от необходимости работы с запросами SQL и написания большого количества кода. Весь генерируемый ORM код хорошо проверен и оптимизирован сторонними разработчиками. Основной минус – это потеря производительности. Вопрос о целесообразности использования ORM по большому счету затрагивается только в больших проектах, которые сталкиваются с высокой нагрузкой. В данном случае приходится выбирать то, что более приоритетно – удобство или производительность. Для небольших проектов использование ORM будет куда более оправдано, чем разработка собственных библиотек для работы с БД. [44]

4.5.5 Выбор ORM

Выбор ORM в большинстве своём продиктован выбором языка программирования, у которой есть некоторое количество готовых решений ORM. Самым популярным решением для выбранного языка программирования является Entity Framework.

Entity Framework является продолжением технологии Microsoft ActiveX Data и предоставляет возможность работы с базами данных через объектно-ориентированный код C#. Этот подход предоставляет ряд существенных преимуществ: вам не нужно беспокоиться о коде доступа к данным, вам не нужно знать деталей работы СУБД и синтаксиса языка запросов SQL, вместо этого вы работаете с таблицами базы данных как с классами C#, с полями этих таблиц как со свойствами классов, а синтаксис SQL-запросов, который в .NET раньше нужно было вставлять в код C# в виде команд, заменен на более удобный подход с LINQ. Entity Framework берет на себя обязанности по преобразованию кода C# в SQL-инструкции. [45]

5 Разработка структуры БД

5.1 Жизненный цикл базы данных

Жизненный цикл базы данных (ЖЦБД) – это процесс проектирования, разработки и поддержания базы данных. Данный процесс состоит из семи этапов: [47]

1) предварительное планирование;
2) проверка осуществимости;
3) определение требований;
4) концептуальное проектирование;
5) логическое проектирование;
6) физическое проектирование;
7) оценка работы и поддержка базы данных.

Опишем главные задачи каждого этапа.

Во-первых, предварительное планирование базы данных – этап, который отвечает за переход от разрозненных данных к интегрированным. Информация представляется в виде краткой концептуальной модели данных.

Во-вторых, проверка осуществимости предполагает подготовку документов по следующим темам: есть ли технология, имеются ли персонал, средства и эксперты, окупится ли запланированная база данных.

В-третьих, определение требований. На данном этапе определяются:

- цели базы данных;
- информационные потребности различных структурных подразделений и их руководителей;
- требования к оборудованию;
- требования к программному обеспечению.

В-четвертых, концептуальное проектирование. На данном этапе происходит сбор требований от пользователей, определяются сущности, их атрибуты и связи, после чего создаётся концептуальная модель.

В-пятых, логическое проектирование. На данном этапе происходит выбор типа модели данных. Концептуальная модель преобразовывается в логическую модель, основанную уже на структурах, характерных для выбранной модели данных.

В-шестых, физическое проектирование. На данном этапе логическая модель расширяется характеристиками, которые необходимы для эффективного хранения и поиска на физических носителях.

В-седьмых, оценка и поддержка базы данных. Поддержка осуществляется внесением изменений в имеющую базу данных, созданием прикладных программ.

5.2 Концептуальная модель

Цель этапа концептуального проектирования – создание концептуальной модели данных на основе представлений пользователей о предметной области. Для достижения поставленной цели выполняется ряд следующих действий: [47]

1. определяем сущности;
2. определяем связи между сущностями;
3. определяем атрибуты;
4. создаём ER-модель предметной области;
5. определяем первичные ключи.

Во-первых, определяем сущности. Для определения сущностей определяются объекты, которые могут существовать независимо друг от друга. Такие объекты называются сущностями. Каждой сущности присваивается имя, понятное пользователям.

Во-вторых, определяем связи между сущностями. Определяются только те связи между сущностями, которые необходимы для удовлетворения требований клиентов. Связям присваиваются осмысленные имена, которые выражаются глаголами.

В-третьих, определяем атрибуты сущностей и присваиваем им имена, понятные пользователям.

Следует указать для атрибутов следующие понятия:

- имя атрибута;
- тип значений;
- значение по умолчанию, если оно есть;
- может ли атрибут принимать Null-значение;
- является ли атрибут составным;
- является ли атрибут расчетным.

В-четвертых, создаём ER-модель предметной области. На основе этой модели создается единый образ предметной области.

В-пятых, определяем первичные ключи для сущностей.

5.3 Логическая модель

5.3.1 Этапы логического проектирования

Цель этапа логического проектирования – преобразование концептуальной модели в логическую модель данных на основе выбранной модели данных. Логическая модель не зависит от особенностей выбранной СУБД.

Для проектирования логической модели проделывают следующее: [47]

1. выбираем модель данных;
2. определяем набор таблиц, основываясь на концептуальной модели;
3. осуществляем нормализацию таблиц;
4. проверяем логическую модель данных;
5. определяем требования поддержки целостности данных.

Во-первых, выбираем модель данных. Чаще всего выбирается реляционная модель данных из-за её наглядности табличного представления данных.

Во-вторых, определяем набор таблиц исходя из концептуальной модели. Имя сущности – имя таблицы. Связываем таблицы с помощью механизма первичных и внешних ключей.

В-третьих, осуществляем нормализацию таблиц. На данном этапе проверяется корректность структуры таблиц, посредством применения к ним процедуры нормализации. В результате нормализации получается гибкая база данных, позволяющая легко вносить в нее нужные расширения.

В-четвёртых, проверяем логическую модели данных на возможность выполнения всех транзакций, предусмотренных пользователями. Транзакция – это набор действий, который выполняется отдельным пользователем.

В-пятых, определяем требования поддержки целостности данных. Эти требования представляют собой ограничения, которые вводят для предотвращения помещения в базу данных противоречивых данных.

Рассматриваются следующие ограничения:

- обязательные данные не допускают Null значения;
- ограничения значений атрибутов;
- ограничение целостности – первичный ключ сущности не содержит Null-значений;
- ссылочная целостность – значение внешнего ключа должно обязательно присутствовать в первичном ключе одной из строк таблицы для родительской сущности;
- ограничения в виде бизнес-правил.

5.3.2 Нормализация

Стоит упомянуть о таком процессе как нормализация. Этот процесс необходим при логическом проектировании. Цель нормализации в разделении данных на основе логических взаимоотношений для минимизации дублирования данных. Повторяющиеся данные расходуют дисковое пространство сервера и затрудняют обслуживание. При внесении изменений в такие данные есть

возможность пропустить какие-то из них из-за того, что они не связаны друг с другом, что может привести к возникновению нарушения целостности в базе данных. [48]

Процесс нормализации состоит из пяти этапов, называемых формами. Первый этап, который называется приведением к первой нормальной форме, должен быть выполнен перед приведением базы данных ко второй нормальной форме. Таким же образом, невозможно привести базу данных к третьей нормальной форме, минуя вторую, и т.п.

Разберём три основные нормальные формы.

5.3.3 Первая нормальная форма

В современных СУБД вложенность не допускается, т.е. каждый столбец неделим. Для решения этой проблемы требуется разбить одну таблицу на несколько. Данный процесс называется приведением к первой нормальной форме.

Алгоритм приведения к первой нормальной форме: [49]

1. атрибуты ключа корневого отношения вносятся во множество ключевых атрибутов отношения, который подчиняется непосредственно корневому, с сохранением порядка кортежей;
2. из корневого отношения удаляются составные атрибуты;
3. корневое отношение включается в результирующее множество отношений;
4. из дерева удаляется корень; если среди полученных отношений имеются отношения, состоящие только из простых атрибутов, то они включаются в результирующее множество;
5. для каждого из оставшихся отношений повторять с п.1 до тех пор, пока множество не останется пустым.

5.3.4 Вторая нормальная форма

Первая нормальная форма позволяет уменьшить избыточность данных в строке. Вторая нормальная форма уменьшает избыточность данных в столбцах.

Если ключ отношения состоит из двух и более столбцов, то возможна неполная функциональная зависимость неключевых атрибутов от ключа. При неполной функциональной зависимости значение в неключевом столбце определяется не всем ключом, а его частью.

Второй нормальной формой называется либо данное отношение, если оно в первой нормальной форме и содержит только полные функциональные зависимости неключевых атрибутов от ключей, либо набор проекций данного отношения, обладающий указанными свойствами. [49]

Из-за неполной функциональной зависимости возникают аномалии удаления и включения.

5.3.5 Третья нормальная форма

Третья нормальная форма связана транзитивными зависимостями (см. рис 5.2.5.1). Знак ? обозначает, что зависимость C от B может иметься (стрелка не перечеркнута) или отсутствовать (стрелка перечеркнута). При отсутствии такой зависимости транзитивная зависимость называется строгой.

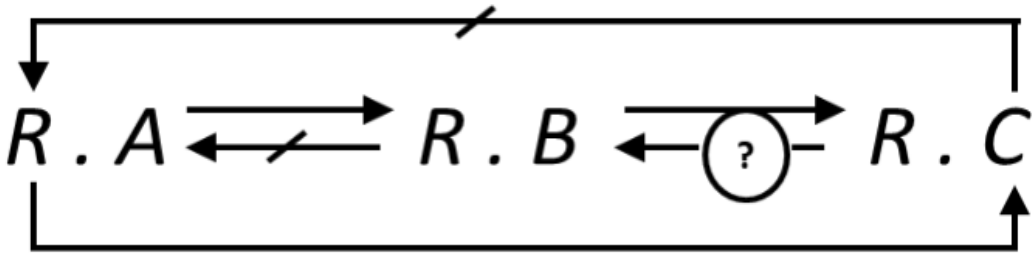

Рис 5.2.5.1 –Транзитивная зависимость

Третьей нормальной формой называется либо данное отношение, если оно во второй нормальной форме и не содержит транзитивных зависимостей неключевых атрибутов от ключей, либо набор проекций данного отношения, обладающий указанными свойствами. [49]

Из-за неполной функциональной зависимости возникают аномалии удаления и включения.

5.4 Физическая модель

Цель этапа физического проектирования – описание конкретной реализации базы данных, которая размещается во внешней памяти компьютера. Цель логического проектирования – что надо сделать, а цель физического проектирования – способ, как это сделать. Выполняются следующие шаги: [48]

1. проектирование таблиц базы данных средствами выбранной СУБД;
2. реализация бизнес-правил в среде выбранной СУБД;
3. проектирование физической организации;
4. разработка стратегии защиты базы данных;
5. организация мониторинга функционирования базы данных и ее настройка.

Во-первых, проектирование таблиц базы данных средствами выбранной СУБД. Осуществляется выбор реляционной СУБД, которую будут использовать для создания базы данных. После изучения документации этой СУБД выполняется проектирование таблиц и схемы их связи в среде СУБД.

Во-вторых, осуществляется реализация бизнес-правил в среде выбранной СУБД.

В-третьих, осуществляется проектирование физической организации базы данных. Определяется лучшая файловая организация для таблиц. Находятся транзакции, которые будут выполняться в проектируемой базе данных. Осуществляется оптимизация производительности базы данных путем создания индексов, которые снижают требования к уровню нормализации таблиц из-за создания специальных дополнительных данных для ускорения поиска по базе данных.

4. Разработка стратегии защиты базы данных. База данных представляет собой ценный корпоративный ресурс, и организации ее защиты уделяется большое внимание.

5. Организация мониторинга функционирования базы данных и ее настройка. После создания физического проекта базы данных организуется

непрерывное слежение за ее функционированием. Полученные сведения об уровне производительности базы данных используются для ее настройки.

5.5 Концептуальная модель данных для движка симулятора профессии

При создании базы данных будут использоваться учебные пособия «Технологии баз данных и знаний» от Толмачёва С.П. [47], «Проектирование реляционных баз данных» от Ярош Е.С., [49]«Введение в системы управления базами данных» от Пушникова А.Ю. [50, 51]

При анализе требований заказчика были выделены следующие сущности (объекты, которые представляют собой неделимое целое, то, что может существовать без других объектов):

1) Главный персонаж, которым будет управлять игрок. Он является главной движущей силой, которая будет двигать игрока к назначенной цели. Персонажем можно ходить по локациям, собирать предметы, выполнять определённые действия в ответ на возникшие события. Персонаж может проиграть или победить. Он обладает атрибутами, характеризирующее его жизненные показатели (здоровье, деньги, имя, уважение, статус (жив, проиграл, выиграл), количество действий, количество доступных действий, условия победы или поражения).

2) Локации, по которым может гулять персонаж. Эти локации представляют собой множество событий, которые могут возникнуть на этих локациях при определённых условиях. У этих локаций есть свои атрибуты, такие как имя локации, описание локации.

3) События, возникающие на локациях и состоящие из таких атрибутов как имя события, описание события. Под событием подразумевается некоторая ситуация, на которую должен ответить персонаж.

4) Действия – ответ персонажа на возникшее событие. Они состоят из имени действия, описания действия, последствий от действия.

5) Последствия – изменение характеристик из-за наступления совершённого действия. Они состоят из определения того, как именно будут меняться атрибуты главного персонажа.

6) Условия – условия возникновения событий, действий и доступности входа на локации. Они состоят из определения того, какие именно показатели атрибутов главного героя будут проверяться.

7) Предметы – некоторые вещи, которые может получить персонаж при определённых условиях.

8) Дополнительные свойства необходимы в тех случаях, когда стандартных атрибутов персонажа не хватает для запланированного сценария; можно привести следующие примеры дополнительных свойств: мускулатура, жир, грусть, радость и подобные.

9) Магазины – специальные события, возникающие на локациях; персонаж может зайти в магазин и купить за определённую сумму нужный себе товар; магазин состоит из следующих атрибутов: имя, описание.

10) Товары – некоторое описание товаров, которые взаимодействуют с изменением атрибутов персонажа, предметов персонажа или дополнительных атрибутов; выполняет те же функции, что и действия, поэтому товары и действия объединяем в одну сущность под названием действия.

Сущности, как правило, связаны между собой. Можно выделить следующие связи:

1) персонаж может гулять по локациям, на которых и происходят все события игры;

2) персонаж владеет предметами, которые он может получать и отдавать;

3) персонаж имеет дополнительные свойства, характеризующие его дополнительные черты, которых нет в основных атрибутах персонажа;

4) на локации возникают события;

5) событие требует действия (ответной меры);

6) действие может делать последствие;

7) на локации могут стоять магазины;

8) магазин продаёт действия (товары);

9) последствие влияет на дополнительные свойства, предметы, основные атрибуты персонажа;

10) условие проверяет дополнительные свойства, предметы, атрибуты персонажа;

11) условие заставляет выполняться действие, событие или локацию;

12) условие влияет на судьбу персонажа, то есть некоторые условия могут являться условиями поражения (конца игры), а другие – условиями победы.

Сущности имеют набор некоторых свойств (атрибутов), характеризующие саму сущность. Можно выделить следующие атрибуты:

1) Персонаж имеет следующие атрибуты:

- Статус – атрибут, показывающий состояние персонажа. Может принимать следующие значения: жив, проиграл, выиграл. Не допускает Null значения.
- Здоровье – атрибут, характеризующий самочувствие персонажа. Принимает значения от 0 до 100. Не допускает Null значения.
- Уважение – атрибут, характеризующий общее отношение общества к персонажу. Принимает значения от 0 до 100. Не допускает Null значения.
- Деньги – атрибут, характеризующий количество средств к существованию у персонажа. Принимает значения от 0 до 65356. Не допускает Null значения. Значение по умолчанию равно нулю.
- Количество действий – атрибут, характеризующий количество действий, которое персонаж получает в начале дня. Принимает значения от 0 до 255. Не допускает Null значения. Значение по умолчанию равно 3.
- Количество доступных действий – атрибут, характеризующий сколько ещё действий персонаж может сделать сегодня. Принимает значения от 0 до 255. Не допускает Null значения.

2) Локация имеет следующие атрибуты:

- Имя – атрибут, характеризующий название локации. Тип данных – строка символов. Не допускает Null значения.
- Описание – атрибут, характеризующий краткое описание объекта сущности. Тип данных – строки символов. Допускает Null значения.

3) Магазин имеет следующие атрибуты:

- Имя – атрибут, характеризующий название объекта сущности. Тип данных – строка символов. Не допускает Null значения.
- Описание – атрибут, характеризующий краткое описание объекта сущности. Тип данных – строки символов. Допускает Null значения.

4) Событие имеет следующие атрибуты:

- Имя – атрибут, характеризующий название объекта сущности. Тип данных – строка символов. Не допускает Null значения.
- Описание – атрибут, характеризующий краткое описание объекта сущности. Тип данных – строки символов. Допускает Null значения.

5) Действие имеет следующие атрибуты:

- Имя – атрибут, характеризующий название объекта сущности. Тип данных – строка символов. Не допускает Null значения.
- Описание – атрибут, характеризующий краткое описание объекта сущности. Тип данных – строки символов. Допускает Null значение.

6) Последствие имеет следующие атрибуты:

- Имя – атрибут, характеризующий название объекта сущности. Тип данных – строка символов. Не допускает Null значения.
- Описание – атрибут, характеризующий краткое описание объекта сущности. Тип данных – строки символов. Допускает Null значения.
- Что изменяем – атрибут, указывающий на дополнительное свойство, предмет или атрибут персонажа. Внешний ключ.

- На что изменить – новое значение атрибута, на который ссылается внешний ключ «что изменяем». Тип данных – целочисленное значение. Допускает Null значение.
- На сколько изменить – значение, которое будет добавлено к значению атрибута, на который ссылается внешний ключ «что изменяем». Тип данных – целочисленное значение. Допускает Null значения.

7) Предмет имеет следующие атрибуты:

- Имя – атрибут, характеризующий название объекта сущности. Тип данных – строка символов. Не допускает Null значения.
- Описание – атрибут, характеризующий краткое описание объекта сущности. Тип данных – строки символов. Допускает Null значение.
- Количество – атрибут, характеризующий количество копий данного объекта сущности. Тип данных – целочисленное значение. Не допускает Null значения. Значение по умолчанию равно нулю.

8) Дополнительное свойство имеет следующие атрибуты:

- Имя – атрибут, характеризующий название объекта сущности. Тип данных – строка символов. Не допускает Null значения;
- Описание – атрибут, характеризующий краткое описание объекта сущности. Тип данных – строки символов. Допускает Null значение.
- Минимальное значение – значение, которое характеризует самое минимальное значение для данного объекта сущности. Тип данных – целочисленное значение. Не допускает Null значения. Значение по умолчанию равно нулю.
- Максимальное значение – значение, которое характеризует самое минимальное значение для данного объекта сущности. Тип данных – целочисленное значение. Не допускает Null значения.
- Значение по умолчанию – значение, которое устанавливается при старте игры у данного объекта сущности. Тип данных – целочисленное значение. Не допускает Null значения.

- текущее значение – значение, которое характеризует измененное значение данного объекта сущности из-за действий в игре. Тип данных – целочисленное значение. Не допускает Null значения.

9) Условие имеет следующие атрибуты:

- Имя – атрибут, характеризующий название объекта сущности. Тип данных – строка символов. Не допускает Null значения.
- Описание – атрибут, характеризующий краткое описание объекта сущности. Тип данных – строки символов. Допускает Null значение.
- Что проверяем – указатель на атрибут персонаж, дополнительное свойство персонажа или предмет. Внешний ключ. Не допускает Null значения;
- Какое должно быть значение – значение, с которым произойдёт сравнение. Тип данных – целочисленное значение. Не допускает Null значения;
- Условие сравнения – логическая операция, которая будет сравнивать текущее значение внешнего ключа «что проверяем» со значением атрибута «какое значение должно быть. Принимает следующие значения: >=, >, <, <=, =, !=. Не допускает Null значения.

Были задокументированы сущности, связи между сущностями. Благодаря этому мы можем составить ER-диаграмму предметной области (см. рисунок 5.5.1).

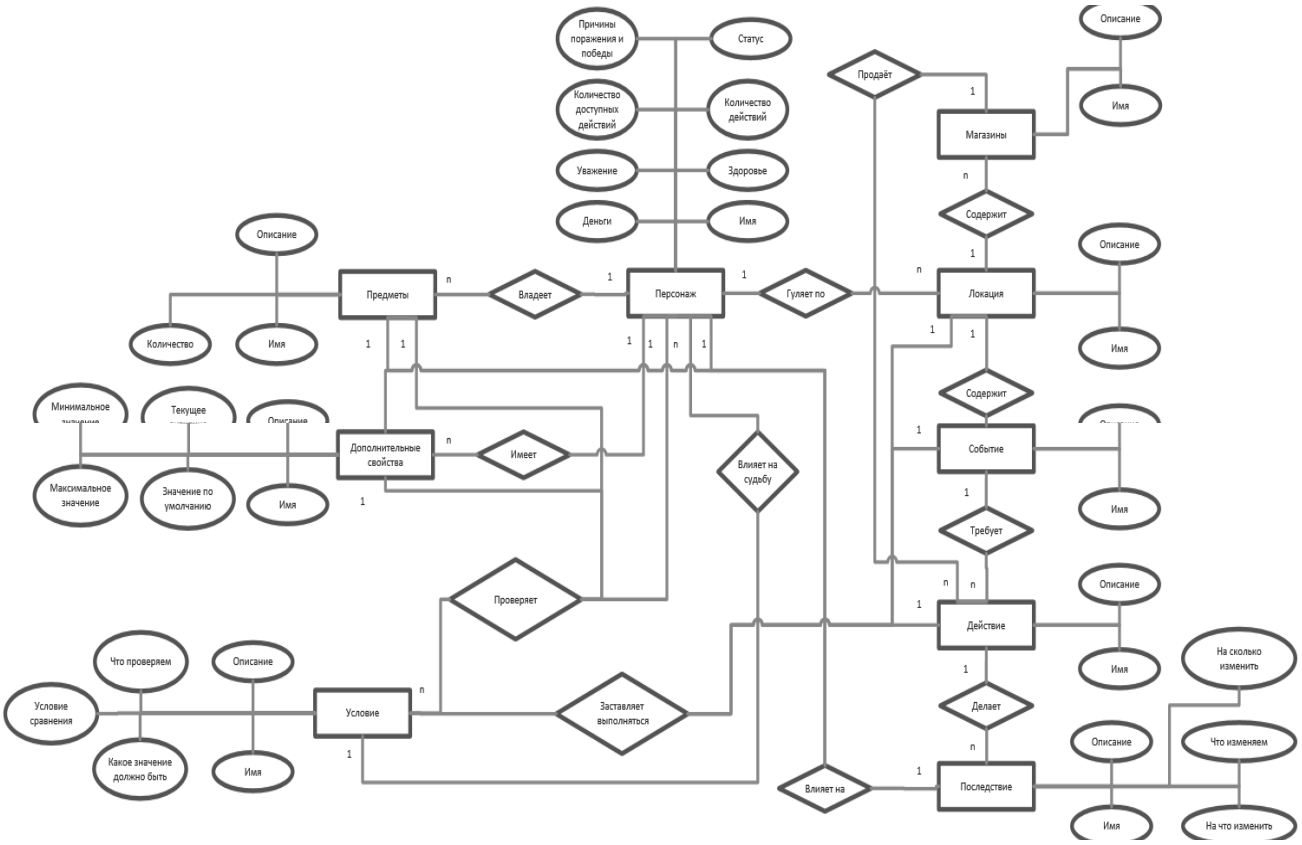

Рисунок 5.5.1 – ER-диаграмма движка симулятора профессии

Первичными ключами для сущностей Локации, События, Действия, Дополнительные действия, Предметы, Последствия, Условия является атрибут ID, представляющий собой автокинкремент.

5.6 Логическая модель данных для движка симулятора профессии

Для реализации хранения данных игры была выбрана реляционная база данных из-за её наглядного табличного представления данных и удобства работы с данными. Были проведена процедура нормализации:

- атрибуты сущности Персонаж были перенесены в отдельную таблицу Свойства;
- была создана таблица Как изменить;
- была создана таблица Какая характеристика;
- была создана таблица Куда Условие;

- была создана таблица, которая связывает таблицы Действия и Последствия;
- была создана таблица Причины победы-поражения;
- была создана таблица, которая связывает таблицы Причины победы-поражения и Условия.

Так же была удалена избыточность, были проверены транзакции, определяемые действиями пользователей в предметной области, были определены ограничения, которые вводятся с целью предотвратить помещение в базу данных противоречивых данных.

Можно выделить следующие таблицы:

1) Таблица Локация имеет следующие атрибуты:

- ID – уникальный идентификатор, по которому можно получить соответствующий кортеж в этой таблице. Имеет следующие ограничения: не допускает Null значения, автоинкремент, первичный ключ, уникальный ключ.
- Имя – значение, характеризующее название локации и являющееся строчкой размером в 255 символов. Не допускает Null значения. Есть проверка на уникальность.
- Описание – значение, характеризующее устройство локации, его суть, и являющееся текстом. Допускает Null значение.

2) Таблица События имеет следующие атрибуты:

- ID – уникальный идентификатор, по которому можно получить соответствующий кортеж в этой таблице. Имеет следующие ограничения: не допускает Null значения, автоинкремент, первичный ключ, уникальный ключ.
- Имя – значение, характеризующее краткое описание события и являющееся текстом. Не допускает Null значения. Есть проверка на уникальность.
- Внешний ключ на соответствующую запись в таблице «Локация», то есть данный столбец ссылается на локацию, которой принадлежит событие. Не допускает Null значения.

- Это магазин? – значение, которое проверяет, не является ли это событие магазином. Не допускает Null значения.

3) Таблица Действия имеет следующие атрибуты:

- ID – уникальный идентификатор, по которому можно получить соответствующий кортеж в этой таблице. Имеет следующие ограничения: не допускает Null значения, автоинкремент, первичный ключ, уникальный ключ.
- Внешний ключ на соответствующую запись в таблице «События», то есть данный столбец ссылается на событие, которому принадлежит действие. Не допускает Null значения.
- Имя – значение, характеризующее краткое описание действия и являющееся текстом. Не допускает Null значения. Есть проверка на уникальность.

4) Таблица Последствия имеет следующие атрибуты:

- ID – уникальный идентификатор, по которому можно получить соответствующий кортеж в этой таблице. Имеет следующие ограничения: не допускает Null значения, автоинкремент, первичный ключ, уникальный ключ.
- Внешний ключ на соответствующую запись в таблице «Какая характеристика». Не допускает Null значения.
- Внешний ключ на соответствующую запись в таблице «Как изменить характеристику». Не допускает Null значения.

5) Таблица «Как изменить характеристику» имеет следующие атрибуты:

- ID – уникальный идентификатор, по которому можно получить соответствующий кортеж в этой таблице. Имеет следующие ограничения: не допускает Null значения, автоинкремент, первичный ключ, уникальный ключ.
- На что изменить – новое значение атрибута. Тип данных – целочисленное значение. Допускает Null значение.
- На сколько изменить – значение, которое будет добавлено к значению атрибута. Тип данных – целочисленное значение. Допускает Null значение.

Данная таблица имеет ограничение:

- Строка может содержать только один из следующих атрибутов «На что изменить» и «На сколько изменить» (исключающее ИЛИ).

6) Таблица «Какая характеристика» имеет следующие атрибуты:

- ID – уникальный идентификатор, по которому можно получить соответствующий кортеж в этой таблице. Имеет следующие ограничения: не допускает Null значения, автоинкремент, первичный ключ, уникальный ключ;
- Внешний ключ на соответствующую запись в таблице «Свойства». Допускает Null значение;
- Внешний ключ на соответствующую запись в таблице «Предметы». Допускает Null значение.

Данная таблица имеет ограничение:

- Строка может содержать только один из следующих атрибутов внешний ключ на соответствующую запись в таблице «Свойства» и внешний ключ на соответствующую запись в таблице «Предметы». (исключающее ИЛИ);

7) Таблица Предметы имеет следующие атрибуты:

- ID – уникальный идентификатор, по которому можно получить соответствующий кортеж в этой таблице. Имеет следующие ограничения: не допускает Null значения, автоинкремент, первичный ключ, уникальный ключ.
- Имя – значение, характеризующее название предмета и являющееся строчкой размером в 255 символов. Не допускает Null значения. Есть проверка на уникальность.
- Описание – значение, характеризующее устройство предмета, его суть, и являющееся текстом. Допускает Null значение.
- Количество – атрибут, характеризующий количество копий данного объекта сущности. Тип данных – целочисленное значение. Не допускает Null значения. Значение по умолчанию равно нулю.

8) Таблица Свойства имеет следующие атрибуты:

- ID – уникальный идентификатор, по которому можно получить соответствующий кортеж в этой таблице. Имеет следующие ограничения: не допускает Null значения, автоинкремент, первичный ключ, уникальный ключ.
- Имя – значение, характеризующее название свойства и являющееся строчкой размером в 255 символов. Не допускает Null значения. Есть проверка на уникальность.
- Описание – значение, характеризующее суть свойства и являющееся текстом. Допускает Null значение.
- Минимальное значение – значение, которое характеризует самое минимальное значение для данного объекта сущности. Тип данных – целочисленное значение. Не допускает Null значения. Значение по умолчанию равно нулю.
- Максимальное значение – значение, которое характеризует самое минимальное значение для данного объекта сущности. Тип данных – целочисленное значение. Не допускает Null значения.
- Значение по умолчанию – значение, которое устанавливается при старте игры у данного объекта сущности. Тип данных – целочисленное значение. Не допускает Null значения.
- Текущее значение – значение, которое характеризует измененное значение данного объекта сущности из-за действий в игре. Тип данных – целочисленное значение. Не допускает Null значения.

В этой таблице должны обязательно храниться следующие записи:

- здоровье – атрибут, характеризующий самочувствие персонажа, принимает значения от 0 до 100;
- уважение – атрибут, характеризующий общее отношение общества к персонажу, принимает значения от 0 до 100;
- деньги – атрибут, характеризующий самочувствие персонажа, принимает значения от 0 до 65356;

- количество действий – атрибут, характеризующий количество действий, которое персонаж получает в начале дня;
- количество доступных действий – атрибут, характеризующий сколько ещё действий персонаж может сделать сегодня;
- статус – атрибут, показывающий состояние персонажа и имеет следующие ограничение: может принимать следующие значения: 0 (жив), -1 (проиграл), 1 (выиграл), не допускает Null значения.

9) Условие имеет следующие атрибуты:

- ID – уникальный идентификатор, по которому можно получить соответствующий кортеж в этой таблице. Имеет следующие ограничения: не допускает Null значения, автоинкремент, первичный ключ, уникальный ключ.
- Внешний ключ на соответствующую запись в таблице «Какая характеристика». Не допускает Null значения.
- какое должно быть значение – значение, с которым произойдёт сравнение. Тип данных – целочисленное значение. Не допускает Null значения.
- условие сравнения – логическая операция, которая будет сравнивать текущее значение внешнего ключа «что проверяем» со значением атрибута «какое значение должно быть. Принимает следующие значения: >=, >, <, <=, =, !=. Не допускает Null значения.

10) Куда Условие имеет следующие атрибуты:

- ID – уникальный идентификатор, по которому можно получить соответствующий кортеж в этой таблице. Имеет следующие ограничения: не допускает Null значения, автоинкремент, первичный ключ, уникальный ключ.
- Внешний ключ на соответствующую запись в таблице «События». Не допускает Null значения.
- Внешний ключ на соответствующую запись в таблице «Действия». Не допускает Null значения.
- Внешний ключ на соответствующую запись в таблице «Локации». Не допускает Null значения.

Данная таблица имеет одно ограничение:

- В строчке может быть только один из этих атрибутов: внешний ключ на соответствующую запись в таблице «Действия», внешний ключ на соответствующую запись в таблице «События», внешний ключ на соответствующую запись в таблице «Локации».

11) Таблица, которая связывает таблицы Действия и Последствия. Имеет следующие атрибуты

- Внешний ключ на соответствующую запись в таблице «Действия». Не допускает Null значения;
- Внешний ключ на соответствующую запись в таблице «Последствия». Не допускает Null значения.

12) Таблица «Причины победы-поражения» имеет следующие атрибуты:

- ID – уникальный идентификатор, по которому можно получить соответствующий кортеж в этой таблице. Имеет следующие ограничения: не допускает Null значения, автоинкремент, первичный ключ, уникальный ключ;
- Сообщение – значение, характеризующее текст, который увидет игрок при достижении причины победы или поражения. Не допускает Null значения. Есть проверка на уникальность.
- Тип – значение типа bool, определяющее тип причины (причина поражения или победы). Не допускает Null значения.

13) Таблица, которая связывает таблицы «Причины победы-поражения» и Условия. Имеет следующие атрибуты

- Внешний ключ на соответствующую запись в таблице «Причины победы-поражения». Не допускает Null значения.
- Внешний ключ на соответствующую запись в таблице «Условия». Не допускает Null значения.

5.7 Физическая модель данных движка симулятора профессии

Для реализации физической модели данных была выбрана СУБД SQLite. Для создания необходимых таблиц, атрибутов, ограничений и бизнес-правил будет применяться программное обеспечение SQLite Expert Professional.

SQLite Expert Professional – мощный визуальный инструмент, который позволяет Вам легко работать с базами данных SQLite3. Содержит эффективные инструменты управления данными. Предоставляет все необходимые инструменты для работы: просмотр, создание, редактирование, копирование, удаление, экспорт объектов, управление учетными записями базы, управление функциями и процедурами.

Пользовательский интерфейс программы SQLite Expert Professional представлен на рисунке 5.7.1. В левой части представлен обозреватель баз данных. В правой части представлены инструменты для работы с выбранной базой данных.

Следует отметить следующие инструменты:

1) Schema – информация для чтения о компонентах таблицы: поля, индексы, триггеры, ограничения (первичный ключ, внешний ключ, проверка на уникальность, проверочное ограничение). Данный инструмент изображён на рисунке 5.7.2.

2) Data – данные таблицы, которая была выбрана в обозревателе баз данных. Данный инструмент изображён на рисунке 5.7.3.

3) DDL – сформированный SQL код описания таблицы и её компонентов. Данный инструмент изображён на рисунке 5.7.4.

4) Design – инструмент для редактирования таблицы. Состоит из следующих вкладок:

- General – имя таблицы;
- Columns – редактор столбцов;
- PrimaryKey – редактор первичных ключей ;
- Indexes – редактор индексов;
- Triggers – редактор триггеров;

- ForeignKeys – редактор внешних ключей;
- UniqueContraints – редактор ограничений уникальности;
- CheckConstraints – редактор проверочных ограничений.

Данный инструмент изображён на рисунке 5.7.5.

5) SQL Builder – конструктор баз данных. Данный инструмент изображён на рисунке 5.7.6.

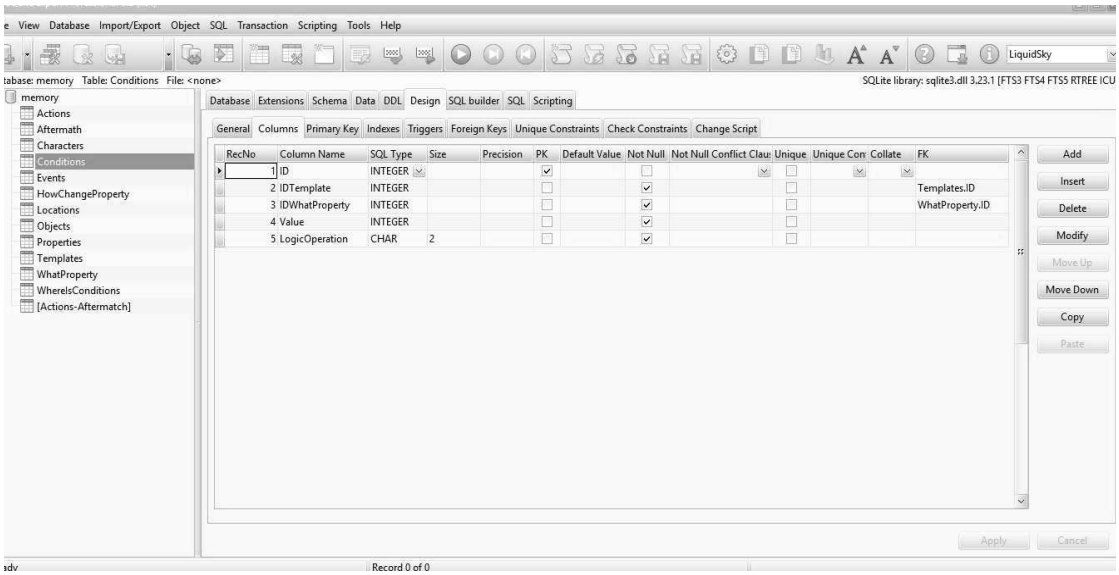

Рисунок 5.7.1 – Пользовательский интерфейс программы SQLite Expert Professional

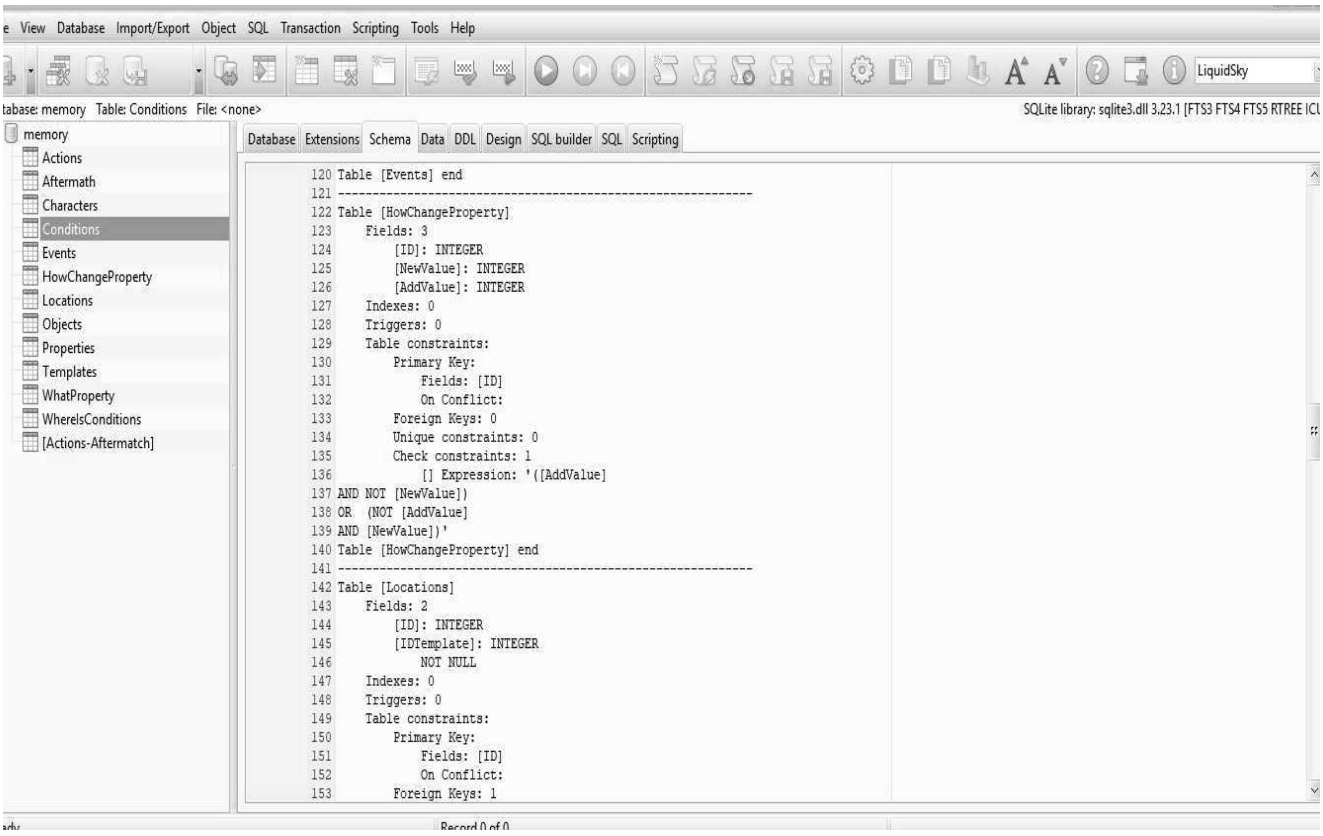

Рисунок 5.7.2 – Информация о базе данных в программе SQLite Expert Professional

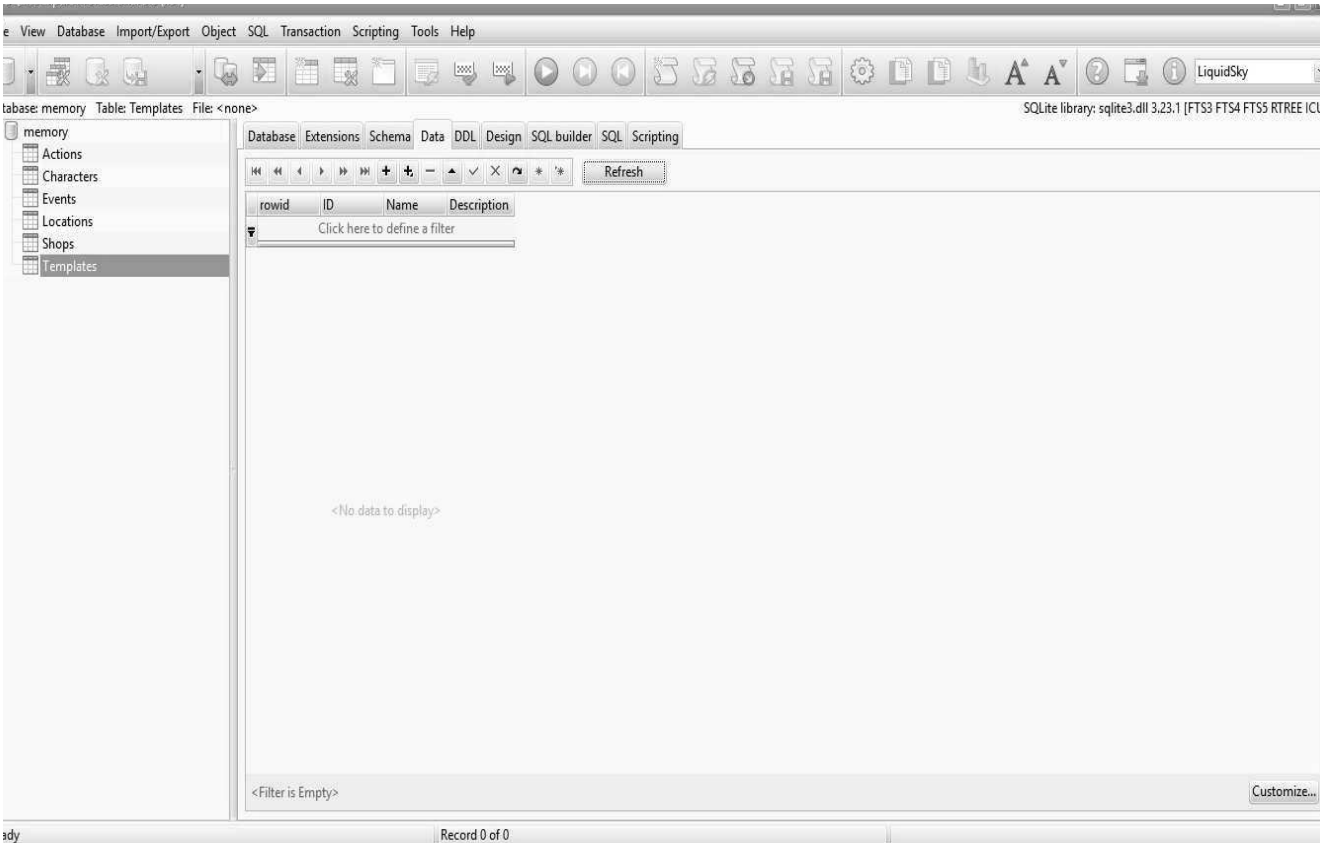

Рисунок 5.7.3 – Данные таблицы в программе SQLite Expert Professional

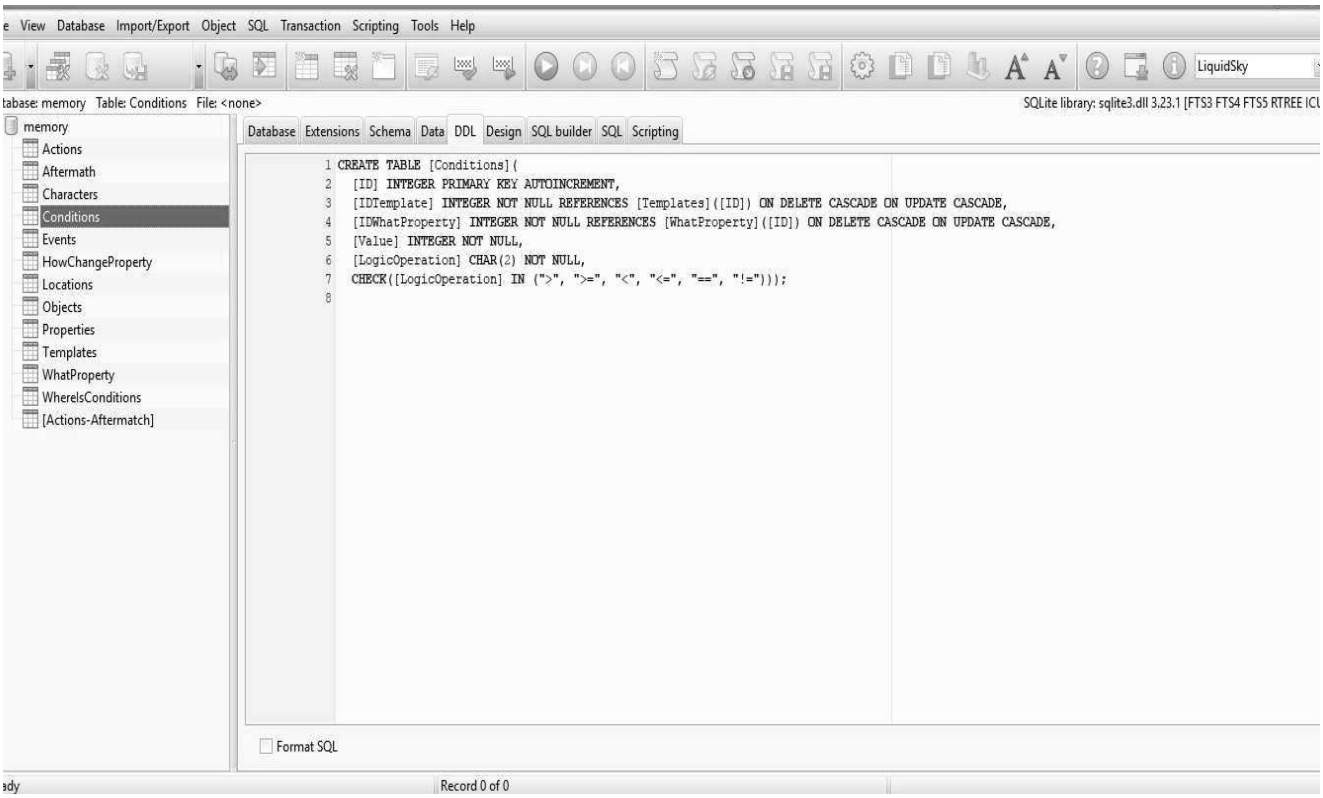

Рисунок 5.7.4 – Информация о базе данных в программе SQLite Expert Professional

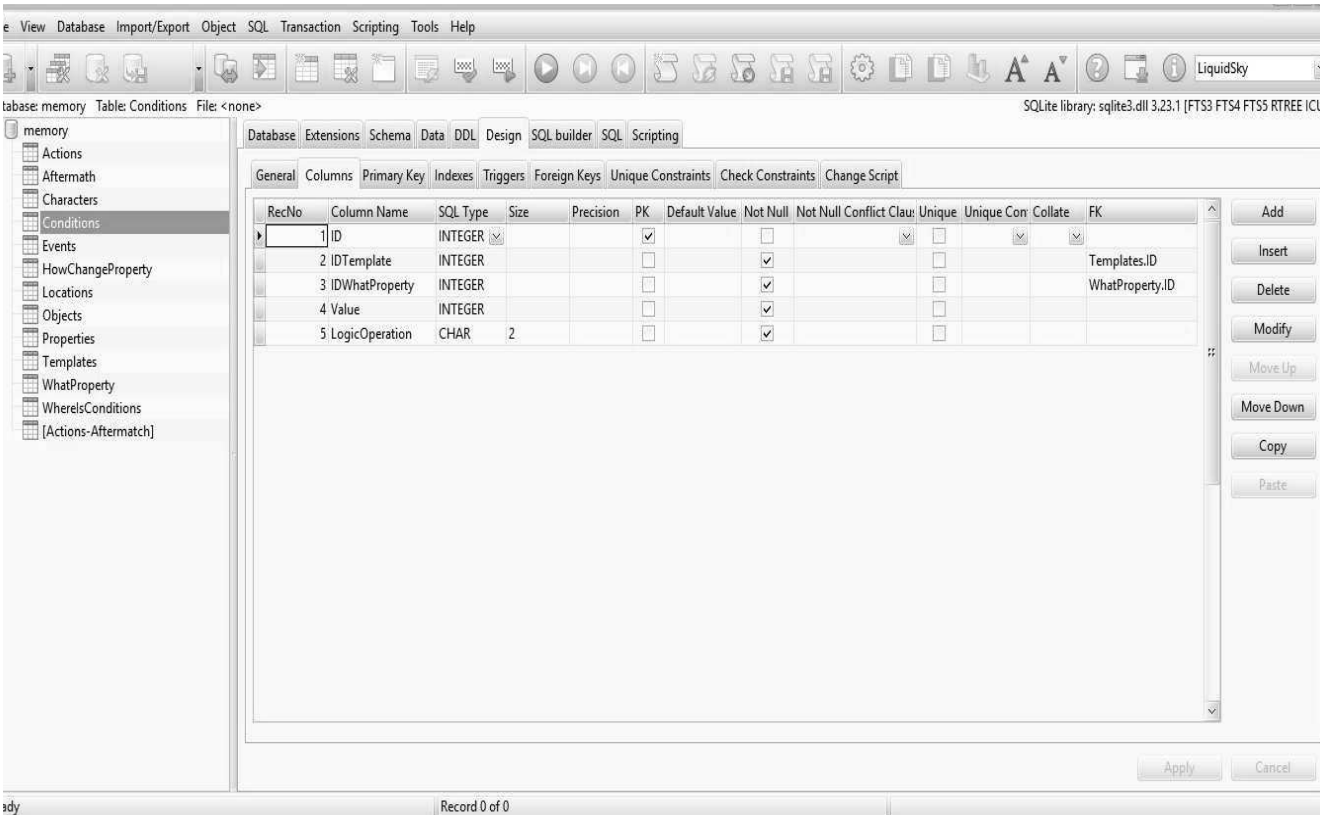

Рисунок 5.7.5 – Редактор таблиц SQLite Expert Professional

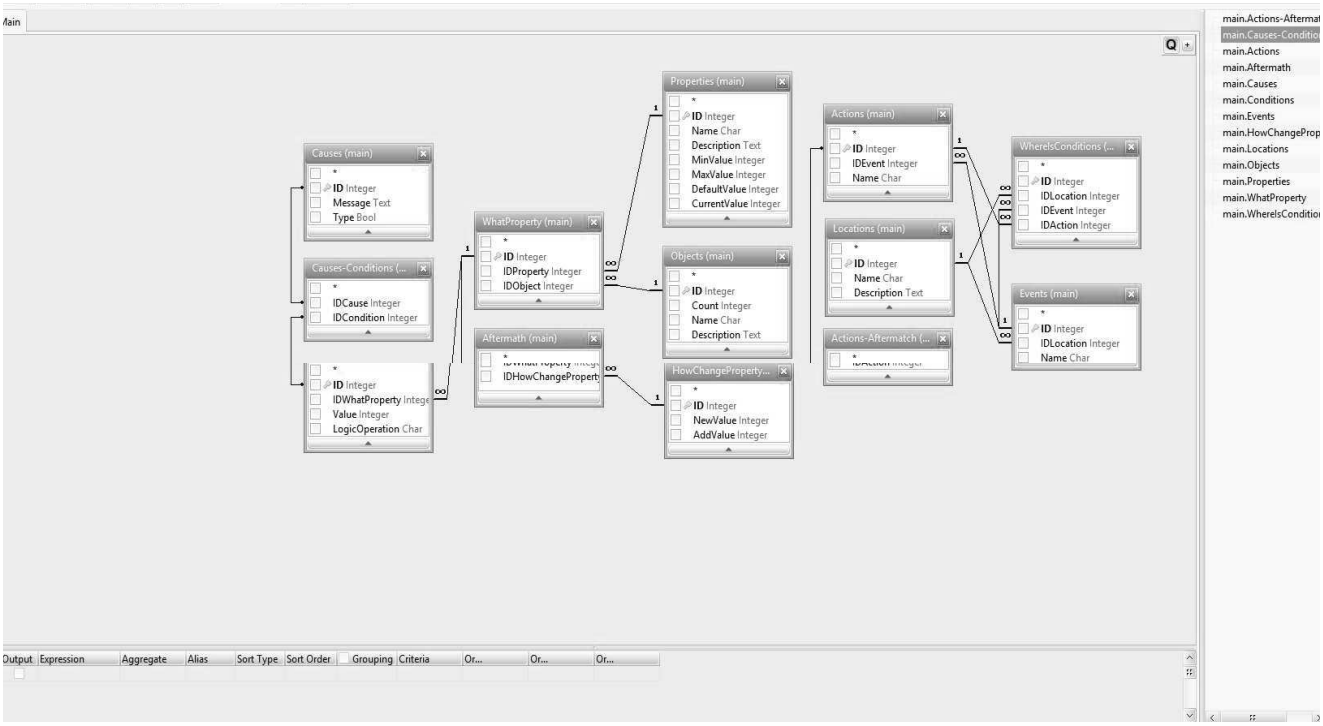

Рисунок 5.7.6 – Конструктор баз данных SQLite Expert Professional

При проектировании физической модели получили следующий таблицы:

1. таблица Actions характеризует действия, которые имеют события;
2. таблица Aftermatch характеризует последствия, которые бывают от действий;
3. таблица Causes характеризует причины поражения или победы персонажа;
4. таблица Conditions характеризует условия возникновения локации, события или действия;
5. таблица Events характеризует события, которые возникают на локации;
6. таблица HowChangeProperty характеризует то, как изменится характеристика;
7. таблица Locations характеризует локации, по которым может гулять персонаж;
8. таблица Objects характеризует предметы, которые имеет персонаж;
9. таблица Properties характеризует свойства персонажа;

10. таблица WhatProperty характеризует ID свойства или предмета, который будет участвовать в условии или последствии;

11. таблица WhereIsConditions характеризует ID локации, действия или события, на которое будет накладываться некоторое условие;

12. таблица ActionsAftermatch связывает таблицы Actions и Aftermatch;

13. таблица CausesConditions связывает таблицы Causes и Conditions.

5.7.1 DDL-код таблиц

```
CREATE TABLE [Actions](
  [ID] INTEGER PRIMARY KEY AUTOINCREMENT,
  [IDEvent] INTEGER NOT NULL REFERENCES [Events]([ID]) ON DELETE CASCADE ON UPDATE CASCADE,
  [Name] CHAR(255) NOT NULL);

CREATE TABLE [Aftermath](
  [ID] INTEGER PRIMARY KEY AUTOINCREMENT,
  [IDWhatProperty] INTEGER NOT NULL REFERENCES [WhatProperty]([ID]) ON DELETE CASCADE ON UPDATE CASCADE,
  [IDHowChangeProperty] INTEGER NOT NULL REFERENCES [HowChangeProperty]([ID]) ON DELETE CASCADE ON UPDATE CASCADE);

CREATE TABLE [Causes](
  [ID] INTEGER PRIMARY KEY AUTOINCREMENT REFERENCES [Templates]([ID]) ON DELETE CASCADE ON UPDATE CASCADE,
  [Message] TEXT NOT NULL UNIQUE,
  [Type] BOOL NOT NULL);

CREATE TABLE [Conditions](
  [ID] INTEGER PRIMARY KEY AUTOINCREMENT,
```

```
        [IDWhatProperty] INTEGER NOT NULL REFERENCES
[WhatProperty]([ID]) ON DELETE CASCADE ON UPDATE CASCADE,
        [Value] INTEGER NOT NULL,
        [LogicOperation] CHAR(2) NOT NULL,
        CHECK([LogicOperation] IN (">", ">=", "<", "<=", "==", "!=")));

      CREATE TABLE [Events](
        [ID] INTEGER PRIMARY KEY AUTOINCREMENT,
        [IDLocation] INTEGER NOT NULL REFERENCES [Locations]([ID]) ON
DELETE CASCADE ON UPDATE CASCADE,
        [Name] CHAR(255) NOT NULL UNIQUE,
        [IsShop] BOOLEAN NOT NULL DEFAULT 0);

      CREATE TABLE [HowChangeProperty](
        [ID] INTEGER PRIMARY KEY AUTOINCREMENT,
        [NewValue] INTEGER,
        [AddValue] INTEGER,
        CHECK((AddValue AND NOT NewValue) OR (NOT AddValue AND
NewValue)));

      CREATE TABLE [Locations](
        [ID] INTEGER PRIMARY KEY AUTOINCREMENT,
        [Name] CHAR(255) NOT NULL UNIQUE,
        [Description] TEXT);

      CREATE TABLE [Objects](
        [ID] INTEGER PRIMARY KEY AUTOINCREMENT,
        [Count] INTEGER NOT NULL DEFAULT 0,
        [Name] CHAR(255) NOT NULL UNIQUE,
```

```
    [Description] TEXT);

  CREATE TABLE [Properties](
    [ID] INTEGER PRIMARY KEY AUTOINCREMENT,
    [Name] CHAR(255) NOT NULL UNIQUE,
    [Description] TEXT,
    [MinValue] INTEGER NOT NULL DEFAULT 0,
    [MaxValue] INTEGER NOT NULL,
    [DefaultValue] INTEGER NOT NULL,
    [CurrentValue] INTEGER NOT NULL);

  CREATE TABLE [WhatProperty](
    [ID] INTEGER PRIMARY KEY AUTOINCREMENT,
    [IDProperty] INTEGER REFERENCES [Properties]([ID]) ON DELETE
CASCADE ON UPDATE CASCADE,
    [IDObject] INTEGER REFERENCES [Objects]([ID]) ON DELETE
CASCADE ON UPDATE CASCADE,
    CHECK(([IDProperty]
  AND NOT [IDObject])
  OR  (NOT [IDProperty]
  AND [IDObject])));

  CREATE TABLE [WhereIsConditions](
    [ID] INTEGER PRIMARY KEY AUTOINCREMENT,
    [IDLocation] INTEGER REFERENCES [Locations]([ID]) ON DELETE
CASCADE ON UPDATE CASCADE,
    [IDEvent] INTEGER REFERENCES [Events]([ID]) ON DELETE
CASCADE ON UPDATE CASCADE,
```

```
[IDAction] INTEGER REFERENCES [Actions]([ID]) ON DELETE CASCADE ON UPDATE CASCADE,
 CHECK(
((([IDLocation]
AND NOT [IDEvent])
OR  (NOT [IDLocation]
AND [IDEvent]))
AND NOT [IDAction])
OR  (NOT (([IDLocation]
AND NOT [IDEvent])
OR  (NOT [IDLocation]
AND [IDEvent]))
AND [IDAction])
));

CREATE TABLE [ActionsAftermatch](
 [IDAftermatch] INTEGER NOT NULL REFERENCES [Aftermath] ON DELETE CASCADE ON UPDATE CASCADE,
 [IDAction] INTEGER NOT NULL REFERENCES [Actions] ON DELETE CASCADE ON UPDATE CASCADE);

CREATE TABLE [CausesConditions]
(
 [IDCause] INTEGER NOT NULL REFERENCES [Causes] ON DELETE CASCADE ON UPDATE CASCADE,
 [IDCondition] INTEGER NOT NULL REFERENCES [Conditions] ON DELETE CASCADE ON UPDATE CASCADE);
```

5.8 Вывод

Благодаря тому, что база данных была спроектирована с помощью трёх моделей (концептуальная, логическая, физическая модели), каждая из которых выполняла свою определённую роль и выводилась из другой с помощью определённых требований, были предотвращены возможные проблемы, которые могли бы возникнуть при разработке базы данных без такого подхода.

К таким проблемам можно отнести:

- ненормализованные таблицы (аномалии удаления и вставки, избыточность данных);
- нарушение целостности (нарушение бизнес-правил и ограничений);
- невозможность выполнения транзакций;
- незаложенные требования в базу данных.

6 Разработка архитектуры движка

6.1 Принципы объектно-ориентированного программирования

Все основанные на объектах языки (C#, Java и т.п.) должны отвечать четырём основным принципам объектно-ориентированного программирования (ООП): [65]

- инкапсуляция;
- наследование;
- полиморфизм;
- абстракция.

Инкапсуляция позволяет скрыть внутренний интерфейс. Обычно класс имеет некоторые внутренние методы, поля, объекты, свойства, доступ к которым для пользователя необходимо скрыть.

Наследование позволяет создавать новый класс на базе другого. Класс, на базе которого создается новый класс, называется базовым, а базирующийся новый класс – наследником. Все свойства и методы базового класса при наследовании переходят в класс наследник.

Полиморфизм – это способность объектов с одним интерфейсом иметь различную реализацию.

Абстракция позволяет выделять из некоторой сущности только необходимые характеристики и методы, которые в полной мере описывают объект данной предметной области.

6.2 Паттерны проектирования

Поиск идеального инструмента, языка, принципа или методологии разработки – это святой Грааль в разработке ПО. Все хотят найти идеальный инструмент, позволяющий справиться со сложностью современных систем и навести порядок в том хаосе, который творится в мире программирования. Но,

может быть, вместо того, чтобы каждый раз хвататься за что-то новое как за спасительную соломинку, стоит понять, что за этой соломинкой скрыто? Ведь если присмотреться, то новый инструмент очень часто оказывается лишь новой оберткой, в которую завернуты старые идеи.

Паттерны проектирования остаются важным инструментом в арсенале разработчика, поскольку они опираются на фундаментальные принципы проектирования. [52]

Паттерны проектирования можно разделить на три типа:

1) порождающие паттерны отвечаю за удобное и безопасное создание новых объектов или семейств объектов;

2) структурирующие паттерны определяют различные сложные структуры, которые изменяют интерфейс или реализацию уже существующих объектов, позволяя облегчить или оптимизировать программу;

3) паттерны поведения отвечают за эффективное взаимодействие между объектами.

6.2.2 Порождающие паттерны

Порождающие паттерны – паттерны, которые создают новые объекты или позволяют получить доступ к уже существующим. То есть те шаблоны, по которым создать новый качественный автомобиль. [52]

6.2.2.1 Одиночка

Часто в системе могут существовать сущности только в единственном экземпляре, например, ведение системного журнала сообщений или драйвер дисплея. В таких случаях необходимо уметь создать единственный экземпляр некоторого типа, предоставить к нему доступ извне и запретить создание нескольких экземпляров того же типа. [55]

К примеру, требуется организовать связь между жителями. Можно связать всех жителей между собой, протянув между ними кабели телефонных линий. В таком случае будет затратно добавить еще одного жителя, так как

потребуется протянуть по еще одной линии к каждому жителю. Чтобы этого избежать, создаётся телефонная станция, которая и будет одиночкой. Если кому-то потребуется связаться с кем-то, он может это сделать через телефонную станцию. То есть для добавления нового жителя нужно будет дополнить записи на самой телефонной станции. Одиночка всегда один.

6.2.2.2 Реестр

Данный паттерн используется для хранения записей, которые в него помещают, и возвращения этих записей по имени, если они потребуются. Телефонная станция из предыдущего примера является реестром по отношению к телефонным номерам жителей.

Например, бухгалтерия является одиночкой, так как она всегда одна и помнит, что с ней происходило с момента ее начала работы. Фирма не создает каждый раз новую бухгалтерию, когда ей требуется выдать зарплату. Бухгалтерия является и реестром, так как она содержит записи о каждом работнике фирмы.

Реестр может являться одиночкой, но не всегда. Например, можно завести в бухгалтерии два журнала, в одном будут работники от А до М, в другом – от Н до Я. Каждый такой журнал будет реестром, но не одиночкой, потому что журналов уже 2.

6.2.2.3 Фабрика

Когда требуется получать какие-то объекты, к примеру, пакеты сока, совершенно не нужно знать, как их делают на фабрике. Следует просто сказать: «сделай пакет апельсинового сока», а фабрика возвращает вам требуемый пакет. Основное предназначение фабрики в том, что можно при необходимости изменять процесс появления пакета сока, хотя потребитель может и не знать об этом.

Как правило, одна фабрика занимается производством только одного рода продуктов. Не рекомендуется фабрику соков создавать с учетом

производства автомобильных покрышек. Паттерн фабрика часто является одиночкой.

6.2.2.4 Строитель

Основное различие этого паттерна со строителем заключается в том, что строитель внутри себя, как правило, содержит все сложные операции по созданию объекта (пакета сока). Если сказать, что требуется создать пакет сока, в ответ строитель запустит целую цепочку различных операций: создание пакета, печать на нем изображений, заправка в него сока, учет того сколько пакетов было создано и т.п. В свою очередь процессы в строителе можно легко изменить (к примеру, изменить рисунок на упаковке), однако потребителю сока этого знать не требуется, он также будет легко получать требуемый ему пакет сока по тому же запросу.

6.2.2.5 Прототип

Данный паттерн напоминает фабрику: он также служит для создания объектов, но иначе. В данном случае есть пустой пакет из-под сока, из которого надо сделать пакет с апельсиновым соком. Если сказать прототипу, чтобы он сделал апельсиновый сок, то он создаст свою копию и заполнит её соком. В данном случае пустой пакет является прототипом, и он создаст на своей основе требуемые вами объекты (пакеты сока) в зависимости от того, что требуется.

6.2.2.6 Фабричный метод

Фабричный метод – это порождающий паттерн проектирования, который определяет общий интерфейс для создания объектов в суперклассе, позволяя подклассам изменять тип создаваемых объектов. [56]

Допустим, фабрика производит пакеты с разными соками. Теоретически для каждого вида сока можно сделать свою производственную линию – но это не эффективно. Практичнее сделать одну линию по производству пакетов, а разделение ввести только на этапе заливки сока.

Для этого создаётся основной отдел по производству пакетов, после чего предупреждаются все подотделы, что они должны производить нужный пакет с соком по простому слову «Хочу!» (т.е. каждый подотдел должен реализовать паттерн под названием фабричный метод). Поэтому каждый подотдел заведует только своим типом сока и реагирует на слово «Хочу!».

Если потребуется пакет апельсинового сока, то следует сказать отделу по производству апельсинового сока: «Хочу!», а он в свою очередь скажет основному отделу по созданию пакетов сока: «Сделай обычный пакет и залей в него этот сок».

Фабричный метод является основой для фабрики, строителя и прототипа.

6.2.2.7 Отложенная инициализация

Иногда требуется что-то иметь под рукой, на всякий случай, но не всегда хочется прилагать каждый раз усилия, чтобы это каждый раз получать/создавать. Для таких случаев используется отложенная инициализация.

Допустим, работник из бухгалтерии должен подготовить отчет о выплатах всех сотрудников. Он может в начале каждого месяца составлять эти отчеты, но некоторые отчеты могут не понадобиться, и тогда, скорее всего, он применит отложенную инициализацию, то есть будет подготавливать этот отчет только тогда, когда он будет запрошен начальством (вышестоящим объектом).

Данный паттерн необходим для оптимизации ресурсов.

6.2.2.8 Внедрение зависимости

Внедрение зависимостей – это стиль настройки объекта, при котором поля объекта задаются внешней сущностью. Другими словами, объекты настраиваются внешними объектами. [57]

Внедрение зависимости позволяет переложить часть ответственности за какой-то функционал на отдельные объекты. Например, если требуется нанять новый персонал, то можно не создавать новый отдел кадров, а внедрить

зависимость от компании по подбору персонала, которая в свою очередь по первому требованию предоставит данные услуги. Внедрение зависимости позволяет перекладывать ответственность отдельных частей компании на внешний объект без потери общей функциональности.

6.2.3 Структурирующие паттерны

Данные паттерны помогают внести порядок и научить разные объекты более правильно взаимодействовать друг с другом. [52]

6.2.3.1 Адаптер

Чтобы заставить работать советскую вилку через евро-розетку требуется переходник. Именно это и делает адаптер – он служит промежуточным объектом между двумя другими, которые не могут работать напрямую друг с другом.

6.2.3.2 Мост

Представим ситуацию, когда требуется работать на разных автомобилях, однако садясь в новый автомобиль желательно знать, как им управлять. Таким образом можно столкнуться с паттерном под названием мост. С одной стороны, есть множество различных автомобилей, но среди них есть общий интерфейс в виде руля, педалей, коробки передач и так далее. Таким образом можно задать правила изготовления автомобилей, по которым можно создавать любые их виды, но за счет сохранения общих правил взаимодействия с ними, можно одинаково управлять каждым из них. Мостом в данном случае является пара двух объектов: конкретного автомобиля и правил взаимодействия с этим (и любым другим) автомобилем.

6.2.3.3 Компоновщик

Компоновщик объединяет группы объектов в древовидную структуру по принципу «часть-целое и позволяет клиенту одинаково работать как с отдельными объектами, так и с группой объектов». [58]

Реализацию паттерна можно представить в виде меню, которое имеет различные пункты. Эти пункты могут содержать подменю, в которых, в свою очередь, также имеются пункты. То есть пункт меню служит с одной стороны частью меню, а с другой стороны еще одним меню. В итоге однообразно можно работать как с пунктом меню, так и со всем меню в целом.

6.2.3.4 Декоратор

Декоратор представляет структурный шаблон проектирования, который позволяет динамически подключать к объекту дополнительную функциональность.

Для определения нового функционала в классах нередко используется наследование. Декораторы же предоставляет наследованию более гибкую альтернативу, поскольку позволяют динамически в процессе выполнения определять новые возможности у объектов. [59]

6.2.3.5 Фасад

Фасад используется для того, чтобы делать сложные вещи простыми. Возьмем для примера автомобиль. Представим, что управление автомобилем происходит немного по-другому: следует нажать одну кнопку, чтобы подать питание с аккумулятора, следует нажать другую, чтобы подать питание на инжектор, следует нажать третью, чтобы включить генератор, и так далее. Всё это было бы очень сложно. Для этого сложные наборы действий заменяются на более простые и комплексные, такие как «повернуть ключ зажигания». В данном случае поворот ключа зажигания и будет фасадом для всего множества внутренних действий автомобиля.

6.2.3.6 Легковес

Легковес — это структурный паттерн проектирования, который позволяет вместить большее количество объектов в отведённую оперативную память. Легковес экономит память, разделяя общее состояние объектов между собой, вместо хранения одинаковых данных в каждом объекте. [60]

В качестве стандартного применения данного паттерна можно привести следующий пример. Текст состоит из отдельных символов. Каждый символ может встречаться на одной странице текста много раз. Однако в компьютерной программе было бы слишком накладно выделять память для каждого отдельного символа в тексте. Гораздо проще было бы определить полный набор символов, например, в виде таблицы из 128 знаков (алфавитно-цифровые символы в разных регистрах, знаки препинания и т.д.), а в тексте применить этот набор общих разделяемых символов вместо сотен и тысяч объектов, которые могли бы использоваться в тексте. И как следствие подобного подхода будет уменьшение количества используемых объектов и уменьшение используемой памяти.

6.2.3.7 Прокси

Прокси предоставляет объект-заместитель, который управляет доступом к другому объекту. То есть создается объект-суррогат, который может выступать в роли другого объекта и замещать его. [61] Данный паттерн позволяет создавать какие-либо специальные механизмы доступа к объекту, что чаще всего направлено именно на улучшение производительности отдельных частей программы.

В реальной жизни можно привести следующий пример: сотрудникам одного из подразделений фирмы регулярно требуется получать информацию о том, какого числа бухгалтерия планирует выплатить зарплату. С одной стороны, каждый из них может индивидуально и регулярно ездить в бухгалтерию для выяснения этого вопроса. С другой стороны, при приближении планируемой даты подразделение может выбрать одного человека, который будет выяснять эту информацию у бухгалтерии, а в последствии уже все в подразделении могут выяснить эту информацию у него, что значительно быстрее. Вот именно этот человек и будет реализовывать прокси, предоставляющий специальный механизм доступа к информации из бухгалтерии.

6.2.4 Паттерны поведения

Эта группа паттернов позволяет структурировать подходы к обработке поведения и взаимодействия объектов. Проще говоря, как должны проходить процессы, в которых существует несколько вариантов течения событий. [52]

6.2.4.1 Цепочка обязанностей

Цепочка обязанностей – поведенческий шаблон проектирования.

Он позволяет избежать жесткой привязки отправителя запроса к получателю, позволяя нескольким объектам обработать запрос. Все возможные обработчики запроса образуют цепочку, а сам запрос перемещается по этой цепочке, пока один из ее объектов не обработает запрос. Каждый объект при получении запроса выбирает, либо обработать запрос, либо передать выполнение запроса следующему по цепочке. [62]

Самым простым примером цепочки обязанностей можно считать получение какого-либо официального документа. Например, требуется получить справку со счета из банка. Так или иначе, следует получить справку, однако, кто именно ее должен дать – пока не ясно. При посещении отделения банка могут сказать: «Мы сейчас заняты, идите в другое отделение», из-за чего требуется зайти в другое отделение банка и там получить справку.

Паттерн реализует цепочку обязанностей, отдельные объекты которой должны обработать ваш запрос. Ваш запрос может быть обработан в первом же отделении, а может и в следующем, если первый недоступен.

6.2.4.2 Команда

Команда — это поведенческий паттерн проектирования, который превращает запросы в объекты, позволяя передавать их как аргументы при вызове методов, ставить запросы в очередь, а также поддерживать отмену операций.

6.2.4.3 Интерпретатор

Сравнить данный паттерн можно с тем, как закладываются часто используемые действия в сокращенный набор слов, чтобы сам интерпретатор потом превратил этот набор в более комплексные осмысленные действия.

Допустим, что из дома вышел член семьи, которому сказали: «Литр молока, половинку белого, 200 грамм творога». По сути ничего особенного не сказали, лишь перечислили набор продуктов, однако велик шанс того, что «интерпретатор» транслирует это в команду «зайди по дороге в продуктовый магазин, купи следующее литр молока, половинку белого, 200 грамм творога и принеси это домой». Паттерн «интерпретатор» призван сократить часто исполняемые действия в более короткое их описание.

6.2.4.4 Итератор

Итератор — это поведенческий паттерн проектирования, который даёт возможность последовательно обходить элементы составных объектов, не раскрывая их внутреннего представления.

Ссылки, которые есть на многих сайтах для переходов по страницам, вроде «следующая», «предыдущая», «в начало» и т.п. по своей сути также являются доступом к «итератору», который отвечает за страницы сайта.

6.2.4.5 Посредник

Посредник — это поведенческий паттерн проектирования, который позволяет уменьшить связанность множества классов между собой, благодаря перемещению этих связей в один класс-посредник.

Если взять пример с армейским строем, то посредником будет командир отделения: ему нет необходимости взаимодействовать с каждым солдатом в отдельности, достаточно отдавать приказания лишь командиру отделения, а он уже сам решит какие действия должны быть выполнены внутри его отделения.

6.2.4.6 Хранитель

Хранитель позволяет выносить внутреннее состояние объекта за его пределы для последующего возможного восстановления объекта без нарушения принципа инкапсуляции. [63]

Допустим, если попросить друга с сотовым телефоном на время запомнить (записать себе) тот номер, что диктуют по телефону, то в этот момент друг был хранителем. Он служит для тех случаев, когда какому-либо объекту требуется сохранить своё состояние (состояние знания номера) в другом объекте (друге), и при необходимости его потом восстановить (спросить у друга номера и тем самым восстановить состояние).

6.2.4.7 Наблюдатель

Наблюдатель — это поведенческий паттерн проектирования, который создаёт механизм подписки, позволяющий одним объектам следить и реагировать на события, происходящие в других объектах.

Например, если подписаться на какую-либо электронную рассылку, то почта станет наблюдателем. Как только осуществилось некоторое событие (например, вышла новая статья или сообщение), всем, кто подписан на это событие (наблюдателям), будет выслано уведомление.

6.2.4.8 Состояние

Состояние — это поведенческий паттерн проектирования, который позволяет объектам менять поведение в зависимости от своего состояния. Извне создаётся впечатление, что изменился класс объекта.

6.2.4.9 Стратегия

Стратегия — это поведенческий паттерн проектирования, который определяет семейство схожих алгоритмов и помещает каждый из них в собственный класс, после чего алгоритмы можно взаимозаменять прямо во время исполнения программы.

Человек, который будет реализовывать стратегию в плане выдачи водительских прав, будет действовать следующим образом. Ему говорят: «Хочу права, денег мало», в ответ получают права через длительное время и с большой тратой ресурсов. Если же ему сказать: «Хочу права, денег много», то права получаются очень быстро. Что именно делал этот человек неизвестно, но от заданных условий он сам выбирает для себя стратегию.

6.2.4.10 Спецификация

Спецификация позволяет описывать, подходит ли данный объект на основе каких-либо критериев. Например, существуют несколько контейнеров для погрузки на судно. Однако, чтобы определить грузить контейнер или нет на определенное судно, нужно выбрать метод, как это определять. Реализация такого метода и является паттерном спецификация. Для каждого контейнера определяется спецификация, совпадает ли страна назначения корабля со страной назначения контейнера.

6.2.4.11 Посетитель

Посетитель — это поведенческий паттерн проектирования, который позволяет добавлять в программу новые операции, не изменяя классы объектов, над которыми эти операции могут выполняться.

Данный паттерн можно сравнить с прохождением обследования в больнице. Однако «посетителем» в терминах паттернов будут сами врачи. К примеру, есть больной, которого требуется обследовать и полечить, но так как за разные обследования отвечают разные врачи, то можно присылать к больному врачей в качестве посетителей. Врач (посетитель) приходит, обследует и делает всё необходимое. Таким образом, следуя простым правилам, можно использовать врачей для разных больных по одним и тем же алгоритмам. Как уже было сказано, паттерном Посетитель в данном случае является врач, который может одинаково обслуживать разные объекты (больных), если его позовут.

6.2.5 Заключение

Все описанные паттерны имеют очень много общего с реальной жизнью и позволяют делать код насколько же простым для чтения и понимания, как и то, что мы видим в реальной жизни.

6.3 Анти-паттерны проектирования

Если паттерны проектирования являются примерами хорошего программирования, то анти-паттерны – это шаблоны ошибок, которые совершаются при решении различных задач. Частью практик хорошего программирования является именно избежание анти-паттернов. [54]

Рассмотрим несколько распространённых анти-паттернов в программировании.

6.3.1 Программирование Копированием-И-Вставкой

Когда от программиста требуется написание двух схожих функций, самым простым решением является написание одной функции, её копирование и внесение некоторых изменений в копию. Из-за этого могут возникнуть следующие проблемы:

- ухудшается переносимость кода – если потребуется подобный функционал в другом проекте, то следует найти места с одним и тем же кодом и переносить их по отдельности;
- понижается качество кода – часто программист забывает вносить требуемые изменения в копии одного и того же кода;
- усложняется поддержка кода – если в изначальном варианте был баг, то этот баг попал во все копии этого кода;
- усложняется проверка кода на ошибки.

Решение данной проблемы простое – программисту следует создавать общие решения.

6.3.2 Спагетти-код

Спагетти-код – избыточной и запутанный код. Такой код так же очень часто содержит в себе множество примеров программирования Копированием-И-Вставкой.

Подобный код не сможет разобрать даже сам автор, если через некоторое время вернётся к нему.

В объектно-ориентированном программировании спагетти-код может представлять объекты с огромными методами. Причинами такого подхода являются разработка по принципу «Коли работает, лучше не трогать», малоэффективная проверка кода на баги, недостаток опыта в ООП разработке.

Использовать спагетти-код повторно невозможно и нежелательно.

6.3.3 Золотой молоток

Золотой молоток – уверенность в полной универсальности решения. На практике это проявляется в применении некоего одного решения для всех задач. Причиной такого подхода является лень к изучению нового – программист-новичок пытается решить все задачи единственным методом, который он понял.

Как правило, для каждой задачи имеется не одно, а несколько оптимальных решений – именно к поиску их и сводится разработка.

6.3.4 Магические числа

Магическое число – константа, использованная в коде для какой-либо цели, в то время как это число не имеет смысла без соответствующего комментария.

Программист, не являющийся автором такого кода, с трудом сможет понять, как это работает. Числа затрудняют понимание кода и его рефакторинг.

6.3.5 Жёсткое кодирование

Жёсткое кодирование – внедрение различных данных об окружении в реализацию. Главная опасность – непереносимость.

6.3.6 Мягкое кодирование

Мягкое кодирование приводит к тому, что программист настраивает всё что можно, из-за чего конфигурация такой системы становится слишком сложной для понимания и настройки.

Перед началом разработки следует определить, что должно можно настраивать, а что может быть настроено автоматически или быть постоянным.

6.3.7 Ненужная сложность

Ненужная сложность может быть внесена в решение любой задачи. Это могут быть необязательные проверки, неоптимизированный код, код с мягким кодированием и т.п. Данный подход ухудшает понимание кода. Причиной является некомпетентность программиста.

Для решения этой проблемы стоит проводить тщательные проверки кода на ошибки и осуществлять эффективный рефакторинг.

6.3.8 Лодочный якорь

Под этим паттерном подразумевается сохранение каких-то частей системы, которые не используются и остались после рефакторинга. Для решения этой проблемы следует осуществлять рефакторинг и заранее продумывать архитектуру системы.

6.3.9 Изобретение велосипеда

Подразумевается, что программист разрабатывает собственное решение для конкретной задачи, для которой уже существуют гораздо лучшие решения, чем придуманное программистом. Данный подход приводит к потере времени, понижению эффективности работы программиста и неоптимальным разработанным решением.

6.3.10 Поток лавы

Может произойти ситуация, в которой некий код очень давно не менялся. Он может быть не документирован или иметь комментарий вида "Вроде работает, хотя неизвестно почему, лучше не трогать". Решается рефакторингом.

6.3.11 Программирование перебором

Многие программисты-новички решают некоторые задачи методом перебора (они подбирают необходимые параметры, изменяют порядок вызова функций и т.п.). Все эти попытки заставить код работать устраняют только симптомы проблемы, но не дают понимания сути происходящего. А если программист не может понять то, как работает код, то он не сможет предусмотреть все варианты развития событий и обязательно о чём-то забудет.

6.3.12 Слепая вера

Этот анти-паттерн характеризуется недостаточной проверкой корректности входных данных, недостаточным исправлением ошибок. Очень часто программист думает, что его код всегда будет в идеальных условиях, никогда не выдаст ошибки и не получит неверных входных данных или данных неверного типа. Следует помнить про проверку входных данных и возможные проблемы у чужого кода, который используется в проекте.

6.3.13 Божественный объект

Божественный объект является анти-паттерном, который довольно часто встречается у разработчиков, связанных с объектно-ориентированными языками. Объект в данном случае используется для решения широкого класса всевозможных задач. В итоге получается код, трудный для понимания.

Решением данной проблемы является разбиение задач на подзадачи, с возможностью решения этих подзадач различными классами, которые выполняют чётко одну определённую задачу и больше ничего.

6.3.14 Выводы

Анти-паттерны надо не просто знать, надо знать их причины и методы борьбы, чтобы в будущем с не сталкиваться со следующими проблемами:

- сложность рефакторинга кода;
- избыточность кода;

- сложность проверки кода на ошибки;
- непредвиденные баги;
- траты лишнего времени на исправление ошибок.

6.4 Разработка движка симулятора профессии

6.4.1 Внутренний и внешний интерфейсы

В программировании разделяют методы и свойства объекта на две группы: [69]

- внутренний интерфейс – это свойства и методы, доступ к которым может быть осуществлен только из других методов объекта, их также называют «приватными»;
- внешний интерфейс – это свойства и методы, доступные снаружи объекта, их называют «публичными».

Приватные функции доступны для вызова только в исходном классе. Они будут использованы для реализации всех требований заказчика: реализация ORM, определение следующих событий, изменение данных в базе данных и т.п.

Публичные функции будут доступны для вызова как в самом классе, так и вне класса. Данные функции и будут реализовывать API-интерфейс, которым будет пользоваться программист.

Этим разделением обязанностей мы реализуем инкапсуляцию – то есть отделяем внутренний интерфейс от внешнего.

Если провести аналогию с кофеваркой – то, что спрятано внутри кофеварки: трубка кипятильника, нагревательный элемент, тепловой предохранитель и так далее – это её внутренний интерфейс. Внутренний интерфейс используется для обеспечения работоспособности объекта, его детали используют друг друга. Например, трубка кипятильника подключена к нагревательному элементу.

Но снаружи кофеварка закрыта специальным кожухом, чтобы никто к ним не подобрался. Детали скрыты и недоступны. Виден лишь внешний

интерфейс. Получив объект, всё, что нужно для пользования им – это знать внешний интерфейс. О внутреннем же знать вообще не обязательно.

6.4.2 Модификаторы доступа

Все поля, свойства, методы, классы в языке C# имеют модификаторы доступа. Модификаторы доступа – это ключевые слова, которые задают объявленный уровень доступности члена или типа. Модификаторы доступа позволяют задать допустимую область видимости для членов класса. То есть контекст, в котором можно употреблять данную переменную или метод.

Они позволяют реализовывать инкапсуляцию, то есть благодаря им программист видит только те поля, свойства и методы, которые предоставляют ему внешний интерфейс. Программисту не нужно знать внутренний интерфейс. Другие члены остаются для него скрыты и прямого доступа он к ним не имеет, так как они для него не нужны – они нужны только для работы самого движка.

Из всех модификаторов доступа можно выделить два модификатора, которые будут использованы при разработке движка:

- public – общедоступный класс или член класса. Поля и методы, объявленные с модификатором public, видны другим классам из текущего пакета и из внешних пакетов.
- private – закрытый класс или член класса, противоположность модификатору public. Закрытый класс или член класса доступен только из кода в том же классе или пространстве имён.

6.4.3 Используемые паттерны и принципы рефакторинга

При разработке движка на языке C# мы будем пользоваться следующими тезисами.

1. Чтобы избежать изобретения велосипеда, то есть такой ситуации, когда программист разрабатывает своё решение с нуля, в то время как есть более удачные и работоспособные решения от других разработчиков, будут

использоваться сторонние решения для доступа к БД и преобразованию реляционных таблиц в классы, а точнее СУБД SQLite для манипуляции данными в базе данных и Entity Framework для связи между двумя несоответствующими понятиями: реляционные таблицы, основанные на сущностях и связях, и классы, основанные на поведении и свойствах.

2. Может возникнуть ситуация, когда один класс отвечает за все возможные функции, что влечёт за собой непереносимый код, в котором сложно разобраться. Так же, подобный код довольно сложно поддерживать, учитывая, что вся система зависит практически только от него. Такой объект называется божественным объектом. Также при создании такого класса может возникнуть так называемый спагетти-код, из-за сложности которого даже сам автор не сможет в нём разобраться. Чтобы избежать подобного божественного объекта, мы распределим обязанности между пятью классами:

- класс, отвечающий за ORM;
- класс, отвечающий за подключение к БД посредством SQLite;
- класс, отвечающий за API, предоставляемый программисту;
- класс, состоящий из определений сущностей;
- класс, состоящий из определения исключений.

3. Так как идёт работа с базой данных, следует каким-то образом произвести взаимодействие с ней, чтобы получать выборки данных из реляционных таблиц. Хорошим тоном будет воспользоваться паттерном проектирования под названием Одиночка. Используя данный паттерн, мы сможем избежать анти-паттернов Программирование Копированием-И-Вставкой и Спагетти-код. К тому же в дальнейших программах мы можем использовать уже готовый класс для получения данных из БД с помощью SQLite. Одиночка всегда один.

4. Реляционная модель акцентирует свое внимание на структуре и связях сущностей, объектная - на их свойствах и поведении. Из-за этого возникает парадигма несоответствия, то есть возникает проблема преобразования

реляционных таблиц в обычные классы и наоборот. Данную проблему может решить Entity Framework, который дает возможность представлять таблицы в виде обычных классов и соответственно обращаться с ними как с простыми классами, не применяя SQL. Применяется паттерн проектирования под названием адаптер.

5. Преобразование реляционных таблиц в обычные классы следует сделать в виде отдельного объекта, использовав паттерн проектирования под названием Одиночка. Одиночка всегда один.

6. Для каждой функции API не следует создавать отдельный класс, так как это приведёт к избыточности кода и применению анти-паттернов Программирование Копированием-И-Вставкой и Спагетти-код. Хорошим тоном будет создать один объект движка, предоставляющий функции API. Класс, отвечающий за логику работы движка симулятора профессии, следует разработать, используя паттерн проектирования под названием Одиночка. Объект движка должен быть один, он и будет реализовывать API, которым будет пользоваться программист при разработке своей игры. Других объектов этого класса не создаётся, так как уже есть единственный объект, который помнит, что с ним происходило с момента начала работы. Одиночка всегда один.

7. Для выборки данных из базы данных будет использоваться ORM, которое преобразует реляционные таблицы в обычные классы. Подобных запросов много, но их объединяет одна характеристика: они используют одни и те же методы, но с разными параметрами. Для того, чтобы избежать реализации однотипных функций, из-за которых возникнет проблема анти-паттернов Программирование Копированием-И-Вставкой и Спагетти-код, необходимо воспользоваться паттерном Строитель. Данный паттерн позволяет выполнять сложные операции пошагово. Строитель даёт возможность использовать один и тот же код строительства для получения разных представлений объектов. Строитель реализует в себе универсальную функцию, которая может выполнить определённую задачу согласно поставленным параметрам.

8. Такие функции как «получить локацию по id», «получить событие по id», «получить все доступные события на локации» и т.п. используют схожие принципы по получению выборки из таблиц. Если реализовывать для них отдельных функции, код будет постоянно расти и расти, возникнет проблема использования анти-паттерна программирования Копированием-И-Вставкой. Для того, чтобы избежать этого, мы воспользовались паттерном под названием Строитель, который обычно идёт вместе с паттерном под названием Фабричный метод. Паттерн под названием Фабричный метод использует в своей основе вызов паттерна Строитель с нужными аргументами. Данный подход уменьшает количество кода, исключаются предпосылки возникновения программирования Копированием-И-Вставкой и связанных с ним проблем: избыточность, сложность проверки кода на ошибки, сложность рефакторинга.

9. При написании кода будет реализован паттерн проектирования под названием Отложенная инициализация. Его суть в том, что какие-либо необходимые операции выполняются непосредственно перед тем, как будет необходим результат этих операций. Данный паттерн служит для оптимизации ресурсов.

10. При подключении к базе данных происходит подключение к базе данных посредством СУБД SQLite. Применяется паттерн проектирования под названием внедрение зависимостей, так как за манипуляцию с данными в базе данных отвечает не сама программа, а внешний объект в виде СУБД SQLite.

11. При написании кода мы воспользуемся такой особенностью C# как перегрузка функций. Под перегрузкой функций подразумевается такая ситуация: мы имеем несколько функций с одинаковым именем, но с различными параметрами. Параметры этих функций могут отличаться порядком следования, типом, количеством. Благодаря такого подходу, мы можем избежать дублирования имён функций, которые выполняют схожие действия, но реализованный по-разному.

12. Для выбора различных путей получения результата будем использовать паттерн проектирования под названием Стратегия, который в зависимости от определённых параметров выберет соответствующее поведение. Инициализация объекта движка может пойти по трём различным путям:

- открыть уже имеющуюся базу данных по указанному пути, если не получится, вернуть ошибку о неправильном пути;
- создать базу данных с необходимыми таблицами по указанному пути, если не получится, вернуть ошибку о неправильном пути;
- предоставить пользователю выбор строки подключения к базе данных посредством СУБД SQLite.

13. Для выбора различных путей получения результата будет использоваться паттерн проектирования под названием Стратегия, который в зависимости от Состояния выберет соответствующее поведение. В зависимости от Состояния персонажа (жив, проиграл, победил) при попытке получить доступные локации, события и действия, может возникнуть следующее:

- программист получит нужные сущности, если игрок жив;
- программист получит исключение, так как персонаж уже окончил игру (проиграл или победил).

14. Персонаж может иметь следующие состояния:

- Персонаж достиг своей цели – игра окончена;
- Персонаж проиграл – игра окончена;
- Персонаж не проиграл и не достиг своей цели – игра продолжается.

В первом и втором случае персонажу уже не будут нужны доступные события и действия, а это значит, что функции могут их не возвращать. То есть в зависимости от состояния персонажа соответствующие функции могут и не возвращать доступные действия и события, говоря, что игра уже окончена. В данном случае применяется паттерн под названием Событие, так как движок в зависимости от состояния персонажа может менять свою работу.

15. При работе с данными игры программист не использует SQL-запросы, они скрыты от него благодаря внедрению перечисляемого типа, который характеризует соответствующую таблицу в базе данных. Данный подход инкапсулирует архитектуру движка, предоставляя программиста более удобное обращение с базой данных посредством перечисляемых объектов. Таким образом реализует паттерн проектирования под названием команда. Объект команды заключает в себе само действие.

16. Возвращение кодов ошибок — давно устаревшая практика процедурного программирования. В современном программировании для обработки ошибок используются специальные классы, называемые исключениями. При возникновении проблемы выбрасывается исключение и оно впоследствии ловится одним из обработчиков исключений. При этом запускается специальный код обработки внештатной ситуации, который игнорируется в обычных условиях. Данный подход избавляет код от множества условных операторов проверки кодов ошибок. Обработчики исключений намного чётче разграничивают нормальный и нештатный путь исполнения программы. Но не стоит этим злоупотреблять, где хватает обычной проверки условия.

6.4.4 Архитектура движка

При проектировании архитектуры движка будут использоваться следующие книги:

1. Тепляков, С.И. Паттерны проектирования на платформе NET; [52]
2. Швец, А. Погружение в Рефакторинг; [70]
3. Швец, А. Погружение в Паттерны проектирования. [71]

6.4.4.1 Диаграмма классов движка

На рисунах 6.4.4.1.1, 6.4.4.1.2, 6.4.4.1.3 представлена архитектура движка в виде диаграммы классов.

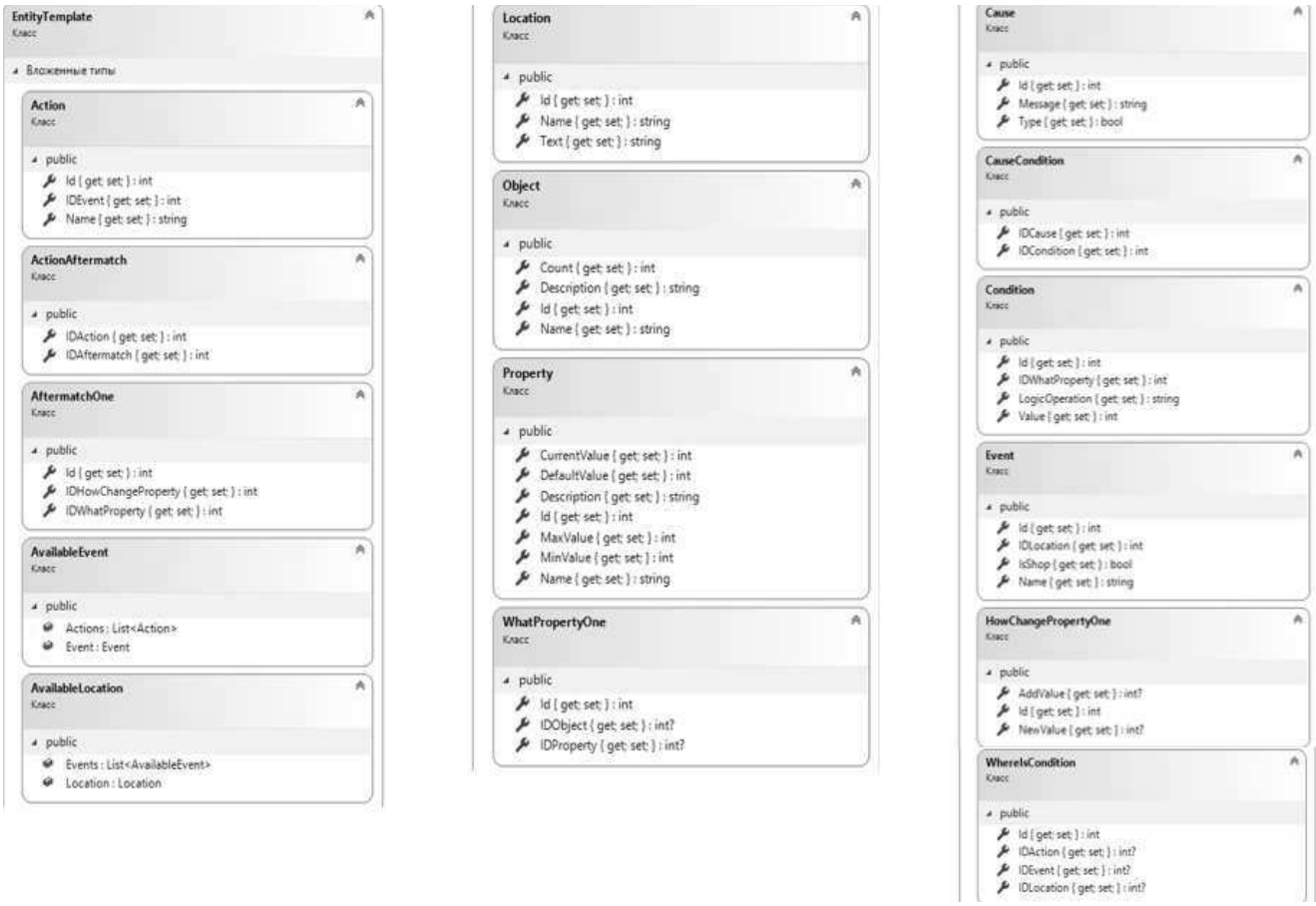

Рисунок 6.4.4.1.1 – Класс EntityFramework

Рисунок 6.4.4.1.2 – Класс Exception и перечисление EntityName

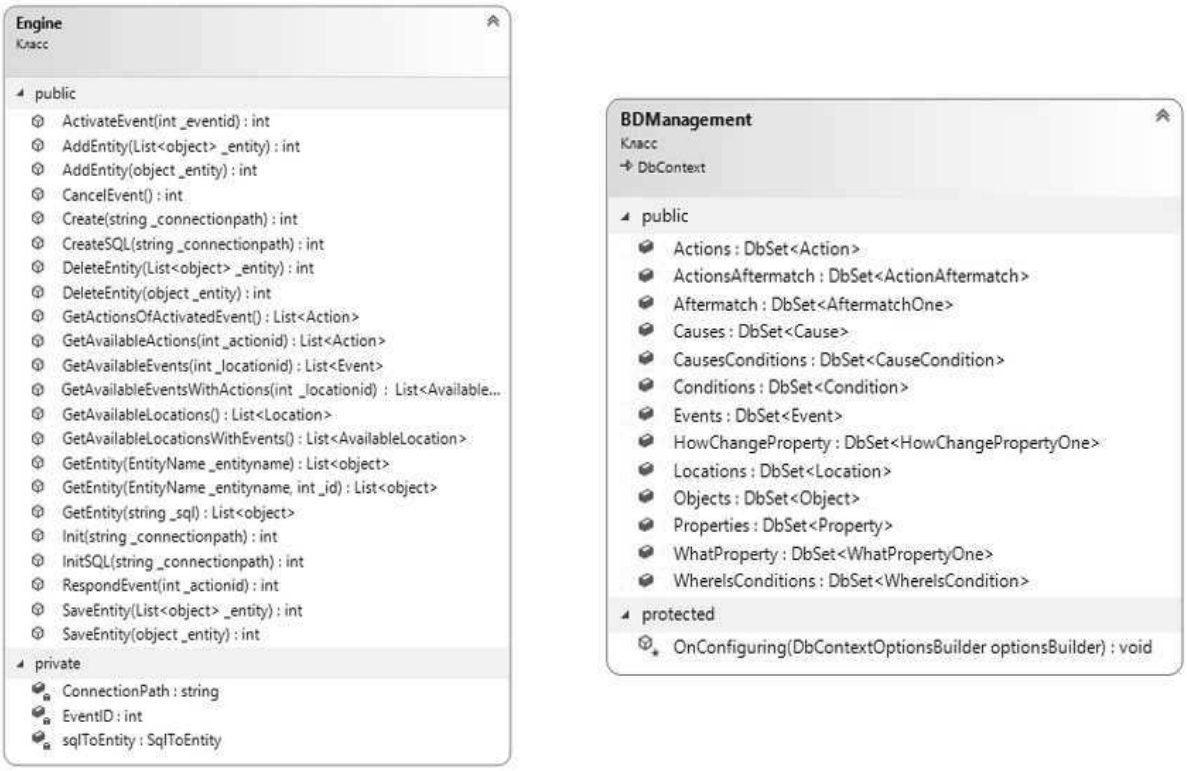

Рисунок 6.4.4.1.3 – Классы Engine и BDManagement

6.4.4.2 Пространство имён

Язык C# имеет одну особенность под названием пространство имён. Чтобы не было путаницы с именами типов и чтобы можно было бы их хоть как-то структурировать, было введено такое понятие, как namespace или пространство имён. Все определяемые классы не существуют сами по себе, а, как правило, заключаются в специальные контейнеры - пространства имен. Данный подход обеспечивает структуризацию библиотек классов.

Пространство имён системы «Симулятор профессии» называется SimProfEngine.

Это пространство имён состоит из:

- самым главным классом в системе «Симулятор профессии» будет являться публичный класс Engine, предоставляющий функции API и который будет содержать в себе приватный объект: объект класса SqlToEntity для обеспечения зависимости с БД посредством СУБД SQLite и EntityFramework;

- приватный класс BDManagement, который нужен для обеспечения связи с БД и который будет виден только внутри этого пространства, вне этого пространства получить прямой доступ к нему не получится;
- приватный класс SqlToEntity, который будет посредником между реляционными таблицами и обычными классами и который будет виден только внутри этого пространства, вне этого пространства получить прямой доступ к нему не получится;
- private enum EntityName {Actions, Aftermatch, Causes, Conditions, Events, HowChangeProperty, Locations, Objects, Properties, WhatProperty, WhereIsConditions, ActionsAftermatch, CausesConditions, Error} – перечисляемый тип, который будет использован в фабричном методе в классе SqlToEntity и который нужен для того, чтобы программист не работал с SQL-запросами, а передавал этот простой объект перечисляемого типа в фабричную функцию для получения нужной таблицы;
- публичный класс EntityTemplate, состоящий из классов, представляющих собой набор свойств, типы которых соответствуют типам столбцов из соответствующих таблиц.

6.4.4.3 Класс EntityTemplate

Класс EntityTemplate имеет следующие классы:

- public class Action – класс, который предоставляет собой набор свойств, типы которых соответствуют типам столбцов из таблицы Actions;
- public class AftermatchOne – класс, который предоставляет собой набор свойств, типы которых соответствуют типам столбцов из таблицы Aftermatch;
- public class Cause – класс, который предоставляет собой набор свойств, типы которых соответствуют типам столбцов из таблицы Causes;
- public class Condition – класс, который предоставляет собой набор свойств, типы которых соответствуют типам столбцов из таблицы Conditions;

- public class Event – класс, который предоставляет собой набор свойств, типы которых соответствуют типам столбцов из таблицы Events;
- public class HowChangePropertyOne – класс, который предоставляет собой набор свойств, типы которых соответствуют типам столбцов из таблицы HowChangeProperty;
- public class Location – класс, который предоставляет собой набор свойств, типы которых соответствуют типам столбцов из таблицы Locations;
- public class Object – класс, который предоставляет собой набор свойств, типы которых соответствуют типам столбцов из таблицы Objects;
- public class Property – класс, который предоставляет собой набор свойств, типы которых соответствуют типам столбцов из таблицы Properties;
- public class WhatPropertyOne – класс, который предоставляет собой набор свойств, типы которых соответствуют типам столбцов из таблицы WhatProperty;
- public class WhereIsCondition – класс, который предоставляет собой набор свойств, типы которых соответствуют типам столбцов из таблицы WhereIsConditions;
- public class ActionAftermatch – класс, который предоставляет собой набор свойств, типы которых соответствуют типам столбцов из таблицы ActionsAftermatch;
- public class CauseCondition – класс, который предоставляет собой набор свойств, типы которых соответствуют типам столбцов из таблицы CausesConditions;
- public class AvailableLocation – вспомогательный класс, содержащий локацию и все события на ней;
- public class AvailableEvent – вспомогательный класс, содержащий событие и все действия на ней.

6.4.4.4 Класс BDManagement

Класс BDManagement разработан для обеспечения связи с БД. Он является производным от DbContext и содержит следующие классы, которые представляют собой отображение реляционных таблиц, и функции:

- public DbSet<Action> Actions – объект, который представляет сущность Actions;
- public DbSet<AftermatchOne> Aftermatch – объект, который представляет сущность Aftermatch;
- public DbSet<Cause> Causes – объект, который представляет сущность Causes;
- public DbSet<Condition> Conditions – объект, который представляет сущность Conditions;
- public DbSet<Event> Events – объект, который представляет сущность Events;
- public DbSet<HowChangePropertyOne> HowChangeProperty – объект, который представляет сущность HowChangeProperty;
- public DbSet<Location> Locations – объект, который представляет сущность Locations;
- public DbSet<Object> Objects – объект, который представляет сущность Objects;
- public DbSet<Property> Properties – объект, который представляет сущность Properties;
- public DbSet<WhatPropertyOne> WhatProperty – объект, который представляет сущность WhatProperty;
- public DbSet<WhereIsCondition> WhereIsConditions – объект, который представляет сущность WhatProperty;
- public DbSet<ActionAftermatch> ActionsAftermatch – объект, который представляет сущность ActionsAftermatch;
- public DbSet<CauseCondition> CausesConditions – объект, который представляет сущность CausesConditions;

- protected override void OnConfiguring (DbContextOptionsBuilder optionsBuilder) – данная функция переопределяет абстрактную функцию для подключения к БД, чтобы можно было работать с Entity Framework.

6.4.4.5 Класс SqlToEntity

Класс SqlToEntity представляет собой паттерн проектирования под названием «адаптер». Он служит для обеспечения связи между двумя несоответствующими понятиями: реляционные таблицы и обычные классы. Реляционная модель акцентирует свое внимание на структуре и связях сущностей, объектная - на их свойствах и поведении. Данный класс содержит следующие свойства и функции:

- private string ConnectionString – переменная, которая представляет собой строку подключения к базе данных;
- private List<object> GetEntityFabric(string _sql) – строитель, который возвращает сущность по заданному в параметре SQL-запросу;
- public List<object> GetEntity(EntityName _entityname) – фабричный метод, который обращается к фабрике с соответствующим SQL-запросом, который определяется параметром перечисляемого типа _entityname;
- public List<object> GetEntity(string _sql) – фабричный метод, который обращается к фабрике с соответствующим SQL-запросом, который определяется параметром _sql, представляющим собой SQL-запрос;
- public List<object> GetEntity(EntityName _entityname, int id) – фабричный метод, который обращается к фабрике с соответствующим SQL-запросом, который определяется параметром перечисляемого типа _entityname и id строки;
- private EntityName WhatIsEntity(List<object> _entity) – метод, который определяет, что за сущность хранится в параметре _entity;

- private int SaveEntity(List<object> _entity) – метод, который сохраняет измененные значения сущности; для определения сущности вызывает метод WhatIsEntity; возвращает 0, если подключение было успешным, 1 при неудаче;
- private int DeleteEntity(List<object> _entity) – метод, который удаляет данную сущность; для определения сущности вызывает метод WhatIsEntity; возвращает 0, если подключение было успешным, 1 при неудаче;
- private int AddEntity(List<object> _entity) – метод, который добавляет данную сущность; для определения сущности вызывает метод WhatIsEntity; возвращает 0, если подключение было успешным, 1 при неудаче;
- private int SaveEntity(object _entity) – метод, который сохраняет измененные значения сущности; для определения сущности вызывает метод WhatIsEntity; возвращает 0, если подключение было успешным, 1 при неудаче;
- private int DeleteEntity(object _entity) – метод, который удаляет данную сущность; для определения сущности вызывает метод WhatIsEntity; возвращает 0, если подключение было успешным, 1 при неудаче;
- private int AddEntity(object _entity) – метод, который добавляет данную сущность; для определения сущности вызывает метод WhatIsEntity; возвращает 0, если подключение было успешным, 1 при неудаче;
- public List<Location> GetAvailableLocations() – фабричный метод, который обращается к фабрике с соответствующим SQL-запросом по выборке локаций, которые доступны на данный момент, если персонаж не закончил игру;
- public List<Event> GetAvailableEvents(int _locationid) – фабричный метод, который обращается к фабрике с соответствующим SQL-запросом по выборке событий, которые доступны на данный момент на данной локации, если персонаж не закончил игру;
- public List<Action> GetAvailableActions(int _actionid) – фабричный метод, который обращается к фабрике с соответствующим SQL-запросом по выборке действий, которые доступны на данный момент у данного события, если персонаж не закончил игру.

6.4.4.6 Класс Engine

Класс Engine является основным классом, с которым будет работать программист. Только этот класс имеет публичный модификатор доступа. Состоит из следующих свойств и методов, реализующих API:

- private SqlToEntity sqlToEntity – объект, реализующий паттерн адаптер;
- private string ConnectionPath – переменная, которая представляет собой строку подключения к базе данных;
- private int EventId – переменная, которая указывает на id события, которое ждёт ответа (по умолчанию равно нулю, то есть никакое событие не ждёт ответа);
- private int ActionAvailability() – проверяет, есть ли возможность выполнять сейчас действия;

- public int Init(string _connectionpath) – инициализация соединения с БД; параметр – путь до файла; возвращает 0, если подключение было успешным, 1 при неудаче;
- public int Create(string _connectionpath) – создаём базу данных по указанному пути в соответствии со структурой системы; параметр – путь до файла; возвращает 0, если подключение было успешным, 1 при неудаче;
- public int InitSQL(string _connectionpath) – инициализация соединения с БД; параметр – строка подключения; возвращает 0, если подключение было успешным, 1 при неудаче;
- public int CreateSQL(string _connectionpath) – создаём базу данных по указанному пути в соответствии со структурой системы; параметр – строка подключения; возвращает 0, если подключение было успешным, 1 при неудаче;
- public List<object> GetEntity(EntityName _entityname) – метод, который вызывает метод GetEntity класса SqlToEntity;

- public List<object> GetEntity(EntityName _entityname, int _id) – метод, который вызывает метод GetEntity класса SqlToEntity;
- public List<object> GetEntity(string _sql) – метод, который вызывает метод GetEntity класса SqlToEntity;
- public int SaveEntity(List<object> _entity) – метод, который вызывает метод SaveEntity класса SqlToEntity; возвращает 0, если подключение было успешным, 1 при неудаче;
- public int SaveEntity(object _entity) – метод, который вызывает метод SaveEntity класса SqlToEntity; возвращает 0, если подключение было успешным, 1 при неудаче;
- public int DeleteEntity(List<object> _entity) – метод, который вызывает метод DeleteEntity класса SqlToEntity; возвращает 0, если подключение было успешным, 1 при неудаче;
- public int DeleteEntity(object _entity) – метод, который вызывает метод DeleteEntity класса SqlToEntity; возвращает 0, если подключение было успешным, 1 при неудаче;
- public int AddEntity(List<object> _entity) – метод, который вызывает метод AddEntity класса SqlToEntity; возвращает 0, если подключение было успешным, 1 при неудаче;
- public int AddEntity(object _entity) – метод, который вызывает метод AddEntity класса SqlToEntity; возвращает 0, если подключение было успешным, 1 при неудаче;
- public List<Location> GetAvailableLocations() – метод, который возвращает доступные локации, т.е. возвращает локации, которые удовлетворяют условиям;
- public List<Event> GetAvailableEvents(int _locationid) – метод, который возвращает доступные события на локации, т.е. возвращает события, которые удовлетворяют условиям;

- public List<AvailableLocation> GetAvailableLocationsWithEvents() – метод, который возвращает все доступные локации, все доступные события на этих локациях, все доступные действия у этих событий;
- public List<AvailableEvent> GetAvailableEventsWithActions(int _locationid) – метод, который возвращает все доступные события на этой локации, все доступные действия у этих событий;
- public List<Action> GetAvailableActions(int _actionid) – метод, который возвращает доступные действия события, т.е. возвращает действия, которые удовлетворяют условиям;
- public int ActivateEvent(int _eventid) – активация события; возвращает 0, если подключение было успешным, 1 при неудаче;
- public int CancelEvent() – отмена ожидания действия на событие; возвращает 0, если подключение было успешным, 1 при неудаче;
- public List<Action> GetActionsOfActivatedEvent() – этот метод возвращает доступные действия у события, которое ожидает действия;
- public int RespondEvent(int _actionid) – ответ действием на событие; возвращает 0, если подключение было успешным, 1 при неудаче.

6.4.4.7 Класс Exceptions

Класс Engine хранит описания исключений, которое могут возникнуть при работе с движком:

- класс GameOver содержит два исключения: Win и Fail, которые возникают в тех случаях, когда программист пытается получить доступные локации, события, действия, назначить событие, отменить событие, ответить на событие при таком условии, когда игра уже окончена.
- Класс InvalidArgument появляется тогда, когда программист хочет сохранить неправильную сущность в базу данных.

6.4.5 Выводы

При разработке движка мы использовали следующие:

- Анти-паттерны. Их необходимо знать, так как зная их, можно понять, что при разработке кода программист свернул не туда, что грозит ему спагетти-кодом, избыточностью, сложностью рефакторинга и проверок на ошибки.
- Паттерны проектирования. Они представляют собой некоторые шаблоны, приёмы того, как правильно сделать ту или иную вещь. Благодаря им можно спроектировать хорошую систему, если правильно ими воспользоваться.
- Принципы ООП. Если разрабатывать некоторую систему на объектно-ориентированном языке, то без соответствующих принципов не обойтись, так как они представляют основу программирования на этом языке.

Благодаря этим приёмам мы смогли спроектировать движок симулятора профессии, стараясь не наступать на ошибки, которые обычно выполняют при разработке подобных систем.

Модель автоматизированного бизнес-процесса Заказчика представлена на рисунке 6.4.5.1.

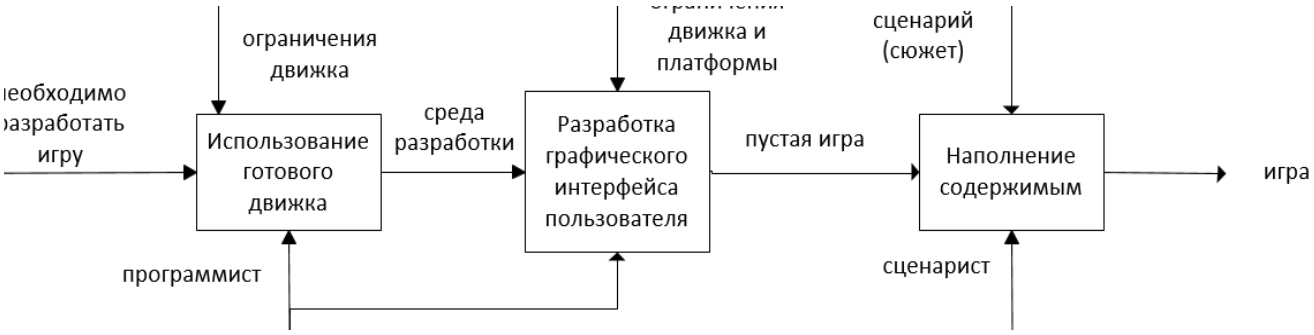

Рисунок 6.4.5.1 – Контекстная диаграмма, модель IDEF0 TO BE верхний уровень.

7 Тестирование продукта

7.1 Тестирование на платформе .Net

Для проверки движка было написано графическое приложение на платформе .Net.

База данных была наполнена следующими данными:

- локация Университет;
- предметы под названиями Студенческий, Плюшевый медведь, Ключи от дома, Пропуск в общежитие, Зачётная книжка;
- четыре события без условия;
- два действия с условием у одного события;
- одно действие без условия у одного события;
- свойства под названиями Здоровье, Известность, Деньги, Количество действий, Количество дней.

На локации Университет могут возникнуть следующие события:

- послушать лекцию;
- сдать долги;
- посидеть на чужой паре;
- отчислиться.

У события «послушать лекцию» есть следующие действия:

- действие «Написать объяснительную о том, что вы не взяли студенческий» появляется всегда, так как для него нет условия возникновения, и имеет последствие в виде уменьшения Известности на 10 пунктов;
- действие «Сказать, что вас все знают, так как вы популярны и вам не нужен студенческий», когда свойство Известность больше 20 пунктов;
- действие «Показать студенческий и идти в аудиторию» появляется всегда, когда у персонажа в предметах есть Студенческий, и имеет последствие в виде увеличения Известности на 10 пунктов.

Для получения локаций, событий и действий использовались функции public List<AvailableLocation> GetAvailableLocationsWithEvents() и public List<AvailableEvent> GetAvailableEventsWithActions(int _locationid).

Примеры работы программы, если у персонажа нет студенческого и он выбрал написать объяснительную, представлены на рисунках 7.1 и 7.2.

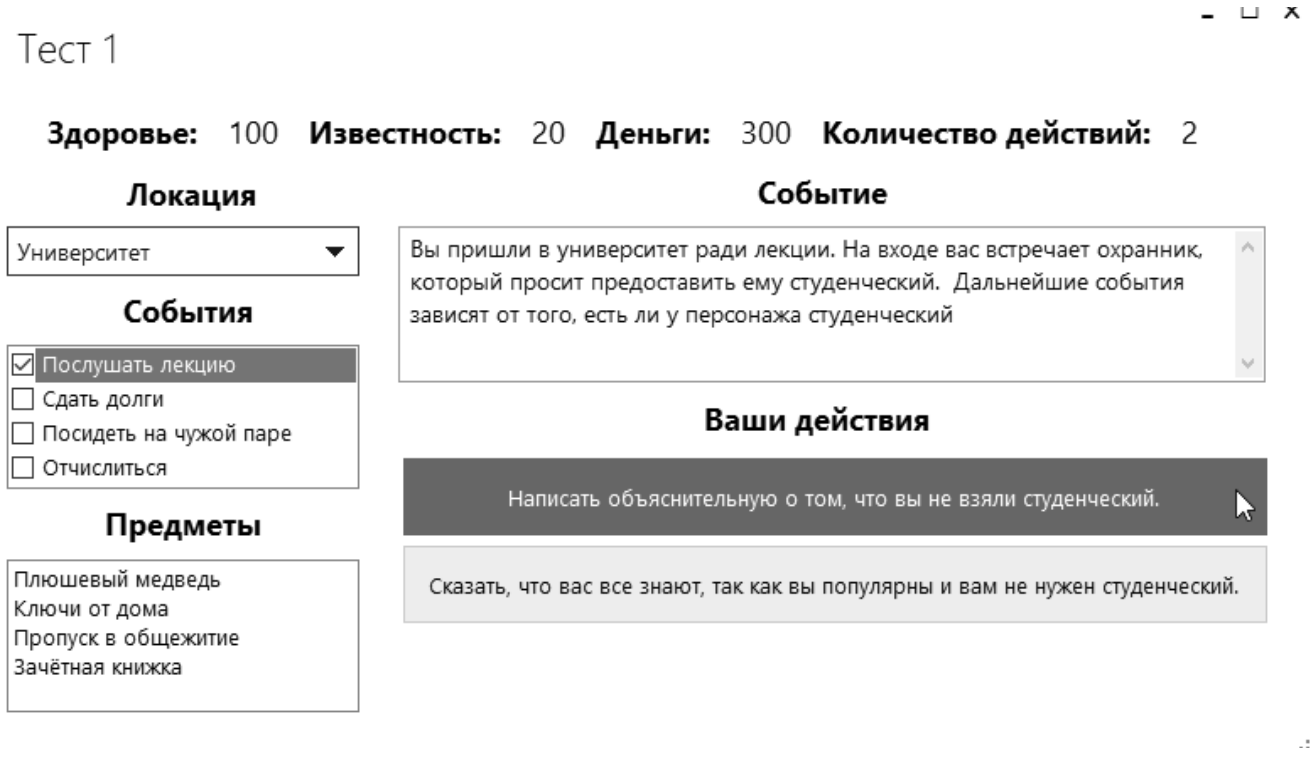

Рисунок 7.1 – Пример работы теста

Рисунок 7.2 – Пример работы теста

Примеры работы программы, если у персонажа есть студенческий и он выбрал показать студенческий охраннику, представлены на рисунках 7.3 и 7.4.

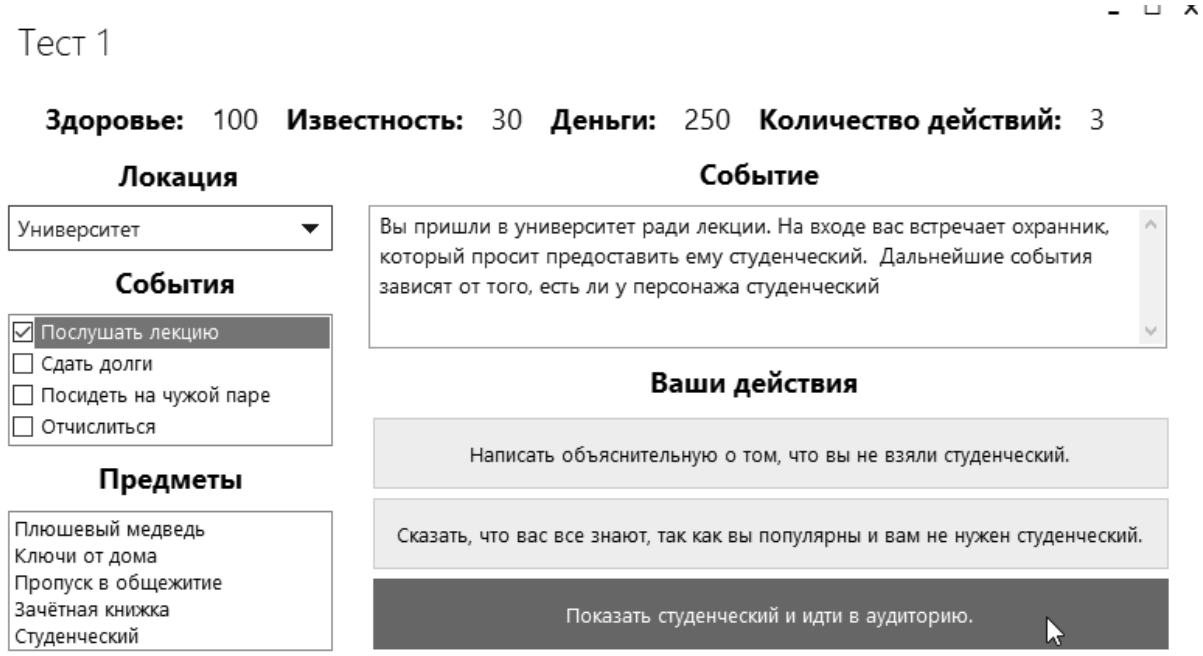

Рисунок 7.3 – Пример работы теста

Рисунок 7.4 – Пример работы теста

Таким образом был проведён тест работоспособности движка, который закончился успехом. Разработка графического приложения с уже готовым движком намного удобнее, чем написание своего движка с нуля.

8 Руководство пользователя

8.1 Алгоритмы работы с движком

При работе с API системы «Симулятор профессии» программист может двигаться в следующих направлениях:

- создание новой БД или редактирование уже существующей БД;
- взаимодействие с API для начала игры.

Первое направление нацелено на наполнение игры контентом. Алгоритм наполнения можно увидеть на рисунке 8.1.1.

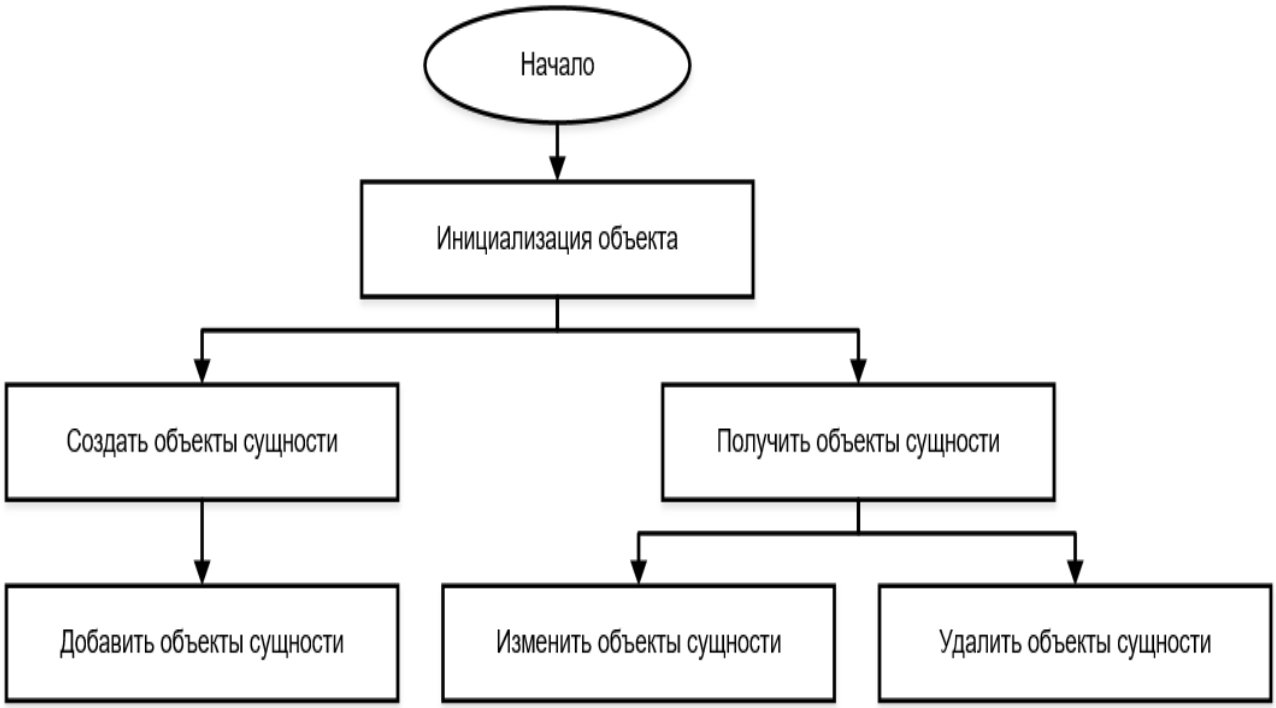

Рисунок 8.1.1 – Наполнение игры контентом

Второе направление нацелено на взаимодействия программиста и уже готового материала. Алгоритм взаимодействия можно увидеть на рисунке 8.1.2.

Количество действий автоматически уменьшается, если декремент текущего значения не даёт ноль, и устанавливается в значение по умолчанию, если декремент текущего значения даёт ноль, при выполнении действия. Также автоматически увеличивается количество дней на единицу, если количество действий восстановил значение по умолчанию.

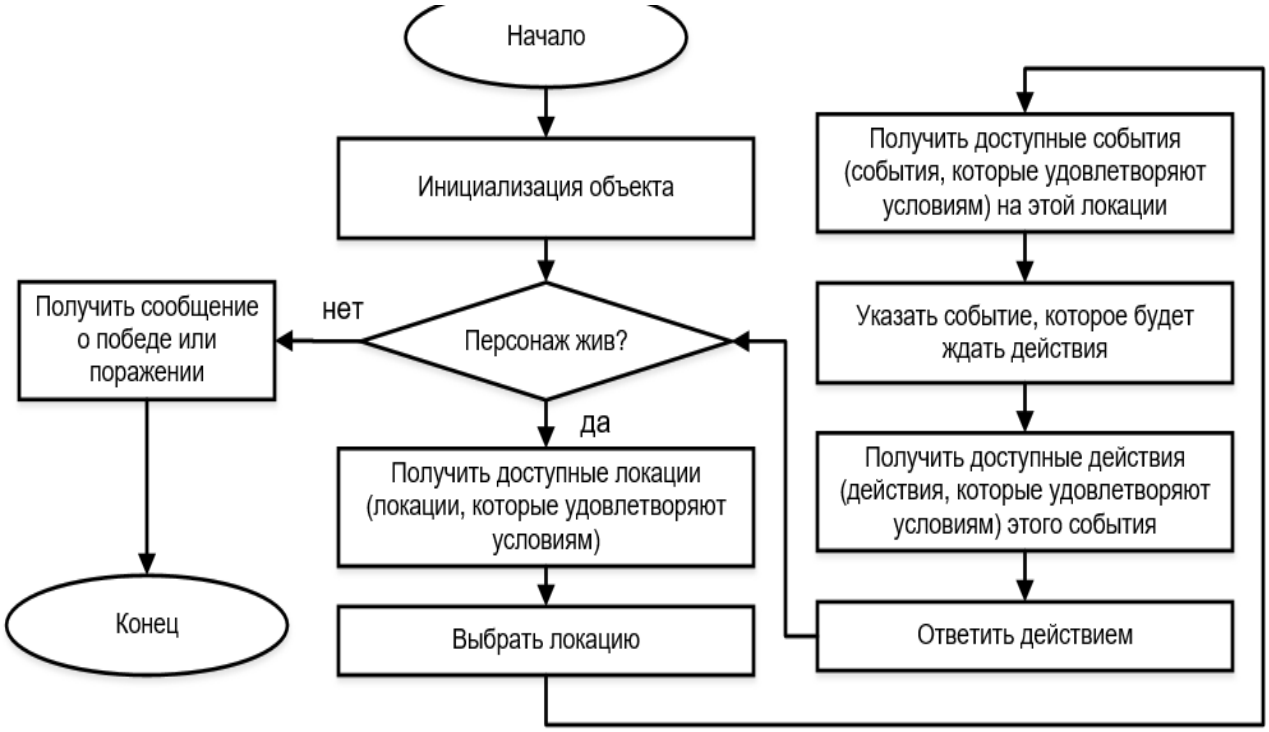

Рисунок 8.1.2 – Алгоритм работы игры

8.2 Инициализация объекта движка

Для того, чтобы можно было работать с базой данных, объект движка необходимо проинициализировать объект с движком. Для этого есть следующие функции:

- public int Init(string _connectionpath) – инициализация соединения с БД; параметр – путь до файла; возвращает 0, если подключение было успешным, 1 при неудаче;
- public int Create(string _connectionpath) – создаём базу данных по указанному пути в соответствии со структурой системы; параметр – путь до файла; возвращает 0, если подключение было успешным, 1 при неудаче;
- public int InitSQL(string _connectionpath) – инициализация соединения с БД; параметр – строка подключения; возвращает 0, если подключение было успешным, 1 при неудаче;
- public int CreateSQL(string _connectionpath) – создаём базу данных по указанному пути в соответствии со структурой системы; параметр – строка подключения; возвращает 0, если подключение было успешным, 1 при неудаче.

8.3 Наполнение игры контентом

Наполнение игры контентом реализуют следующие функции:

- public List<object> GetEntity(EntityName _entityname) – метод, который вызывает метод GetEntity класса SqlToEntity;
- public List<object> GetEntity(EntityName _entityname, int _id) – метод, который вызывает метод GetEntity класса SqlToEntity;
- public List<object> GetEntity(string _sql) – метод, который вызывает метод GetEntity класса SqlToEntity;
- public int SaveEntity(List<object> _entity) – метод, который вызывает метод SaveEntity класса SqlToEntity; возвращает 0, если подключение было успешным, 1 при неудаче;
- public int SaveEntity(object _entity) – метод, который вызывает метод SaveEntity класса SqlToEntity; возвращает 0, если подключение было успешным, 1 при неудаче;
- public int DeleteEntity(List<object> _entity) – метод, который вызывает метод DeleteEntity класса SqlToEntity; возвращает 0, если подключение было успешным, 1 при неудаче;
- public int DeleteEntity(object _entity) – метод, который вызывает метод DeleteEntity класса SqlToEntity; возвращает 0, если подключение было успешным, 1 при неудаче;
- public int AddEntity(List<object> _entity) – метод, который вызывает метод AddEntity класса SqlToEntity; возвращает 0, если подключение было успешным, 1 при неудаче;
- public int AddEntity(object _entity) – метод, который вызывает метод AddEntity класса SqlToEntity; возвращает 0, если подключение было успешным, 1 при неудаче.

8.3 Начало игры

За осуществление геймплея отвечают следующие функции:

- public List<Location> GetAvailableLocations() – метод, который возвращает доступные локации, т.е. возвращает локации, которые удовлетворяют условиям;
- public List<Event> GetAvailableEvents(int _locationid) – метод, который возвращает доступные события на локации, т.е. возвращает события, которые удовлетворяют условиям;
- public List<Action> GetAvailableActions(int _actionid) – метод, который возвращает доступные действия события, т.е. возвращает действия, которые удовлетворяют условиям;
- public int ActivateEvent(int _eventid) – активация события; возвращает 0, если подключение было успешным, 1 при неудаче;
- public int CancelEvent() – отмена ожидания действия на событие; возвращает 0, если подключение было успешным, 1 при неудаче;
- public List<Action> GetActionsOfActivatedEvent() – этот метод возвращает доступные действия у события, которое ожидает действия;
- public int RespondEvent(int _actionid) – ответ действием на событие; возвращает 0, если подключение было успешным, 1 при неудаче;
- public List<AvailableLocation> GetAvailableLocationsWithEvents() – метод, который возвращает все доступные локации, все доступные события на этих локациях, все доступные действия у этих событий;
- public List<AvailableEvent> GetAvailableEventsWithActions(int _locationid) – метод, который возвращает все доступные события на этой локации, все доступные действия у этих событий.

ЗАКЛЮЧЕНИЕ

Была проанализирована предметная область мобильной разработки: перспективы развития, ключевые понятия предметной области, поставленная задача, достоинства и недостатки существующих решений.

Были собраны требования заказчика: маркетинговые требования, связанные со стимулами, целями создания продукта и сути проекта, и требования к продукту, связанные с профилями пользователей и пользовательских историй.

Были выбраны и обоснованы технологии, которые мы будем использовать при разработке нашего продукта.

Была разработана база данных, согласно всем требованиям жизненного цикла баз данных.

Была разработана структура движка с использованием паттернов проектирования, принципов ООП согласно требованиям заказчика.

Также были проведены тесты работоспособности созданного решения и написана документация для пользователя.

В результате выполнения задания из ВКР была реализована система под названием «Движок симулятор профессии» согласно требованиям заказчика. Все требования заказчика были удовлетворены.

Данную систему можно модернизировать, добавив:

- текстовые характеристики персонажа;
- условия возникновения событий, связанные логическими операциями ИЛИ, И, НЕ;
- возникновение случайных событий, которые возникают утром или при входе на определённую локацию.

Поставленную цель разработка движка симулятора профессии выполнила.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Индустрия компьютерных игр – Википедия. – https://ru.wikipedia.org/wiki/Индустрия_компьютерных_игр.

2. ВШБИ-Игровая индустрия: геймдев (gamedev). – http://hsbi.hse.ru/articles/igrovaya-industriya-geymdev/.

3. ПЕРСОНАЛЬНЫЙ КОМПЬЮТЕР (ПК) - это... – https://methodological_terms.academic.ru/1329.

4. Игровая приставка – Википедия. – https://ru.wikipedia.org/wiki/Игровая_приставка.

5. Значение слова СМАРТФОН. – https://kartaslov.ru/значение-слова/смартфон.

6. Аркадный автомат – Википедия. – https://ru.wikipedia.org/wiki/Аркадный_автомат.

7. Что такое игровой движок? Не во всех словарях найдешь определение этого словосочетания. Но если мы сами попробуем. – https://kanobu.ru/pub/251719/.

8. Игровая Индустрия. - Моя идея. – idea.moi-portal.ru/igrovaya-industriya/.

9. Рынок игр в России и в мире (Mobile, Console, ПК, MMOG, Social). Динамика, структура, тендеции, тренды, прогнозы до 2018-2021 гг. Итоги 2016 г. - Онлайн-игры, спорт, киберспорт | ИКТ Аналитика на json.tv. – http://json.tv/ict_telecom_analytics_view/issledovanie-mirovogo-i-rossiyskogo-rynka-igr-2016-god-20170502014806.

10. Геймер 2.0 Новые тренды игрового рынка. – https://futurist.ru/articles/1010-geymer-2-0-novie-trendi-igrovogo-rinka.

11. Мобильные игры занимают более половины всего игрового рынка. – https://www.ixbt.com/news/2018/05/02/mobilnye-igry-zanimajut-bolee-poloviny-sego-igrovogo-rynka.html.

12. Игровая индустрия растет за счет мобильных игр – Gmbox. – https://gmbox.ru/materials/30649-igrovaya-industriya-rastet-za-schet-mobilnih-igr.

13. Unity (игровой движок) – Википедия. – https://ru.wikipedia.org/wiki/Unity_(игровой_движок).

14. Полный обзор Unity 5 – Devgam. – devgam.com/polnyj-obzor-unity-5.

15. Туториал по Unreal Engine. Часть 1: знакомство с движком / Хабр. – https://habr.com/post/344446/.

16. Орёл или решка: сравнение Unity и Unreal Engine – DTF. – https://dtf.ru/7227-orel-ili-reshka-sravnenie-unity-i-unreal-engine.

17. Туториал по Unreal Engine. Часть 2: Blueprints / Хабр. – http://devgam.com/polnyj-obzor-unity-5.

18. Разработка простой игры в Game Maker. Эпизод 0. – https://habr.com/post/255995/.

19. Химонин, Ю.И. Сбор и анализ требований к программному продукту. – https://pmi.ru/profes/Software_Requirements_Khimonin.pdf.

20. Что такое малый и средний бизнес, определения и отличия. – http://bazaidei.ru/chto-takoe-malyj-i-srednij-biznes-opredeleniya-i-otlichiya/.

21. СУБД – Lurkmore. – https://habr.com/post/344394/.

22. Введение в ORM (Object Relational Mapping) - Блог вебразработчика. – internetka.in.ua/orm-intro/.

23. Язык программирования – Национальная библиотека им. Н. Э. Баумана. – https://ru.bmstu.wiki/Язык_программирования.

24. Обзор языков программирования. – bourabai.kz/alg/classification04.htm.

25. Высокоуровневый язык программирования – Википедия. – https://ru.wikipedia.org/wiki/Высокоуровневый_язык_программирования.

26. ЯЗЫКИ ПРОГРАММИРОВАНИЯ - StudFiles. – https://studfiles.net/preview/3837635/page:2/.

27. C Sharp – Википедия. – https://ru.wikipedia.org/wiki/C_Sharp.

28. Среды разработки программного обеспечения. – nit.miem.edu.ru/sbornik/2009/plen/008.html.

29. IntelliJ IDEA для новичков: советы и секреты – GeekBrains. – https://geekbrains.ru/events/770.

30. NetBeans IDE - универсальная интегрированная среда разработки приложений | Статьи о программном обеспечении. – http://shpora.net/index.cgi?act=view &id=30970.

31. Обзор платформы Eclipse - как её использовать | Статьи о программном обеспечении. – http://bourabai.kz/alg/classification04.htm.

32. Работаем с Mono. Часть 1: Основные принципы Mono, инструменты, создание простейшего приложения. – https://www.ibm.com/developerworks/ru/library/r-define-scope-development-environment/index.html.

33. Microsoft Visual Studio – Википедия. – https://ru.wikipedia.org/wiki/Microsoft_Visual_Studio.

34. База даных - Taxslov.ru. – taxslov.ru/law/law256.htm.

35. Базы данных и СУБД. – http://timeweb.com/ru/community/articles/bazy-dannyh-i-subd-1.

36. Представление об организации БД и СУБД. Структура данных и система запросов на примерах БД различного назначения. – https://sites.google.com/ site/okotsitomsk/informatika/4-3-predstavlenie-ob-organizacii-bd-i-subd-struktura-dannyh-i-sistema-zaprosov-na-primerah-bd-razlicnogo-naznacenia.

37. Обзор систем управления базами данных (СУБД) для систем контроля и управления доступом (СКУД). – https://www.parsec.ru/articles/obzor-sistem-upravleniya-bazami-dannykh-subd-dlya-sistem-kontrolya-i-upravleniya-dostupom-skud/.

38. Базы данных и их классификация. – http://helpiks.org/9-41077.html.

39. Персональные и профессиональные СУБД. Сравнительные характеристики SQL СУБД: Oracle, SQL Server, IBM DB2, Informix. – http://shpora.net/index.cgi?act=view&id=30970.

40. Системы управления базами данных (СУБД) для систем контроля и управления доступом. – http://www.tzmagazine.ru/jpage.php?uid1=1181&uid2=1182&uid3=1194.

41. SQLite – замечательная встраиваемая БД (часть 1) / Хабр. – https://habr.com/post/149356/.

42. Введение в ORM (Object Relational Mapping) – Блог вебразработчика. – https://habr.com/post/237889/.

43. Обзор средств объектно-реляционной проекции (ORM) для платформы .NET. – https://www.arbinada.com/en/node/33.

44. Обзор средств объектно-реляционной проекции (ORM) для платформы .NET | Mechanics of software. – http://internetka.in.ua/orm-intro/.

45. Работа с Entity Framework 6. – https://professorweb.ru/my/entity-framework/6/level1/.

46. Платформа ADO.NET Entity Framework. – https://msdn.microsoft.com/ru-ru/library/bb399572(v=vs.100).aspx.

47. Толмачёв, С.П. Технологии баз данных и знаний / С.П. Толмачёв, Е.С. Толмачёва, С.Л. Замковец. – Минск: Современные знания, 2008. – 141 с.

48. Проектирование базы данных MySQL. – http://addphp.ru/materials/mysql/1_3.php.

49. Ярош, Е.С. Проектирование реляционных баз данных: учебное пособие / Е.С. Ярош. – Челябинск: Издательский центр ЮУрГУ, 2012. – 41 с.

50. Пушников, А.Ю. Введение в системы управления базами данных. Часть 1. Реляционная модель данных: Учебное пособие / А.Ю. Пушников. – Уфа: Изд-во Башкирского ун-та, 1999. – 108 с.

51. Пушников, А.Ю. Введение в системы управления базами данных. Часть 2. Нормальные формы отношений и транзакции: Учебное пособие / А.Ю. Пушников. – Уфа: Изд-во Башкирского ун-та, 1999. – 138 с.

52. Тепляков, С.И. Паттерны проектирования на платформе NET / С.И. Тепляков. – СПб.: Питер, 2016. – 320 с.

53. Паттерны ООП в метафорах / Хабр. – https://habr.com/post/136766/.

54. Что такое анти-паттерны? / Хабр. – https://habr.com/post/59005/.

55. Паттерн (шаблон) проектирования Singleton (одиночка, одноэлементное множество, синглет). – http://cpp-reference.ru/patterns/creational-patterns/singleton/.

56. Фабричный метод. – https://refactoring.guru/ru/design-patterns/factory-method.

57. Dependency injection / Хабр. – https://habr.com/post/350068/.

58. Компоновщик (Composite) | Паттерны в C# и .NET. – https://metanit.com/sharp/patterns/4.4.php.

59. Декоратор (Decorator) | Паттерны в C# и .NET. – https://metanit.com/sharp/patterns/4.1.php.

60. Приспособленец (Flyweight) | Паттерны в C# и .NET. – https://metanit.com/sharp/patterns/4.7.php.

61. Заместитель (Прокси) | Паттерны в C# и .NET. – https://metanit.com/sharp/patterns/4.5.php.

62. Цепочка Обязанностей (Chain of responsibility) | Паттерны в C# и .NET. – https://metanit.com/sharp/patterns/3.7.php.

63. Хранитель (Memento) | Паттерны в C# и .NET. – https://metanit.com/sharp/patterns/3.10.php.

64. Посетитель (Visitor) | Паттерны в C# и .NET. – https://metanit.com/sharp/patterns/3.11.php.

65. Албахари, Д. C# 7.0. Карманный справочник / Джозеф Албахари, Бен Албахари; пер. с англ. Ю. Артеменк. – М.: Вильямс, 2017. – 224 с.

66. Понятие объектно-ориентированного программирования (ООП). Классы и объекты. – http://mycsharp.ru/post/23/2013_06_20_ponyatie_obektno-orientirovannogo_programmirovaniya_(oop)_klassy_i_obekty.html.

67. Обзор языка C# — руководство по C# | Microsoft Docs. – https://docs.microsoft.com/ru-ru/dotnet/csharp/tour-of-csharp/.

68. Обзор технологии NET Framework - Программные продукты – Статьи. – http://www.interface.ru/home.asp?artId=24046.

69. Внутренний и внешний интерфейс. – https://learn.javascript.ru/internal-external-interface.

70. Швец, А. Погружение в Рефакторинг / А. Швец. – https://refactoring.guru/ru/refactoring/course

71. Швец, А. Погружение в Паттерны проектирования / А. Швец. – https://refactoring.guru/ru/design-patterns/book.