**МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ**

**УЧРЕЖДЕНИЕ ОБРАЗОВАНИЯ**

**ГОМЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ**

**УНИВЕРСИТЕТ ИМЕНИ П. О. СУХОГО**

Факультет автоматизированных и информационных систем

Кафедра «Информационные технологии»

направление специальности 1-40 05 01-01 Информационные системы и технологии (в проектировании и производстве)

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

к курсовому проекту

по дисциплине «Объектно-ориентированное программирование»

на тему: «**WINDOWS FORM ПРИЛОЖЕНИЕ ДЛЯ СИСТЕМЫ РАСПРОСТРАНЕНИЯ БИЛЕТОВ В ТЕРЕ**»

Исполнитель: студент гр. ИТП-21

Бондарев Е.Ю.

Руководитель: доцент

Курочка К.С.

Дата проверки: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Дата допуска к защите: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Дата защиты: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Оценка работы: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Подписи членов комиссии

по защите курсового проекта: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Гомель 2023

**СОДЕРЖАНИЕ**

[ВВЕДЕНИЕ 5](#_Toc135075645)

[1 ТЕХНИЧЕСКИЕ ПОДХОДЫ К РЕАЛИЗАЦИИ ПРИЛОЖЕНИЯ ДЛЯ СИСТЕМЫ ПРОДАЖИ БИЛЕТОВ 6](#_Toc135075646)

[1.1 Обзор аналогов 6](#_Toc135075647)

[1.2 Объектно-ориентированное программирование 7](#_Toc135075648)

[1.3 Сравнительный анализ языков объектно-ориентированного программирования 8](#_Toc135075649)

[1.4 Принципы *SOLID* 9](#_Toc135075650)

[1.5 Модели хранения данных 10](#_Toc135075651)

[1.6 *Windows Forms* технология 11](#_Toc135075652)

[1.7 Язык разметки документов *XML* 12](#_Toc135075653)

[1.8 Технология *LINQ* 12](#_Toc135075654)

[2 АЛГОРИТМИЧЕСКИЙ АНАЛИЗ ПОСТАВЛЕНОЙ ЗАДАЧИ И ОПИСАНИЕ ПРОГРАММНОГО КОМПЛЕКСА 14](#_Toc135075655)

[2.1 Архитектура программного комплекса 14](#_Toc135075656)

[2.2 Структура данных 14](#_Toc135075657)

[2.3 Компоненты приложения 16](#_Toc135075658)

[2.4 Структура классов разработанного программного комплекса 18](#_Toc135075659)

[3 ПОЛЬЗОВАТЕЛЬСКИЙ ИНТЕРФЕЙС И ТЕСТИРОВАНИЕ 24](#_Toc135075660)

[3.1 Пользовательский интерфейс 24](#_Toc135075661)

[3.3 Модульное тестирование приложния 27](#_Toc135075662)

[ЗАКЛЮЧЕНИЕ 30](#_Toc135075663)

[Список используемых источников 31](#_Toc135075664)

[ПРИЛОЖЕНИЕ А Графические представления интерфейса программы 32](#_Toc135075665)

[ПРИЛОЖЕНИЕ Б Листинг программы 35](#_Toc135075666)

[ПРИЛОЖЕНИЕ В Руководство системного программиста 78](#_Toc135075667)

[ПРИЛОЖЕНИЕ Г Руководство программиста 79](#_Toc135075668)

[ПРИЛОЖЕНИЕ Д Руководство пользователя 80](#_Toc135075669)

[ПРИЛОЖЕНИЕ Е Иерархическая схема классов приложения 81](#_Toc135075670)

# ВВЕДЕНИЕ

Система продажи билетов имеет важное значение для многих организаций, включая кинотеатры, театры, музеи, концертные залы и другие. Она облегчает процесс покупки билетов для клиентов и обеспечивает безопасный и удобный способ продажи билетов для организаторов мероприятий.

Эта система также может быть использована для контроля доступа на мероприятия, что обеспечивает безопасность и предотвращает мошенничество. Кроме того, система продажи билетов может предоставлять информацию о доступности мест, ценах на билеты, расписании и других деталях мероприятия. Это помогает клиентам выбрать подходящее время и место для посещения мероприятия и обеспечивает гладкое и эффективное проведение самого мероприятия.

Существует множество различных систем продажи билетов, включая онлайн-платформы, автоматизированные кассы и системы, управляемые операторами. Каждая система имеет свои преимущества и недостатки, и организации выбирают систему, которая наиболее соответствует их потребностям и бюджету.

Одним из главных преимуществ систем продажи билетов является их доступность для клиентов. Клиенты могут легко купить билеты, не выходя из дома, используя онлайн-платформы или мобильные приложения. Это особенно удобно для тех, кто не имеет времени или возможности посетить кассу в физическом месте продажи билетов.

Благодаря системам продажи билетов организации могут лучше контролировать количество проданных билетов и заранее знать количество зрителей на мероприятии.

Более того, система продажи билетов может помочь в сборе данных и аналитике. Организаторы мероприятий могут использовать данные о продажах билетов для анализа поведения клиентов и популярности определенных мероприятий. Это может помочь им в принятии решений относительно маркетинговых стратегий и улучшения опыта посетителей в будущем.

В конечном итоге, система продажи билетов является важным элементом в организации мероприятий, которая обеспечивает удобство и безопасность для клиентов и организаторов. С помощью системы продажи билетов клиенты могут легко выбирать и оплачивать желаемые места на мероприятии, а организаторы могут контролировать доступ к мероприятию и отслеживать продажи. Система продажи билетов также управляет финансами организации, позволяя ей получать доход от проданных билетов и распределять его среди всех заинтересованных сторон. Кроме того, система продажи билетов помогает планировать будущие мероприятия, анализируя данные о посещаемости, предпочтениях и отзывах клиентов. Она может повысить эффективность организации мероприятий и улучшить опыт посетителей.

# 1 ТЕХНИЧЕСКИЕ ПОДХОДЫ К РЕАЛИЗАЦИИ ПРИЛОЖЕНИЯ ДЛЯ СИСТЕМЫ ПРОДАЖИ БИЛЕТОВ

## 1.1 Обзор аналогов

В настоящее время на рынке существует огромное количество различных систем продажи билетов, предоставляемых компаниями-поставщиками. Каждая из этих систем обладает своими уникальными особенностями и функциональностью, которые могут быть востребованы для различных типов мероприятий и организаторов.

Некоторые системы продажи билетов могут предоставлять возможность онлайн-бронирования мест, что особенно удобно для крупных мероприятий, где необходимо забронировать много мест для групп посетителей. Другие системы могут предоставлять возможность создания персонализированных билетов с уникальными *QR*-кодами для более безопасного контроля доступа на мероприятие.

Для театров, как и для любых других мероприятий, существует множество систем продажи билетов, которые предоставляют различные инструменты для управления продажами и улучшения опыта посетителей. Одной из таких систем является *Ticketmaster*. Это крупнейшая система продажи билетов в мире, которая позволяет театрам создавать схему зала и бронировать места, а также отправлять электронные билеты на почту или мобильный телефон посетителя. Кроме того, *Ticketmaster* предлагает инструменты для управления продажами, маркетинга и аналитики, что может быть полезно для организаторов театральных мероприятий.

Другой системой продажи билетов, которая может быть полезна для театров, является *Eventbrite*. Эта система позволяет театрам создавать персонализированные страницы мероприятий с подробной информацией, фотографиями и видео. Театры могут использовать *Eventbrite* для продажи билетов на свои спектакли и события, а также для сбора пожертвований и регистрации гостей.

Еще одной системой продажи билетов, которая может быть полезна для театров, является *Brown Paper Tickets*. Эта система специализируется на мероприятиях с небольшим бюджетом, включая театральные спектакли. *Brown Paper Tickets* предоставляет возможность продажи билетов онлайн и в реальном времени, а также отправляет билеты на почту или мобильный телефон посетителя.

Наконец, система *Spectra* также может быть полезна для театров. Она предлагает инструменты для управления продажами, маркетинга и аналитики, а также позволяет театрам создавать персонализированные билеты с уникальными *QR*-кодами, которые могут использоваться для контроля доступа на мероприятие.

В целом, выбор системы продажи билетов для театров зависит от конкретных потребностей и желаемых функций. Каждая из упомянутых выше платформ предлагает уникальные возможности для управления продажами и улучшения опыта посетителей.

## 1.2 Объектно-ориентированное программирование

ООП (объектно-ориентированное программирование) *–* это подход к программированию, в котором программа разбивается на объекты, каждый из которых имеет свои свойства и методы. В ООП данные и функции, работающие с этими данными, объединяются в объекты, которые обладают определенным поведением.

Основные принципы объектно-ориентированного программирования включают в себя наследование, инкапсуляцию и полиморфизм. Наследование позволяет создавать новые классы на основе уже существующих, а инкапсуляция *–* скрыть детали реализации объекта от пользователя и предоставить только интерфейс для работы с ним. Полиморфизм позволяет использовать один и тот же интерфейс для работы с разными объектами, что делает код более гибким и удобным для использования.

Объектно-ориентированное программирование позволяет писать более читаемый и понятный код, что упрощает его сопровождение и развитие. В ООП используются классы, объекты, наследование и другие конструкции, которые позволяют разбивать код на логические блоки и обеспечивают лучшую организацию и структурирование кода.

ООП широко используется в современном программировании и применяется в различных областях, от создания веб-сайтов до разработки игр и мобильных приложений. Некоторые языки программирования, такие как *Java*, *Python* и *C*++, предназначены для ООП и предоставляют мощные инструменты для создания объектно-ориентированных программ.

Объектно-ориентированное программирование также позволяет использовать наследование для повторного использования кода и уменьшения дублирования. Кроме того, объектно-ориентированное программирование позволяет разделять ответственность между различными объектами, что упрощает разработку сложных систем.

Важным принципом ООП являются абстракция, полезная для описания общих понятий и выделения ключевых характеристик, и композиция, позволяющая объединять объекты в более крупные структуры.

Важным аспектом ООП является использование классов для создания объектов. Класс определяет свойства и методы объекта, а каждый созданный объект имеет свои собственные значения свойств, но использует одни и те же методы класса.

Другим важным принципом ООП является инкапсуляция, которая обеспечивает скрытие внутренней реализации объекта и предоставляет интерфейс для работы с ним. Это позволяет изменять внутреннюю реализацию объекта без влияния на другие части программы, использующие этот объект. Объектно-ориентированное программирование также обеспечивает возможность полиморфизма, который позволяет объектам разных классов иметь одинаковые методы и использоваться в качестве аргументов функций без необходимости знать их конкретный тип.

Еще одним важным аспектом ООП является использование наследования интерфейсов, который позволяет создавать новые классы на основе уже существующих интерфейсов. Это позволяет увеличивать гибкость и расширяемость программы.

Наконец, ООП поддерживает принцип композиции объектов, который позволяет создавать сложные объекты из более простых, используя их в качестве компонентов. Это позволяет создавать гибкие и масштабируемые системы.

Общие преимущества ООП включают повторное использование кода, уменьшение дублирования, упрощение разработки и поддержки кода, улучшение гибкости и расширяемости программы, а также увеличение понятности и читаемости кода.

## 1.3 Сравнительный анализ языков объектно-ориентированного программирования

Объектно-ориентированные языки программирования стали широко распространенными в различных сферах компьютерной технологии, начиная от разработки программного обеспечения до создания игр и приложений для мобильных устройств. Существует множество объектно-ориентированных языков программирования, и каждый из них имеет свои сильные и слабые стороны. В этой статье мы рассмотрим несколько из самых популярных объектно-ориентированных языков программирования и сравним их.

*Java* – один из наиболее широко используемых объектно-ориентированных языков программирования. *Java* был создан в 1995 году компанией *Sun* *Microsystems. Java* является очень популярным языком для создания приложений для мобильных устройств и веб-приложений. Он также широко используется в различных индустриальных приложениях. *Java* является статически типизированным языком программирования, что означает, что типы данных определяются на этапе компиляции.

*C++* – это объектно-ориентированный язык программирования, который разработан на основе языка *С*. Он был создан в 1983 году Бьёрном Страуструпом. *C++* является очень быстрым и эффективным языком программирования, и он широко используется для разработки игр, научных и инженерных приложений и других высокопроизводительных систем. *C++* является статически типизированным языком программирования, и для работы с ним требуется определенный уровень опыта.

*Python* – это интерпретируемый объектно-ориентированный язык программирования. *Python* был создан в 1991 году Гвидо ван Россумом. Он широко используется для разработки научных и статистических приложений, веб-приложений и приложений для машинного обучения. *Python* является динамически типизированным языком программирования, что означает, что типы данных определяются во время выполнения.

*Java, C++* и *Python* имеют свои сильные и слабые стороны. *Java* является быстрым и эффективным языком программирования, и он хорошо подходит для создания приложений для мобильных устройств, веб-приложений и больших индустриальных систем. Однако, он может быть сложным для начинающих программистов, так как требует знания множества стандартных библиотек и фреймворков.

*C++* также является очень быстрым и эффективным языком программирования, и он может быть использован для создания высокопроизводительных систем и игр. Однако, *C++* может быть сложным для изучения и требует большого опыта для написания безопасного и эффективного кода.

*Python*, в свою очередь, является очень простым и интуитивно понятным языком программирования, и он может быть использован для создания широкого спектра приложений, включая научные и статистические приложения, веб-приложения и приложения для машинного обучения. Однако, *Python* может быть не таким быстрым и эффективным, как *Java* или *C++*, особенно при работе с большими объемами данных.

В заключение, выбор объектно-ориентированного языка программирования зависит от конкретных требований проекта и уровня опыта программиста. *Java* и *C++* лучше подходят для создания крупных индустриальных приложений и высокопроизводительных систем, в то время как *Python* может быть отличным выбором для создания простых и быстрых прототипов или приложений, связанных с обработкой данных. Но, независимо от выбранного языка программирования, важно иметь хорошее понимание основ объектно-ориентированного программирования и умение применять его в практических проектах.

Выбор объектно-ориентированного языка программирования зависит от конкретных требований проекта и уровня опыта программиста. *Java, C++, C#* –это языки программирования, которые лучше всего подходят для создания крупных индустриальных приложений и высокопроизводительных систем. Однако, если вы предпочитаете создавать приложения для *Windows*, то язык *C#* станет отличным выбором, так как он обладает широкой поддержкой со стороны *Microsoft* и имеет большую базу библиотек и инструментов для разработки приложений для *Windows*. Но, независимо от выбранного языка программирования, важно иметь хорошее понимание основ объектно-ориентированного программирования и умение применять его в практических проектах.

## 1.4 Принципы *SOLID*

*SOLID* – это аббревиатура, которая означает пять принципов объектно-ориентированного программирования, разработанных Робертом Мартином (*Robert C. Martin*). Эти принципы помогают разработчикам создавать гибкие, расширяемые и легко поддерживаемые приложения.

Вот краткое описание каждого из принципов *SOLID*:

1. Принцип единственной ответственности (*Single Responsibility Principle, SRP*) - класс должен иметь только одну причину для изменения. Каждый класс должен быть ответственен только за одну часть функциональности программы. Это позволяет легко поддерживать и тестировать код.
2. Принцип открытости/закрытости (*Open/Closed Principle, OCP*) - классы должны быть открыты для расширения, но закрыты для изменения. Это означает, что вы должны использовать наследование или интерфейсы для добавления новой функциональности, а не изменять существующий код.
3. Принцип подстановки Барбары Лисков (*Liskov Substitution Principle, LSP*) - объекты должны быть заменяемы друг за друга без изменения поведения программы. Это означает, что вы должны использовать наследование, только если это абсолютно необходимо, и если классы-наследники действительно могут заменять своих родителей.
4. Принцип разделения интерфейса (*Interface Segregation Principle, ISP*) - клиенты не должны зависеть от интерфейсов, которые они не используют. Лучше создавать несколько узких интерфейсов, чем один большой. Это уменьшает связанность между классами и делает код более гибким.
5. Принцип инверсии зависимостей (*Dependency Inversion Principle, DIP*) - модули верхнего уровня не должны зависеть от модулей нижнего уровня. Оба типа модулей должны зависеть от абстракций. Это означает, что вы должны программировать на уровне абстракций, а не конкретных реализаций.
6. Применение принципов *SOLID* помогает создавать более чистый, гибкий и легко поддерживаемый код, который легче расширять и модифицировать в будущем.

## 1.5 Модели хранения данных

Существует ряд различных методов сохранения информации, каждый из которых специализирован для решения определенных задач и имеет свои сильные и слабые стороны.

Например, реляционные базы данных широко используются для хранения структурированных данных, таких как финансовые отчеты, данные клиентов и т.д. Однако они могут быть неэффективными для работы с неструктурированными данными, такими как изображения или видео.

Для хранения неструктурированных данных могут использоваться *NoSQL* базы данных, такие как *MongoDB* или *Cassandra*. Они хорошо подходят для хранения больших объемов данных, не имеющих строгой структуры, таких как данные социальных медиа, информация об устройствах Интернета вещей (*IoT*) и т.д.

Еще один подход - использование облачных хранилищ, таких как *Amazon S3* или *Microsoft Azure*. Эти хранилища обеспечивают высокую отказоустойчивость и масштабируемость, что делает их идеальными для хранения крупных объемов данных.

Независимо от выбранной модели хранения данных, важно выбирать ее в соответствии с требованиями вашей конкретной задачи, а также учитывать ограничения бюджета и доступные ресурсы.

Кроме того, при выборе модели хранения данных необходимо учитывать требования к скорости чтения и записи данных, а также к безопасности и конфиденциальности. Например, если данные содержат конфиденциальную информацию, то необходимо выбирать модель хранения данных, которая обеспечивает надежную защиту от несанкционированного доступа.

Также стоит учитывать возможность интеграции выбранной модели хранения данных с другими системами и приложениями, с которыми ваша компания работает.

В целом, выбор модели хранения данных - это компромисс между различными факторами, такими как требования к хранению данных, доступные ресурсы и бюджет, безопасность, скорость доступа и масштабируемость. Важно выбрать модель, которая наилучшим образом соответствует вашим требованиям и поможет эффективно решать поставленные задачи.

## 1.6 *Windows Forms* технология

*Windows Forms* – это платформа пользовательского интерфейса для создания классических приложений *Windows*. Она обеспечивает один из самых эффетивных способов создания классических приложений с помощью визуального конструктора в *Visual Studio*. Такие функции, как размещение визуальных элементов управления путем перетаскивания, упрощают создание классических приложений

В *Windows Forms* можно разрабатывать графически сложные приложения, которые просто развертывать, обновлять, и с которыми удобно работать как в автономном режиме, так и в сети. Приложения *Windows Forms* могут получать доступ к локальному оборудованию и файловой системе компьютера, на котором работает приложение.

Приложение *Windows Forms* представляет собой событийно-ориентированное приложение, поддерживаемое *Microsoft .NET Framework*. В отличие от пакетных программ, большая часть времени тратится на ожидание от пользователя каких-либо действий, как, например, ввод текста в текстовое поле или клика мышкой по кнопке. При выполнении пользователем какого-либо действия с формой или одним из ее элементов управления создается событие. Приложение реагирует на эти события, как задано в коде, и обрабатывает события при их возникновении.

*Windows Forms* также поддерживает многопоточность, что позволяет создавать приложения, которые могут выполнять несколько задач одновременно. Она также предоставляет широкий спектр визуальных элементов управления, таких как кнопки, поля ввода, списки и многое другое, которые могут быть легко настроены и управляемы в коде.

*Windows Forms* была представлена в .*NET Framework* 1.0 и быстро стала одной из наиболее популярных платформ для разработки классических приложений *Windows*. С развитием технологий, таких как *WPF* и *UWP, Windows Forms* по-прежнему остается популярным выбором для создания классических приложений и остается поддерживаемой в последних версиях .*NET Framework.*

## 1.7 Язык разметки документов *XML*

*XML (Extensible Markup Language)* - это один из самых распространенных языков разметки документов в мире. Его популярность объясняется тем, что *XML* является стандартным форматом для обмена информацией между различными системами и приложениями. Благодаря своей гибкости и расширяемости, *XML* используется во многих областях, включая веб-приложения, базы данных, настройки программного обеспечения, научные исследования и многое другое.

В отличие от *HTML*, который используется для создания веб-страниц, *XML* не определяет, как информация должна выглядеть на экране. Вместо этого, *XML* определяет структуру и семантику данных, что делает его более универсальным и удобным для использования в различных областях.

Файлы *XML* состоят из элементов, каждый из которых определяет структуру данных. Каждый элемент содержит открывающий и закрывающий теги, между которыми находится содержимое элемента. Некоторые элементы могут содержать атрибуты, которые задают дополнительные характеристики элемента.

Одной из особенностей *XML* является его расширяемость. Разработчики могут определять свои собственные элементы, атрибуты и правила, которые будут использоваться в файле *XML*. Это делает *XML* очень гибким и удобным для использования в различных областях.

*XML* также поддерживает возможность использования схемы (*XSD*), которая определяет структуру и типы данных, которые могут содержаться в *XML-*документе. С помощью схемы можно проверять корректность структуры и содержимого *XML*-документа, что делает его более надежным и удобным для обмена данными.

Кроме того, *XML* является кроссплатформенным и независимым от языка программирования форматом, что позволяет использовать его для обмена данными между различными операционными системами и языками программирования.

Одним из примеров применения *XML* является *SOAP (Simple Object Access Protocol)*, протокол, используемый для обмена данными между веб-службами *(web services). SOAP* использует *XML* для передачи данных, что обеспечивает стандартизацию и возможность взаимодействия между различными веб-службами.

Также *XML* широко применяется для хранения и передачи данных в базах данных. Например, язык запросов *XQuery* позволяет извлекать данные из *XML-*документов, что делает его удобным для работы с данными, организованными в иерархическую структуру.

## 1.8 Технология *LINQ*

Для работы с *LINQ* необходимо использовать набор методов расширения, которые предоставляются *.NET Framework*. Эти методы позволяют выполнять различные операции над источниками данных, такие как выборка, фильтрация, сортировка, группировка и агрегация.

Один из ключевых принципов *LINQ* – это ленивая вычислительная модель. Это означает, что запросы не выполняются немедленно, а создаются объекты, представляющие запрос, и результаты вычисляются только при необходимости. Это позволяет уменьшить количество работы, которое требуется выполнить при обработке данных, и ускорить выполнение запросов.

*LINQ to SQL* – это часть *LINQ*, которая позволяет работать с базами данных, используя язык *SQL. LINQ to SQL* позволяет разработчикам выполнять запросы к базам данных, используя обычный синтаксис *LINQ*. Он также предоставляет возможность автоматически генерировать объекты, которые представляют таблицы в базе данных, что упрощает работу с данными.

*LINQ to XML* – это часть *LINQ*, которая позволяет работать с *XML*-документами*. LINQ to XML* предоставляет возможность выполнения запросов к *XML*-документам с помощью синтаксиса *LINQ*. Он также позволяет создавать, изменять и удалять узлы в *XML*-документах.

В целом, *LINQ* предоставляет мощные возможности для работы с данными и значительно упрощает процесс разработки приложений. Он позволяет написать более эффективный и читаемый код, а также обеспечивает универсальный подход к работе с различными источниками данных.

*LINQ* также поддерживает концепцию отложенного выполнения запросов, что позволяет оптимизировать запросы и увеличить производительность приложений. Это достигается благодаря тому, что *LINQ* не выполняет запрос непосредственно при его создании, а только при обращении к результатам запроса.

Кроме того, *LINQ* также позволяет создавать собственные операторы запросов, что дополнительно упрощает и оптимизирует процесс работы с данными. Это позволяет разработчикам создавать более гибкие и универсальные приложения, которые могут легко адаптироваться под различные потребности.

Наконец, *LINQ* также поддерживает использование лямбда-выражений, что позволяет упростить и ускорить процесс написания запросов. Лямбда-выражения представляют собой короткий и компактный синтаксис, который позволяет создавать функции-анонимы и передавать их в качестве аргументов.

# 2 АЛГОРИТМИЧЕСКИЙ АНАЛИЗ ПОСТАВЛЕНОЙ ЗАДАЧИ И ОПИСАНИЕ ПРОГРАММНОГО КОМПЛЕКСА

## 2.1 Архитектура программного комплекса

В рамках данного курсового проекта разрабатывается универсальное и надежное приложение, которое позволяет эффективно и удобно продавать билеты на спектакли в театре и вести их учет. Для создания приложения используется мощный и гибкий язык программирования C#, а в качестве источника данных используются XML-файлы, что обеспечивает быстрый доступ и удобство в работе с информацией. Для доступа к коллекциям используется эффективная и удобная технология LINQ, что позволяет быстро находить необходимые данные. Для создания графического интерфейса применяется инструмент Windows form. В есть возможность многопользовательского доступа к системе, что обеспечивает максимальную целостность и безопасность данных. Это позволяет использовать приложение на практике в любом месте и в любых условиях.

## 2.2 Структура данных

Для успешного выполнения курсовой работы необходимо разработать хранилище данных в виде базы данных основанных на XML-файлов. База данных обеспечивает надежность и целостность данных, что является важным аспектом при создании программного комплекса. В данном курсовом проекте будет использоваться XML-файлы в качестве источника данных:

– выделение сущностей и их атрибутов, которые будут храниться в базе данных, и формирование по ним таблиц;

– определение уникальных идентификаторов (первичных ключей) объектов, которые хранятся в строках таблицы;

– определение отношений между таблицами с помощью внешних ключей;

– нормализация данных в базе данных.

Основные задачи проектирования базы данных:

– хранение в XML-файлах всей необходимой информации;

– возможность получения данных по всем необходимым запросам;

– избыточность и дублирование данных;

– целостность базы данных.

База данных состоит из четырех таблиц:

– *SpectacleModel;*

–*TicketsModel;*

–*UsersModel.*

Модель *SpectacleModel* содержит данные о спектаклях, которые включают в себя название спектакля, информацию об авторе и жанре, дату проведения спектакля, уникальный идентификатор каждого спектакля, словарь категорий и цен на каждую категорию, а также количество свободных мест на спектакле. Для связывания таблицы *SpectacleModel* с таблицей *Genres* используется внешний ключ. Этот ключ позволяет установить связь между таблицами на основе значений в полях. В данном случае, таблица *Genres* содержит информацию о жанрах спектаклей, которая связана с таблицей *SpectacleModel* через внешний ключ, который ссылается на поле *Genre*. Данные в таблице *SpectacleModel* могут быть использованы для мониторинга продаж билетов на спектакли, планирования расписания спектаклей и анализа популярности жанров и авторов.

Таблица *TicketModel* содержит информацию о билетах на спектакли. Эта таблица включает в себя шесть полей. Поле *Id* хранит уникальный идентификатор билета, а поле *ticketId* также хранит *id* билета, которое используется для связи с другими таблицами. Поле *Owner* хранит уникальное имя владельца билета из таблицы *UsersModel.* Это поле используется для связи с таблицей пользователей, которая содержит информацию о зарегистрированных пользователях системы. Поля *Date, Title, Category* и *Price* связаны с таблицей *SpectacleModel* по внешнему ключу *Date*. Это означает, что каждый билет связан с конкретным спектаклем из таблицы *SpectacleModel*, используя дату проведения спектакля. Поле *Date* в таблице *TicketModel* связывается с полем *Date* в таблице *SpectacleModel*, которое также является уникальным идентификатором спектакля. Поля *Title, Category* и *Price* в таблице *TicketModel* содержат соответствующие данные о спектакле, на который был куплен билет.

В таблице *UsersModel* содержатся данные о зарегистрированных пользователях системы. Таблица состоит из трех полей: *Login, Password* и *Role*. Поле *Login* в таблице *UsersModel* используется в качестве первичного ключа, поскольку каждый пользователь имеет уникальное имя. Поле *Password* хранит пароль пользователя, который используется для аутентификации при входе в систему. Поле *Role* является перечислением состоящим из трех ролей: *admin,* *registered* и *guest*. Роль определяет уровень доступа пользователя в системе. Пользователи с ролью *admin* имеют полный доступ ко всем функциям системы, пользователи с ролью *registered* имеют доступ только к определенным функциям, а пользователи с ролью *guest* имеют ограниченный доступ к системе.

Таблица *Log* создана для ведения статистических отчетов и содержит информацию о покупках и возвратах билетов на спектакли. Таблица состоит из семи полей, каждое из которых содержит информацию о соответствующей операции. Поле *Method* хранит информацию о типе операции - покупке или возврате билета пользователем. Поле *Owner* содержит уникальное имя владельца билета из таблицы *UsersModel* и используется для связи с таблицей пользователей, которая содержит информацию о зарегистрированных пользователях системы. Поля *SpectacleName, SpectacleDate, Category* и *Price* содержат информацию о соответствующем спектакле, которая была получена из таблицы *TicketModel*. В поле *Timestamp* хранится дата и время проведения соответствующей операции, дата и время выводятся в формате обратного преобразования даты и времени в соответствии со стандартом *ISO* 8601, например, 2015-07-17T17:04:43.4092892+03:00.

Для повышения удобочитаемости отчетности, было принято решение о конвертации *XML*-файла с информацией о покупках и возвратах билетов на спектакли в формат *Excel*. При конвертации сохраняются все столбцы из таблицы *Log*. Кроме того, для лучшей визуализации данных, было установлено окрашивание строк в зеленый цвет для информации о покупках билетов и в красный цвет для информации о возвратах билетов.

* 1. **Компоненты приложения**

Разработка приложения на основе компонентов - это очень эффективный способ создания программного обеспечения. Каждый компонент включает в себя определенную функциональность, что позволяет разрабатывать их независимо от других, ускоряя процесс разработки и облегчая поддержку кода в будущем.

При использовании приложения, пользователь взаимодействует с программой через графический интерфейс, который содержит элементы управления, такие как кнопки. Код, связанный с интерфейсом и классами, реализующими логику программы, находится в графических компонентах интерфейса. При нажатии на кнопку пользователь вызывает событие, которое запускает код, работающий с классами, реализующими логику программы. Эти классы затем обращаются к классам, работающим с источником данных. Таким образом, можно создать общую схему работы приложения, не обращая внимания на конкретную реализацию этих классов (обобщенная функциональная схема приложения представлена на рисунке 2.1).

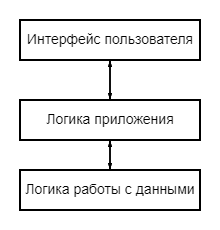


Рисунок 2.1 – Обобщённая функциональная схема приложения

Каждый из этих блоков может быть разработан отдельно, что делает изменение их отдельно друг от друга более простым и безопасным. Одино из преимуществ использования компонентов заключается в том, что если необходимо изменить способ хранения данных, например, перейти от *XML*-файлов к базе данных, необходимо изменить только несколько классов, ответственных за работу с данными. В таком случае другие компоненты приложения не затрагиваются, что делает процесс изменения более простым и безопасным.

Классы, работающие с данными, обычно не зависят от источника данных, что означает, что они не знают, откуда получены данные, и им важно только получить нужную информацию, вызвав соответствующий метод класса, отвечающего за работу с данными. Такой подход делает алгоритмы приложения более универсальными, поскольку они могут использоваться в различных системах без серьезных изменений.

Для корректной работы приложения важно правильно реализовать классы, которые будут работать с разными типами объектов. Классы должны содержать информацию, необходимую для работы с объектами, например, класс «билет» должен содержать информацию о купленном билете, дате покупки и сведения о пользователе, совершившем покупку. Классы также должны иметь соответствующий функционал для работы с объектами.

Для удобства работы с объектами приложения необходимо создать следующие объекты: «билет», «спектакль», «пользователь», «модель пользователя», «типы пользователей» (администратор, гость, авторизованный пользователь), «учет продажи билетов» и «фабрику пользователей». Каждый объект имеет свои особенности и функционал, например, объект «учет продажи билетов» содержит информацию о проданных билетах на спектакль, а объект «фабрика пользователей» создает пользователей определенного типа.

Важно учесть, что различные типы пользователей могут иметь разный доступ к функционалу приложения, например, только у администратора может быть доступ к функционалу управления пользователями.

Кроме того, каждый класс должен иметь определенный набор методов для работы с объектами. Например, у класса «билет» должны быть методы для создания, редактирования и удаления билетов, а у класса «пользователь» – методы для аутентификации, регистрации и изменения учетных данных и просмотра информации о спектаклях.

Для удобства работы с объектами можно использовать различные структуры данных, такие как массивы, списки, словари и т.д. Например, можно использовать перечисление для хранения информации ролях пользователей и категориях спектаклей.

Кроме того, важно учитывать особенности взаимодействия между объектами. Например, при продаже билетов необходимо учитывать информацию о спектакле и количестве доступных мест, а также проверять права доступа пользователя при редактировании информации о других пользователях.

В целом, разработка приложения, работающего с различными типами объектов, требует внимательного проектирования архитектуры и правильной реализации классов и методов для работы с объектами.

## 2.4 Структура классов разработанного программного комплекса

Для облегчения группировки и структурирования данных, было решено создать несколько папок:

– папка *img* содержит все картинки содержащиеся в курсовом проекте*;*

– папка *Logs* содержит классы для работы и формирования статистических отчетов;

– папка *Models* содержит сущности описанные выше;

– папка *Repositories* содержит три класса содержащих основные *CRUD* операции для работы со всеми базами данных в проекте;

– папка *Services* содержит классы для взаимодействия графического интерфейса с классами находящимися в папке *Repositories*;

– папка *Users* содержит классы пользователей с определенными ролями;

– папка *XMLData* содержит *XML*-файлы содержащие данные о жанрах, спектаклях, билетах и пользователях, также в папке хранится информация о статистических отчетах.

Для работы с формированием статических отчетов были созданы специальные классы *TicketServicesXmlLoggingDecorator* и *XmlToExcel*. Класс *TicketServicesXmlLoggingDecorator* является декоратором класса *TicketServices* и дополняет методы *AddTicket* и *DeleteTicket,* которые добавляют и удаляют билеты в базе данных. Заполнение данными происходит в методе *LogOperation*, в результате которого в XML-файл добавляется элемент, содержащий поля, описывающие вид операции, данные о пользователе, информацию о спектакле и время проведения операции.

Для конвертации *XML*-файла в формат *Excel* был создан класс *XmlToExcel.* Класс конвертации содержит метод разделения данных в соответствии с видом исходного *XML*-файла и добавляет окраску полей в зависимости от вида операции.

Класс *SpectacleManager* содержит методы добавления, удаления, обновления и вывода всех пользователей.

Метод *GetAll* возвращает коллекцию объектов типа *SpectacleModel*, которая содержит информацию о представлениях, полученную из XML-документа. Метод использует *LINQ to XML* для выбора всех элементов *spectacle* класса *SpectacleModel* из корневого элемента *XML*-документа и преобразует каждый элемент *spectacle* в объект *SpectacleModel.* Для каждого элемента *spectacle* метод выбирает значения свойств объекта *SpectacleModel* из соответствующих элементов *XML*-документа. Кроме того, метод преобразует строковые значения элементов *XML* в соответствующие типы данных, такие как *DateTime* и *Int32*, используя методы *Parse*. Также, метод использует метод *ToDictionary* для преобразования элементов *category* в словарь, где ключом является значение атрибута *name*, а значением - десятичное число, полученное из значения элемента *category*.

Метод *Add* добавляет новый объект *SpectacleModel* в *XML*-документ. Сначала метод выполняет проверку входных данных с помощью метода *DataValidate*, который определен ниже в коде. Если данные корректны, то метод продолжает свое выполнение. Далее, метод проверяет, есть ли уже спектакль на указанную дату в *XML*-документе. Если есть, то метод генерирует исключение с сообщением об ошибке. Затем, метод получает идентификатор жанра по названию с помощью метода *GetGenreIdByName*. Метод создает новый элемент *spectacle* с помощью класса *XElement* и заполняет его свойствами объекта *SpectacleModel*, переданного в качестве параметра. Значения для свойств *title*, *author*, *genre* и *freePlase* извлекаются непосредственно из объекта *SpectacleModel*. Значение свойства *date* преобразуется в строку в формате *yyyy-MM-dd* и добавляется в новый элемент *spectacle*. Для свойства *categories* из объекта *SpectacleModel* метод создает новый элемент *category* и добавляет его к новому элементу *spectacle*. Каждый элемент *category* содержит значение из словаря *Categories* в свойстве *Value* и ключ из словаря *Categories* в атрибуте *name*. Наконец, метод добавляет новый элемент *spectacle* к корневому элементу *XML*-документа и сохраняет изменения в файл. Если элемент *spectacle* с таким же названием уже существует, то метод перезапишет его значениями из переданного объекта *SpectacleModel*.

Метод *Update* обновляет значения свойств объекта *SpectacleModel* в *XML*-документе. Сначала метод выполняет проверку входных данных с помощью метода *DataValidate*, который определен где-то в коде. Если данные корректны, то метод продолжает свое выполнение. Затем метод вызывает метод *GetElement(item)*, который получает элемент *XML*-документа, соответствующий переданному объекту *SpectacleModel*. Далее, метод обновляет значения свойств объекта *SpectacleModel* в *XML*-документе с помощью метода *SetElementValue.* Для свойств *author*, *genre* и *freePlase* метод устанавливает новые значения, переданные в объекте *SpectacleModel*. Значение свойства *date* преобразуется в строку в формате *yyyy-MM-dd* и устанавливается в качестве нового значения свойства *date* элемента *XML*-документа. Затем метод проходит по каждому элементу *category* элемента *XML*-документа, связанному с переданным объектом *SpectacleModel*. Для каждого элемента *category* метод проверяет, существует ли элемент с атрибутом *name*, равным ключу текущего элемента словаря *Categories* объекта *SpectacleModel*. Если элемент существует, то метод устанавливает новое значение свойства *Value* элемента *category.* Если элемент не существует, то метод генерирует исключение с сообщением об ошибке.

Метод *Delete* удаляет элемент *XML*-документа, соответствующий объекту *SpectacleModel*, переданному в качестве аргумента метода. Сначала метод вызывает метод *GetElement(item),* который получает элемент *XML*-документа, соответствующий переданному объекту *SpectacleModel*. Затем метод вызывает метод *Remove()* для полученного элемента, который удаляет его из *XML*-документа. Наконец, метод сохраняет изменения в *XML*-документе с помощью метода *Save*.

Метод *GetElement* ищет элемент *XML*-документа, соответствующий объекту *SpectacleModel*, переданному в качестве аргумента метода. Сначала метод получает дату из объекта *SpectacleModel* и приводит ее к типу *DateTime*, чтобы использовать ее для поиска элемента в *XML*-документе. Затем метод вызывает метод *FirstOrDefault* для выборки первого элемента из корневого элемента *XML*-документа, который содержит элементы *spectacle* с датой, соответствующей переданной дате. Если элемент найден, то метод возвращает его. Если элемент не найден, то метод выбрасывает исключение *ArgumentException*. Метод проверяет корректность переданных данных объекта *SpectacleModel.* Он создает временный *XML*-элемент, используя переданные данные, и добавляет к нему элементы *category* из словаря *Categories*, содержащего категории и их стоимости. Затем метод создает новый *XML*-документ на основе временного элемента, и использует метод *Validate* для проверки его соответствия схемам *XML*-документа, заданным в поле *schemas*. Если при проверке данных была обнаружена ошибка, метод выбрасывает исключение *ArgumentException* с сообщением об ошибке. Если ошибок не было обнаружено, метод возвращает *true*.

Также в классе находятся четыре метода для работы с жанрами спектаклей. Метод *GetGenreIdByName* получает на вход название жанра (представленное в виде строки) и ищет элемент в документе, который соответствует данному названию. Если элемент найден и его атрибут *id* может быть преобразован в целочисленное значение, то метод возвращает это значение в качестве идентификатора жанра. Если жанр не найден в базе данных, метод выбрасывает соответствующее исключение с сообщением.

Cтатический метод с именем *GetGenreNameById* принимает на вход целочисленный параметр *id*. Метод ищет в документе жанры (*genre*), ищет элемент, у которого атрибут *id* совпадает со значением, переданным в параметре *id*, и возвращает текстовое значение (*Value*) найденного элемента. Если жанр с заданным *id* не найден, метод выбрасывает исключение *ArgumentException* с сообщением об ошибке.

Метод *AddGenre* добавляет новый жанр в *XML*-документ, содержащий информацию о жанрах спектаклей. Сначала метод определяет максимальный идентификатор жанра среди уже существующих записей в документе. Затем создается новый элемент *genre* с указанным именем и новым идентификатором, который увеличивается на один от максимального значения. Этот элемент добавляется в корневой элемент документа. После этого происходит сохранение документа в файл.

Метод *GetAllGenres* возвращает список всех жанров, которые содержатся в файле с жанрами. В методе используется *LINQ to XML* для выборки элементов с тегом *genre* из корневого элемента *XML*-документа *genreDoc*. Затем из каждого элемента выбирается значение *Value* и добавляется в список. В итоге метод возвращает список всех жанров, представленных в *XML*-файле.

Статический класс *TicketManager* создан для взаимодействия с *XML*-файлом содержащем данные о билетах. Метод GetAll возвращает все билеты, представленные в *XML*-документе в виде списка объектов *TicketModel*. Для каждого элемента *ticket* метод создает новый объект *TicketModel*, заполняя его свойства значениями из соответствующих элементов *ticket* и *spectacle* в *XML*-документе. При заполнении свойства *Title* используется метод *ShowSpectacle* класса *SpectacleServices*, который возвращает объект *SpectacleModel* для спектакля, соответствующего дате, указанной в элементе *ticket*. Свойство *Category* заполняется на основе значения элемента *category* в *XML*-документе, которое преобразуется в значение перечисления *Categorias*. Полученные объекты T*icketModel* добавляются в результирующий список и возвращаются из метода.

Метод *Add* добавляет новый билет в *XML*-файл с информацией о покупателе, дате, категории, цене и присваивает уникальный идентификатор для нового билета. Если данные не проходят валидацию, выбрасывается исключение *ArgumentException*. После добавления билета *XML*-файл сохраняется.

Метод *Delete* удаляет из файла билет с указанным идентификатором *ticketId*. Если билет с таким идентификатором найден в файле, то он удаляется из дерева элементов XML и изменения сохраняются в файле. Если билет не найден, выбрасывается исключение с сообщением о том, что билет с указанным идентификатором не существует.

*DataValidate* осуществляет валидацию объекта типа *TicketModel* путем создания временного экземпляра типа *XElement*, заполненного данными из объекта *TicketModel*, и проверки его на соответствие заданной схеме *XSD* при помощи метода *Validate* класса *XDocument*. Если валидация прошла успешно, метод возвращает значение *true*, иначе – *false*.

Класс *UserManager* предназначен для обработки и управления данными о пользователях в соответствующем *XML*-файле. В классе находится семь методов для работы с базой данных.

Метод *GetAll* возвращает список всех пользователей из *XML*-документа. Он использует *LINQ to XML* для выборки элементов *user* из корневого элемента *XML*-документа, а затем создает и инициализирует объекты *UserModel* из элементов *user*. Каждый объект *UserModel* получает свои значения свойств из элементов *login*, *password* и *role*. Значение свойства *Role* является строкой, которую нужно преобразовать в соответствующее значение перечисления *Role*.

Метод *Add* добавляет нового пользователя в файл *XML* базы данных. Если данные о пользователе не проходят проверку на валидность или если пользователь с таким логином уже существует, выбрасывается исключение. Затем создается новый элемент *XML* с информацией о пользователе, а затем добавляется в корневой элемент документа. Документ сохраняется в файле. Важно отметить, что сохранение файла происходит через промежуточный *MemoryStream*, чтобы предотвратить запись поврежденного файла, если что-то пойдет не так во время сохранения.

Метод *Update* обновляет информацию о пользователе в *XML*-файле. Если переданный в метод объект *UserModel* содержит неверные данные, то выбрасывается исключение *ArgumentException.* Если пользователь с таким логином не существует, то также выбрасывается исключение *ArgumentException*. В противном случае метод ищет в *XML*-файле пользователя с переданным логином, обновляет его пароль и роль и сохраняет изменения в *XML*-файле.

Метод *Delete* удаляет пользователя из *XML*-файла по его логину. Сначала метод проверяет, что такой пользователь существует в базе, если пользователь не найден, то выбрасывается исключение. Если пользователь найден, то соответствующий элемент удаляется и изменения сохраняются в файле.

Метод *UserValid* проверяет, существует ли в *XML*-файле запись о пользователе с логином, указанным в параметре *user*. Если существует, метод возвращает *false*, иначе *true*.

Метод *DataValidate* производит валидацию данных пользователя *UserModel*, используя *XML*-схему. Сначала он создает новый экземпляр *XElement*, который содержит переданный в метод объект *UserModel*. Затем он создает новый экземпляр *XDocument*, содержащий только этот элемент, и вызывает метод *Validate*, передавая ему этот документ и обработчик ошибок. Если в результате валидации не было обнаружено ошибок, метод возвращает *true*, иначе – false.

Классы *SpectacleServices, TicketServices* и *UserServices* – это важная часть приложения, которая обеспечивает корректную работу методов графического интерфейса и классов, которые работают с соответствующими базами данных. Каждый из этих классов содержит методы для добавления, удаления и изменения соответствующих данных в базе данных. Без этих классов приложение было бы не в состоянии осуществлять взаимодействие с базой данных и предоставлять пользователю все необходимые функции. Например, *SpectacleServices* содержит методы для добавления новых спектаклей, удаления существующих спектаклей и обновления информации о существующих спектаклях. Аналогично, *TicketServices* и *UserServices* предоставляют методы для работы с билетами и пользователями соответственно.

Классы *Administrator, Guest* и *Registered* наследуются от класса *User* и реализуют соответствующие их правам методы. Базовый класс *User* содержит основные методы отображения спектаклей, так как по условию курсовой работы пользователи из любой категории могут просматривать информацию о спектаклях в том или ином виде. Для этого в классе были добавлены методы *ShowAllSpectacles* и *ShowSpectacle* из класса *SpectacleServices*.

Класс *Registered* расширяет функциональность базового класса *User*, предоставляя методы для работы с данными пользователя, такие как *UpdateUserData* для обновления личных данных, а также методы из класса *TicketServices* для добавления, удаления и обновления данных о билетах. Кроме того, класс Registered предоставляет метод *GetThisSpectacle* для просмотра более подробной информации о спектаклях. Эти методы доступны только для зарегистрированных пользователей с соответствующими правами доступа.

Класс *Administrator*, также наследуясь от класса *User*, имеет доступ к административной панели, которая позволяет управлять спектаклями, пользователями и билетами. В частности, у класса *Administrator* есть методы для добавления, редактирования и удаления спектаклей, пользователей и билетов, а также методы для генерации статистических отчетов.

Класс *Guest*, наследуясь от класса *User*, имеет возможность просматривать информацию о спектаклях, но не имеет доступа к функционалу регистрации, авторизации и работы с билетами.

Класс *UserFactory* представляет фабрику пользователей, которая создает объекты пользователей (*User*) в зависимости от их роли (*Role*). Конструктор класса *UserFactory* принимает три параметра типа *UserServices, SpectacleServices* и *TicketServices*, которые представляют сервисы для работы с пользователями, спектаклями и билетами соответственно. Метод *CreateUser* принимает объект *UserModel* и на основе его поля Role создает новый объект пользователя, соответствующий данной роли. Если роль пользователя – *Role.admin*, то создается объект типа *Administrator*, если *Role.registered*, то создается объект типа *Registered*, в противном случае создается объект типа *Guest*. Для созданных объектов пользователей задается логин, а также передаются сервисы *SpectacleServices, UserServices* и *TicketServices*, которые были переданы в конструктор класса.

Преимущества использования фабрики пользователей заключаются в том, что она позволяет создавать объекты пользователей с помощью одного метода, основываясь на определенном параметре (роли пользователя). Это делает код более гибким и удобным для дальнейшей разработки и поддержки, так как если в будущем необходимо добавить новую роль пользователя, достаточно будет расширить метод *CreateUser*, а не переписывать весь код. Кроме того, фабрика пользователей позволяет изолировать создание объектов пользователей от остального кода приложения, что способствует соблюдению принципа единственной ответственности и повышает его модульность и расширяемость. Также использование фабрики пользователей облегчает тестирование кода, так как при необходимости можно создавать объекты пользователей с разными ролями для проверки различных сценариев работы приложения.

# 3 ПОЛЬЗОВАТЕЛЬСКИЙ ИНТЕРФЕЙС И ТЕСТИРОВАНИЕ

## 3.1 Пользовательский интерфейс

Помимо этого, приложение также должно обеспечивать возможность работы с различными объектами программы, такими как спектакли, авторы, пользователи, билеты и жанры. Каждый из этих объектов требует отдельного окна для работы с ним. Таким образом, в программе предусмотрены следующие графические окна:

– *MainForm* – главное окно приложения, которое содержит основную информацию о билетах и пользователях;

– *LoginForm* – окно, которое содержит в себя форму авторизации;

– *RegistrForm* – окно, предназначенное регистрации нового пользователя;

– *TicketBuyForm* – окно, предназначенное для покупки билетов и редактирования данных о спектаклях;

– *UserTicketsForm* – окно для просмотра и удалениия билетов указанного пользователя;

При запуске приложения пользователь увидит приветственное окно *LoginForm*, которое является общим для всех типов пользователей и отображено на рисунке А.1. Это окно содержит два поля для ввода данных о пользователе, такие как имя пользователя и пароль, а также три кнопки: «Войти», «Войти как гость» и «Регистрация».

Если пользователь выберет кнопку «Регистрация», то откроется форма RegistrForm, которая также представлена на рисунке А.2. Форма содержит поля для ввода имени и пароля нового пользователя. После ввода данных, пользователь должен подтвердить регистрацию, и введенные данные будут проходить проверку. При успешной регистрации новый пользователь будет создан в *XML*-файле, а форма регистрации автоматически закроется.

Если же пользователь выберет кнопку «Войти как гость», то он попадет на форму *MainForm*, которая представлена на рисунке А.3. Данная форма содержит элементы, которые присущи гостю, включая таблицу с информацией о спектаклях и поля для поиска спектаклей по жанру и дате. В правом верхнем углу находится кнопка «Регистрация», при нажатии на которую пользователь сможет вернуться на форму *LoginForm*.

При успешной авторизации с ролью «Пользователь» пользователь увидит соответствующее окно MainForm. Кнопка «Регистрация» в правом верхнем углу изменится на «Выйти», и появится возможность покупки билета и просмотра информации о пользователе. Если пользователь выберет один из спектаклей, то на экране появится форма *TicketBuyForm*, которая представлена на рисунке А.4. Здесь пользователь сможет ознакомиться с информацией о выбранном спектакле и купить билет, нажав на кнопку «Купить». Также в правом верхнем углу отображается информация о пользователе, при нажатии на которую появится окно *UserTicketsForm*, которое представлено на рисунке А.5. Здесь пользователь сможет просмотреть список приобретенных билетов и отменить покупку, если это необходимо.

При авторизации с ролью «Администратор» пользователь получает доступ к различным функциям. В частности, на главной форме (*MainForm*) ему становится доступно редактирование и удаление спектаклей. Также он может создавать отчеты о продажах билетов, которые будут представлены в виде *EXCEL*-таблицы.

Помимо этого, администратор может перейти на новую вкладку «Панель администрации», на которой расположена таблица со всеми пользователями и поля для добавления и изменения их данных. Если администратор выберет конкретного пользователя в таблице, то он сможет просмотреть список его билетов и удалить их через форму *UserTicketsForm*. Также администратор может редактировать данные этого пользователя. Если же при вводе имени пользователя в соответствующее поле окажется, что такого пользователя не существует, то программа предложит администратору создать его.

Таким образом, администратор получает расширенные возможности по управлению спектаклями и пользователями, что позволяет ему более эффективно управлять системой.

Графический интерфейс для каждого окна представлен в приложении А.

## 3.3 Тестирование пользовательского интерфейса

Для начала работы с приложением необходимо открыть файл *App.exe*. После запуска приложения появится окно авторизации, где пользователь может войти в систему или продолжить работу в качестве гостя. Чтобы авторизоваться, пользователю нужно ввести свои данные и нажать кнопку «Войти». Если пользователь не зарегистрирован, он может зарегистрироваться, нажав соответствующую кнопку. Если пользователь вводит неверные данные, ему будет выведено сообщение об ошибке.

После входа в систему в качестве зарегистрированного пользователя будут доступны следующие функции:

– просмотр афишы театра;

– просмотр сведений о спектакле;

– просмотр всех приобретенных билетов;

– покупка билетов на интересующий пользователя спектакль;

– возможность отмены билета;

– поиск спектаклей по определенной дате;

– поиск спектаклей по наличию свободных мест;

– поиск спектаклей по жанру.

Для того, чтобы купить билет на спектакль, пользователю необходимо нажать на выбранный спектакль, откроет окно, что содержажит сведения о спектакле, в котором можно купить билет. В данном окне отображается информация о жанре дате и ценах на различные категории. Ниже находится кнопка «Купить», при нажатии на которую совершается покупка билета.

Когда пользователь приобретает билеты, он может в дальнейшем проверить список своих покупок. Для этого нужно перейти в главное окно программы, выбрать информацию о пользователе и затем выбрать опцию «Мои билеты». Это откроет окно, где пользователь может просмотреть свои приобретенные билеты. Если пользователь захочет вернуть билет, он может выбрать кнопку «Возврат билета». После этого билет будет удален из списка купленных и количество доступных билетов на спектакль увеличится.

Если пользователь вошел систему в качестве администратора, то ему предоставляется следующий функционал:

– добавление спектаклей;

– редактирование спектакля;

– удаление спектакля;

– добавление жанра;

– просмотр пользователей;

– редактирование пользователей;

– удаление пользователей;

– изменение цен билетов;

– формирование статистических отчетов о проданных билетах.

В окне управления спектаклями администратор может редактировать, добавлять и удалять спектакли, выбрав нужный спектакль из таблицы. Для добавления нового спектакля администратор может нажать на кнопку «Добавить» или на пустое поле таблицы. При добавлении нового спектакля администратор должен выбрать жанр из выпадающего списка, ввести дату, название и установить цены на каждую категорию билетов. Если спектакль с указанной датой уже существует, появится сообщение об ошибке.

Для редактирования спектакля администратор должен выбрать нужный спектакль в таблице. При нажатии на нужное поле появится окно для редактирования и удаления спектакля. Если необходимо добавить новый жанр, администратор может нажать на кнопку «Добавить» во всплывающем окне. Внешний вид окна для редактирования спектакля представлен на рисунке А.6.

Для работы с пользователями администратор должен перейти во вкладку «Администрация» в которой содержится таблица со всеми пользователями. Для редактирования информации о пользователе администратор должен выбрать нужного пользователя в списке и нажать на его имя. Тогда откроется окно, в котором администратор сможет изменить информацию о пользователе, а также увидеть данные о приобретенных им билетах. Если нужный пользователь отсутствует в списке, то администратор может добавить его, введя имя пользователя и нажав на кнопку «Добавить». Если такого пользователя нет в базе данных, программа предложит создать его. После добавления пользователь будет доступен в списке пользователей.

Администратор также может составлять различные статистические отчеты. Для работы с отчетами администратор должен нажать на кнопку «Отчеты» в подробной информации о пользователе. После сформируется отчет в виде EXCLE-таблицы.

## 3.3 Модульное тестирование приложния

Тестирование кода на *C#* является важным шагом в разработке программного обеспечения. Оно позволяет выявить ошибки и недочеты в коде, а также убедиться, что он работает корректно и соответствует требованиям.

Рассмотрим некоторые основные принципы тестирования кода на *C#:*

1. *Unit* – тестирование *Unit*-тестирование – это процесс тестирования отдельных компонентов кода (например, методов и функций) для проверки их работоспособности. Для написания *unit*-тестов на C# используется фреймворк *NUnit* или *Microsoft Unit Testing Framework*.
2. Интеграционное тестирование Интеграционное тестирование – это процесс проверки взаимодействия между различными компонентами приложения. Для написания интеграционных тестов на C# можно использовать *NUnit, MSTest* или другие фреймворки.
3. Автоматизированное тестирование Автоматизированное тестирование – это процесс написания скриптов для автоматического тестирования приложения. Для написания автоматизированных тестов на *C#* можно использовать фреймворк *Selenium WebDriver* или другие инструменты.

В проекте было создано отдельный проект *CourseWorkTests* для тестов. Проект содержит тори класса для тестов *SpectacleManagerTests, TicketManagerTests* и *UserManagerTests.*

В классе *SpectacleManagerTests* были созданы методы для тестирования всех возможных *CRUD* метадов: *AddAddsNewSpectacle, UpdateUpdatesSpectacle TestDataValidateValidDataReturnsTrue* и *DeleteDeletesSpectacle.*

Метод *AddAddsNewSpectacle* представляет собой юнит-тест, который проверяет, что метод *Add* объекта *SpectacleManager* правильно добавляет новый элемент в коллекцию всех «спектаклей» и корректно обновляет ее размерность.

*UpdateUpdatesSpectacle* представляет собой юнит-тест, который проверяет, что метод *Update* объекта *SpectacleManager* правильно обновляет информацию о существующем спектакле в коллекции всех «спектаклей».

*DeleteDeletesSpectacle* – юнит-тест, который проверяет, что метод *Delete* объекта *SpectacleManager* правильно удаляет спектакль из коллекции всех «спектаклей».

Тест *TestDataValidateValidDataReturnsTrue* является юнит-тестом для метода *DataValidate* объекта *SpectacleManager.* Метод *DataValidate* принимает объект *SpectacleModel* и проверяет, что все его поля соответствуют определенным правилам валидации.

Класс *TicketManagerTests* содержит методы для тестирования класса *TicketManage.* Тест *GetAllReturnsTicketModels* проверяет, что метод *GetAll* объекта *TicketManager* возвращает коллекцию моделей *TicketModel*

*AddValidTicketModelAddsTicketToXml* проверяет, что при добавлении допустимой модели билета в коллекцию билетов (*TicketManager*), количество элементов в коллекции увеличивается на 1.

Тест *DeleteExistingTicketIdRemovesTicketFromXml* проверяет функциональность удаления билета с помощью метода *Delete* класса *TicketManager*. Сначала создается спектакль, чтобы использовать его для добавления билета, а затем берется *ID* первого билета в коллекции, полученной из метода *GetAll TicketManager*, чтобы удалить его. Затем происходит проверка того, что кол-во билетов уменьшилось на 1, и что билет с указанным *ID* больше не присутствует в коллекции. В конце удаляется созданный спектакль, чтобы не повлиять на другие тесты.

В классе *UserManagerTests* содержатся методы для класса *UserManager*. Тест *GetAllReturnsAllUsers* проверяет, что метод *GetAll* класса *UserManager* возвращает всех пользователей системы. В блоке *Arrange* создаются ожидаемые пользователи и создаётся *XML*-документ, содержащий информацию об этих пользователях. В блоке *Act* вызывается метод *GetAll*, который должен вернуть всех пользователей из системы. В блоке *Assert* проверяется, что количество возвращенных пользователей больше нуля.

*AddValidUserAddsUserToXmlDoc* проверяет, что метод *Add* класса *UserManager* успешно добавляет нового пользователя в список пользователей и что этот пользователь может быть найден в этом списке по его имени пользователя. Также этот тест в конце удаляет добавленного пользователя из списка для того, чтобы он не остался в списке после выполнения теста.

Тест *UpdateValidUserUpdatesUserInXmlDoc* проверяет, что метод *Update* менеджера пользователей обновляет информацию о существующем пользователе в документе *XML*. Сначала в методе *Arrange* создается объект *existingUser* класса *UserModel*, который представляет существующего пользователя, которого нужно обновить. Затем создается новый пользователь *updatedUser*, который будет содержать обновленную информацию, и добавляется в документ *XML* с помощью метода *Add* менеджера пользователей. После этого вызывается метод *Update* с объектом *updatedUser* в качестве аргумента. В методе *Assert* проверяется, что информация обновленного пользователя была сохранена в документе *XML* с помощью метода *GetAll*. В конце метода *updatedUser* удаляется из документа *XML* с помощью метода *Delete*.

Тест *DeleteUserExistsRemovesUser* проверяет, что метод *Delete* класса *UserManager* корректно удаляет пользователя из системы. Сначала создается новый пользователь *user* и он добавляется в систему с помощью метода *Add.* Затем вызывается метод *Delete* для удаления этого пользователя.

Далее можно представлен рузультат выполнения всех тестов (рисунок 3.1).

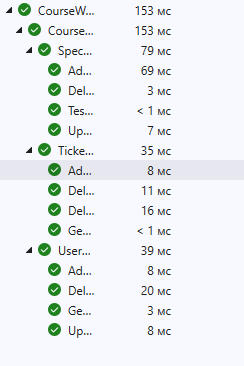


Рисунок 3.4 – Модульные тесты

На скриншоте представлены все методы классов *SpectacleManagerTests, TicketManagerTests и UserManagerTests* с результатами их прохождения. Галочка рядом с каждым методом означает, что тест успешно прошел проверку и программа работает правильно. Всего приведено 12 тестов, и все они прошли проверку.

# ЗАКЛЮЧЕНИЕ

В результате выполнения курсового проекта было разработано *Windows form* приложение для системы распространения билетов в театре и учета проданных билетов на спектакли в театре. Приложение было разработано на языке *C#* с в качестве источника данных использовались *XML*-файлы*.* Для доступа к данным было использована технология *LINQ.*

Результатом разработки курсового проекта, является приложение с графическим интерфейсом *Windows form*. Приложение имеет многопользовательский доступ к системе, что решает проблему с целостностью данных, а также с безопасностью системы.

Это приложение имеет простой интерфейс, который доступен для всех пользователей. Оно предназначено для тех, кто хочет посетить театр и заказать билеты на представление, а также для театров, чтобы пользователи могли просмотреть представления и заказать билеты. Приложение многофункционально и легко в использовании. С его помощью пользователь может просматривать доступные представления, а также использовать кнопку поиска для упрощения поиска нужного представления. Пользователь также может добавлять и удалять билеты. Администратор может добавлять и удалять пользователей, изменять тип и цену представления, а также добавлять новые представления в театр. Результаты курсового проекта показали, что все задачи были успешно выполнены и реализованы в полном объеме.

# Список используемых источников

1. Практическое руководство к курсовому проектированию по курсу «Ин­форматика» для студентов технических специальностей дневной и заочной форм обучения – Гомель: ГГТУ им. П.О. Сухого, 2019. – 32 с.

2. Рихтер Джеффри. CLR via C#. Программирование на платформе Microsoft .NET Framework 4.5 на языке C# – ООО Издательство «Питер», 2013. – 896 с.

3. Шилдт Герберт. C# 4.0: полное руководство: учебное пособие – ООО «И.Д. Вильямс», 2011. – 41 с.

4. Мухортов В.В., Рылов В.Ю. Объектно-ориентированное программирование, анализ и дизайн: Методическое пособие. – Новосибирск, издательство «ИМ СО РАН», 2002. – 108 с.

5. Крис Фиайли. Быстрый старт– Издательство «Москва», 2017. – 12 с.

6. MSDN [Электронный ресурс].: Объект Database (DAO) – Режим доступа: https://docs.microsoft.com/ru-ru/office/client-developer/access/desktop-database-reference/database-object-dao – Дата доступа: 12.03.2020

7. Metanit [Электронный ресурс].: Введение в ADO.NET – Режим доступа: https://metanit.com/sharp/adonet/1.1.php – Дата доступа: 12.03.2020

8. MSDN [Электронный ресурс].: Рекомендации по модульному тестированию для .NET Core и .NET Standard – Режим доступа: https://docs.microsoft.com/ru-ru/dotnet/core/testing/unit-testing-best-practices – Дата доступа: 13.03.2020

9. Шварц Б., Зайцев П., Ткаченко В. Ш33 MySQL по максимуму. 3-е изд. - СПб.: Питер, 2018. - 864 с.: ил. - (Серия «Бестселлеры O'Reilly»).

10. Professor Web [Электронный ресурс].: Появление LINQ – Режим доступа: https://professorweb.ru/my/LINQ/base/level1/1\_1.php – Дата доступа: 15.03.2020

# ПРИЛОЖЕНИЕ А

(обязательное)

**Графические представления интерфейса программы**

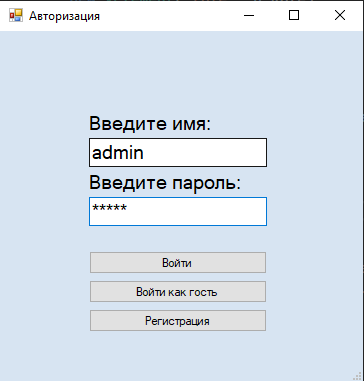


Рисунок А1 – Форма авторизации

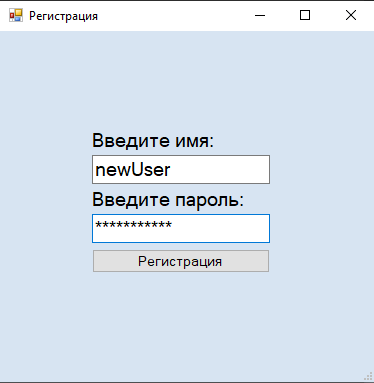


Рисунок А2 – Форма регистрации

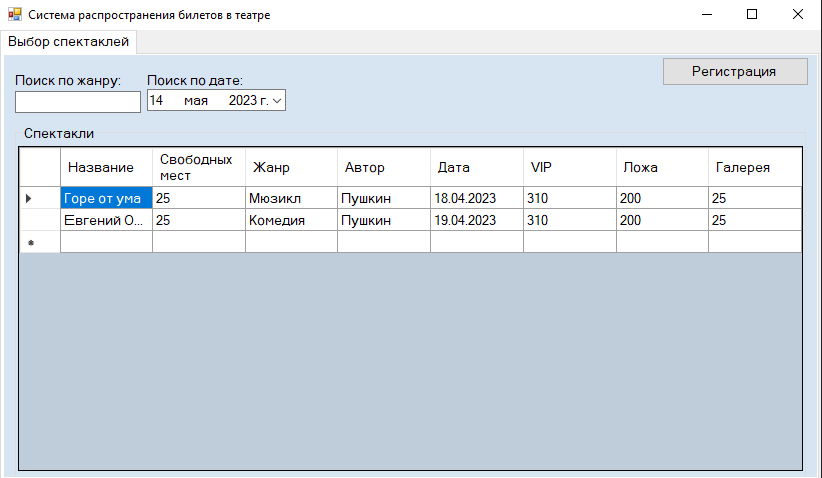


Рисунок А3 – Главное окно пользователя

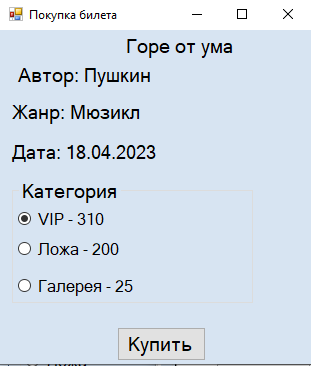


Рисунок А4 – Форма покупки билета на спектакль

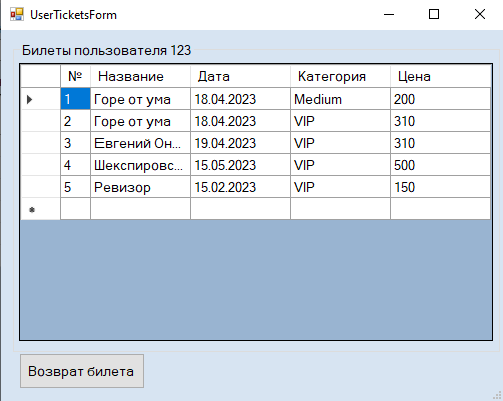


Рисунок А5 – Форма информации о билетах

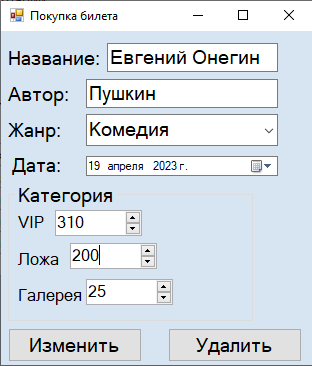


Рисунок А6 – Форма изменения информации о билетах

# ПРИЛОЖЕНИЕ Б

(обязательное)

**Листинг** **программы**

SpectacleModel

using System.Collections.Generic;

using System;

using System.Windows.Forms;

public enum Categorias

{

VIP,

Medium,

Standart

}

/// <summary>

/// Класс сущности спектакля

/// </summary>

public class SpectacleModel

{

public string Title { get; set; }

public string Author { get; set; }

public string Genre { get; set; }

public DateTime Date { get; set; }

public Dictionary<Categorias, decimal> Categories { get; set; }

public int FreePlace { get; set; }

}

TicketModel

using System;

/// <summary>

/// Класс сущности билета

/// </summary>

public class TicketModel

{

public int Id { get; set; }

public string Owner { get; set; }

public DateTime Date { get; set; }

public string Title { get; set; }

public Categorias Category { get; set; }

public decimal Price { get; set; }

}

UserModel

public enum Role

{

admin,

registered,

guest

}

/// <summary>

/// Класс сущности пользователя

/// </summary>

public class UserModel

{

public string Login { get; set; }

public string Password { get; set; }

public Role Role { get; set; }

}

SpectacleManager

using System.Collections.Generic;

using System.Linq;

using System.Xml.Linq;

using System;

using System.Xml.Schema;

using System.Collections.ObjectModel;

/// <summary>

/// Предоставляет методы для работы с спектаклями в XML-документе.

/// </summary>

public static class SpectacleManager

{

private static readonly string \_xmlFilePath;

private static readonly string \_xmlGanrePath;

private static readonly XDocument \_xmlDoc;

private static readonly XmlSchemaSet \_schemas;

private static readonly XDocument \_genreDoc;

static SpectacleManager()

{

\_xmlFilePath = "C:\\Users\\Evgeni\\Desktop\\CourseWork\\App\\XMLData\\spectacle.xml";

\_xmlGanrePath = "C:\\Users\\Evgeni\\Desktop\\CourseWork\\App\\XMLData\\genres.xml";

\_xmlDoc = XDocument.Load(\_xmlFilePath);

\_schemas = new XmlSchemaSet();

\_schemas.Add(null, "C:\\Users\\Evgeni\\Desktop\\CourseWork\\App\\XMLData\\spectacle.xsd");

\_genreDoc = XDocument.Load(\_xmlGanrePath);

}

/// <summary>

/// Получить все спектакли.

/// </summary>

/// <returns>Список всех спектаклей.</returns>

public static IEnumerable<SpectacleModel> GetAll()

{

return \_xmlDoc.Root.Elements("spectacle").Select(x => new SpectacleModel

{

Title = x.Element("title").Value,

Author = x.Element("author").Value,

Genre = GetGenreNameById(Int32.Parse(x.Element("genre").Value)),

Date = DateTime.Parse(x.Element("date").Value),

Categories = x.Elements("category").ToDictionary(y => (Categorias)Enum.Parse(typeof(Categorias), y.Attribute("name").Value), y => decimal.Parse(y.Value)),

FreePlace = Int32.Parse(x.Element("freePlase").Value)

});;

}

/// <summary>

/// Добавить спектакль.

/// </summary>

/// <param name="item">Новый спектакль.</param>

/// <exception cref="ArgumentException">Исключение, если спектакль на эту дату уже существует.</exception>

public static void Add(SpectacleModel item)

{

if (DataValidate(item))

{

DateTime date = item.Date.Date;

if (GetAll().Any(x => x.Date.Date == date))

{

throw new ArgumentException($"Спектакль на дату {date.ToShortDateString()} уже существует.");

}

int genreId = GetGenreIdByName(item.Genre);

XElement newSpectacle = new XElement("spectacle",

new XElement("title", item.Title),

new XElement("author", item.Author),

new XElement("genre", genreId),

new XElement("date", item.Date.ToString("yyyy-MM-dd")),

new XElement("freePlase", item.FreePlace)

);

foreach (var category in item.Categories)

{

newSpectacle.Add(new XElement("category", category.Value.ToString(),

new XAttribute("name", category.Key)));

}

\_xmlDoc.Root.Add(newSpectacle);

\_xmlDoc.Save(\_xmlFilePath);

}

}

/// <summary>

/// Работа с жанрами

/// </summary>

/// <summary>

/// Получить Id жанра по названию.

/// </summary>

/// <param name="name">Название жанра</param>

/// <returns>Id искомого жанра</returns>

public static int GetGenreIdByName(string name)

{

var genreElement = \_genreDoc.Descendants("genre")

.FirstOrDefault(x => x.Value == name);

if (genreElement != null && int.TryParse(genreElement.Attribute("id").Value, out int genreId))

{

return genreId;

}

throw new ArgumentException($"Жанр {name} не найден в базе данных.");

}

/// <summary>

/// Получить название жанра по Id.

/// </summary>

/// <param name="id">Id жанра</param>

/// <returns>Название искомого жанра</returns>

private static string GetGenreNameById(int id)

{

var genreElement = \_genreDoc.Descendants("genre")

.FirstOrDefault(x => x.Attribute("id")?.Value == id.ToString());

if (genreElement != null)

{

return genreElement.Value;

}

throw new ArgumentException($"Жанр с id {id} не найден в базе данных.");

}

/// <summary>

/// Добавить новый жанр

/// </summary>

/// <param name="name">Название жанра</param>

public static void AddGenre(string name)

{

int maxId = \_genreDoc.Root.Elements("genre").Select(g => (int)g.Attribute("id")).Max();

XElement newGenre = new XElement("genre", new XAttribute("id", maxId + 1), name);

\_genreDoc.Root.Add(newGenre);

\_genreDoc.Save(\_xmlGanrePath);

}

public static List<string> GetAllGenres()

{

return \_genreDoc.Root.Elements("genre").Select(x => x.Value).ToList();

}

/// <summary>

/// Обновить спектакль.

/// </summary>

/// <param name="item">Спектакль, который нужно обговить.</param>

public static void Update(SpectacleModel item)

{

if (DataValidate(item))

{

XElement spectacleToUpdate = GetElement(item);

spectacleToUpdate.SetElementValue("author", item.Author);

spectacleToUpdate.SetElementValue("genre", GetGenreIdByName(item.Genre).ToString());

spectacleToUpdate.SetElementValue("date", item.Date.ToString("yyyy-MM-dd"));

spectacleToUpdate.SetElementValue("freePlase", item.FreePlace);

foreach (var category in item.Categories)

{

var categoryToUpdate = spectacleToUpdate.Elements("category").FirstOrDefault(x => Enum.Parse(typeof(Categorias), x.Attribute("name").Value).ToString() == category.Key.ToString());

if (categoryToUpdate != null)

{

categoryToUpdate.SetValue(category.Value.ToString());

}

else throw new Exception(categoryToUpdate.ToString());

}

\_xmlDoc.Save(\_xmlFilePath);

}

}

/// <summary>

/// Удаляет спектакль.

/// </summary>

/// <param name="item">Спектакль, который нужно удалить.</param>

public static void Delete(SpectacleModel item)

{

XElement spectacleToDelete = GetElement(item);

spectacleToDelete.Remove();

\_xmlDoc.Save(\_xmlFilePath);

}

/// <summary>

/// Получает элемент спектакля из XML-документа.

/// </summary>

/// <param name="item">Спектакль, элемент которого нужно получить.</param>

/// <returns>Элемент спектакля.</returns>

public static XElement GetElement(SpectacleModel item)

{

DateTime date = item.Date.Date;

XElement element = \_xmlDoc.Root.Elements("spectacle").FirstOrDefault(x => DateTime.Parse(x.Element("date").Value).Date == date);

if (element != null)

{

return element;

}

else throw new ArgumentException("Элемент не найден.");

}

/// <summary>

/// Проверяет, что данные спектакля проходят валидацию.

/// </summary>

/// <param name="item">Спектакль, данные которого нужно проверить.</param>

/// <returns>true, если данные проходят валидацию, false в противном случае.</returns>

public static bool DataValidate(SpectacleModel item)

{

XElement tmpSpectacle = new XElement("spectacles",

new XElement("spectacle",

new XElement("title", item.Title),

new XElement("author", item.Author),

new XElement("genre", item.Genre),

new XElement("date", item.Date.ToString("yyyy-MM-ddTHH:mm:ss")))

);

foreach (var category in item.Categories)

{

tmpSpectacle.Element("spectacle").Add(new XElement("category", category.Value,

new XAttribute("name", category.Key)));

}

var xdoc = new XDocument(tmpSpectacle);

xdoc.Validate(\_schemas, (o, e) =>

{

throw new ArgumentException(e.Message);

});

return true;

}

}

TicketManager

using System;

using System.Collections.Generic;

using System.Linq;

using System.Xml.Linq;

using System.Xml.Schema;

using App.Services;

/// <summary>

/// Предоставляет методы для управления билетами в документе XML.

/// </summary>

public static class TicketManager

{

private static readonly string \_xmlFilePath;

private static readonly XDocument \_xmlDoc;

private static readonly XmlSchemaSet \_schemas;

private static SpectacleServices \_spectacleServices;

static TicketManager()

{

\_xmlFilePath = "C:\\Users\\Evgeni\\Desktop\\CourseWork\\App\\XMLData\\tickets.xml";

\_xmlDoc = XDocument.Load(\_xmlFilePath);

\_schemas = new XmlSchemaSet();

\_schemas.Add(null, "C:\\Users\\Evgeni\\Desktop\\CourseWork\\App\\XMLData\\tickets.xsd");

\_spectacleServices = new SpectacleServices();

}

/// <summary>

/// Получает все билеты.

/// </summary>

/// <returns>Список всех билетов.</returns>

public static IEnumerable<TicketModel> GetAll()

{

return \_xmlDoc.Root.Elements("ticket").Select(t =>

new TicketModel

{

Id = int.Parse(t.Attribute("id").Value),

Owner = t.Element("owner").Value,

Date = DateTime.Parse(t.Element("date").Value),

Title = \_spectacleServices.ShowSpectacle(DateTime.Parse(t.Element("date").Value)).Title,

Category = (Categorias)Enum.Parse(typeof(Categorias), t.Element("category").Value),

Price = int.Parse(t.Element("price").Value)

});

}

/// <summary>

/// Добавляет новый билет.

/// </summary>

/// <param name="ticket">Новый билет для добавления.</param>

/// <exception cref="ArgumentException">Выдается, когда данные недействительны.</exception>

public static void Add(TicketModel ticket)

{

if (!DataValidate(ticket))

{

throw new ArgumentException("Данные не валидны.");

}

int id = \_xmlDoc.Root.Elements("ticket").Max(t => int.Parse(t.Attribute("id").Value)) + 1;

XElement newTicket = new XElement("ticket",

new XAttribute("id", id),

new XElement("owner", ticket.Owner),

new XElement("date", ticket.Date.ToString("yyyy-MM-dd")),

new XElement("category", ticket.Category),

new XElement("price", ticket.Price)

);

\_xmlDoc.Root.Add(newTicket);

\_xmlDoc.Save(\_xmlFilePath);

}

/// <summary>

/// Удаляет билет.

/// </summary>

/// <param name="ticketId">Идентификатор удаляемого билета.</param>

/// <exception cref="ArgumentException">Выдается, когда билет с указанным идентификатором не существует.</exception>

public static void Delete(int ticketId)

{

XElement ticketToDelete = \_xmlDoc.Root.Elements("ticket")

.SingleOrDefault(t => t.Attribute("id")?.Value == ticketId.ToString());

if (ticketToDelete != null)

{

ticketToDelete.Remove();

\_xmlDoc.Save(\_xmlFilePath);

}

else

{

throw new ArgumentException($"Ticket with id {ticketId} does not exist.");

}

}

/// <summary>

/// Проверяет, что данные билета проходят валидацию.

/// </summary>

/// <param name="ticket">Спектакль, данные которого нужно проверить.</param>

/// <returns>true, если данные проходят валидацию, false в противном случае.</returns>

private static bool DataValidate(TicketModel ticket)

{

bool isValid = true;

var tmpTicket = new XElement("tickets",

new XElement("ticket",

new XAttribute("id", int.MinValue),

new XElement("owner", ticket.Owner),

new XElement("date", ticket.Date.ToString("yyyy-MM-dd")),

new XElement("category", ticket.Category),

new XElement("price", ticket.Price)

));

var xdoc = new XDocument(tmpTicket);

xdoc.Validate(\_schemas, (o, e) =>

{

isValid = false;

});

return isValid;

}

}

UserManager

using System;

using System.Collections.Generic;

using System.IO;

using System.Linq;

using System.Xml.Linq;

using System.Xml.Schema;

/// <summary>

/// Предоставляет методы для работы с пользователями в файле XML.

/// </summary>

public static class UserManager

{

private static readonly string \_xmlFilePath;

private static readonly XDocument \_xmlDoc;

private static readonly XmlSchemaSet \_schemas;

static UserManager()

{

\_xmlFilePath = "C:\\Users\\Evgeni\\Desktop\\CourseWork\\App\\XMLData\\users.xml";

\_xmlDoc = XDocument.Load(\_xmlFilePath);

\_schemas = new XmlSchemaSet();

\_schemas.Add(null, "C:\\Users\\Evgeni\\Desktop\\CourseWork\\App\\XMLData\\users.xsd");

}

/// <summary>

/// Возвращает все объекты UserModel из файла XML.

/// </summary>

/// <returns>Все объекты UserModel из файла XML.</returns>

public static IEnumerable<UserModel> GetAll()

{

return \_xmlDoc.Root.Elements("user").Select(u =>

new UserModel

{

Login = u.Element("login").Value,

Password = u.Element("password").Value,

Role = (Role)Enum.Parse(typeof(Role), u.Element("role").Value)

});

}

/// <summary>

/// Добавляет новый объект UserModel в файл XML.

/// </summary>

/// <param name="user">Объект UserModel для добавления.</param>

/// <exception cref="ArgumentException">Вызывается, когда данные не валидны или пользователь с таким логином уже существует.</exception>

public static void Add(UserModel user)

{

if (!DataValidate(user))

{

throw new ArgumentException("Данные не валидны.");

}

if (!UserValid(user))

{

throw new ArgumentException("Пользователь с таким логином уже существует.");

}

var newUser = new XElement("user",

new XElement("login", user.Login),

new XElement("password", user.Password),

new XElement("role", user.Role)

);

\_xmlDoc.Root.Add(newUser);

using (var ms = new MemoryStream())

{

\_xmlDoc.Save(ms);

using (var file = new FileStream(\_xmlFilePath, FileMode.Create, FileAccess.Write))

{

ms.WriteTo(file);

}

}

}

// <summary>

/// Обновляет объект UserModel в файле XML.

/// </summary>

/// <param name="user">Объект UserModel для обновления.</param>

/// <exception cref="ArgumentException">Вызывается, когда данные не валидны или пользователь с таким логином не существует.</exception>

public static void Update(UserModel user)

{

if (!DataValidate(user))

{

throw new ArgumentException("Двнные не валидны.");

}

if (UserValid(user))

{

throw new ArgumentException("Пользователя с таким логином не существует.");

}

XElement userToUpdate = \_xmlDoc.Root.Elements("user")

.SingleOrDefault(u => u.Element("login").Value == user.Login);

if (userToUpdate != null)

{

userToUpdate.Element("password").Value = user.Password;

userToUpdate.Element("role").Value = user.Role.ToString();

\_xmlDoc.Save(\_xmlFilePath);

}

}

// <summary>

/// Удаление объекта UserModel в файле XML.

/// </summary>

/// <param name="user">Объект UserModel для удаления.</param>

/// <exception cref="ArgumentException">Вызывается, когда пользователя не существует.</exception>

public static void Delete(UserModel user)

{

if (UserValid(user))

{

throw new ArgumentException("Пользователя с таким логином не существует.");

}

XElement userToDelete = \_xmlDoc.Root.Elements("user")

.SingleOrDefault(u => u.Element("login").Value == user.Login);

if (userToDelete != null)

{

userToDelete.Remove();

\_xmlDoc.Save(\_xmlFilePath);

}

}

// <summary>

/// Проверка уникальности пользователя.

/// </summary>

/// <param name="user">Объект UserModel для проверки валидации.</param>

private static bool UserValid(UserModel user)

{

if (\_xmlDoc.Root.Elements("user").Any(u => u.Element("login").Value == user.Login))

{

return false;

}

else return true;

}

/// <summary>

/// Проверяет, что данные пользователя проходят валидацию.

/// </summary>

/// <param name="user">Пользователь которого нужно проверить.</param>

/// <returns>true, если данные проходят валидацию, false в противном случае.</returns>

private static bool DataValidate(UserModel user)

{

bool isValid = true;

var tmpSpectacle = new XElement("users",

new XElement("user",

new XElement("login", user.Login),

new XElement("password", user.Password),

new XElement("role", user.Role)

));

var xdoc = new XDocument(tmpSpectacle);

xdoc.Validate(\_schemas, (o, e) =>

{

isValid = false;

});

return isValid;

}

}

ISpectacleServices

using System;

using System.Collections.Generic;

namespace App.Services

{

internal interface ISpectacleServices<T>

{

// Вывод всех спектаклей

IEnumerable<T> ShowAllSpectacles();

T ShowSpectacle(DateTime dateTime);

IEnumerable<T> ShowSpectacle(string spectacleTitle);

// Добавление спектакля

void AddNewSpectacle(string title, string author, string genre, DateTime date, decimal vipPrise, decimal mediumPrice, decimal standartPrice);

// Изменение спектакля

void UpdateSpectacle(string title, string author, string genre, DateTime date, decimal vipPrise, decimal mediumPrice, decimal standartPrice);

// Удаление спектакля

void DeleteSpectacle(DateTime date);

}

}

ITicketServices

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using static System.Windows.Forms.VisualStyles.VisualStyleElement.StartPanel;

namespace App.Services

{

internal interface ITicketServices<T>

{

// Метод для вывода всех пользователей

IEnumerable<T> GetTicket();

// Метод для вывода пользователя по имени

IEnumerable<T> GetTicket(string owner);

// Метод для вывода пользователя по дате

T GetTicket(int id);

// Добавление нового пользователя

void AddTicket(string userName, SpectacleModel spectacleModel, Categorias category);

// Метод для удаления пользователя

void DeletTicket(int id);

}

}

IUserServices

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace App.Services

{

internal interface IUserServices<T>

{

// Метод для вывода всех пользователей

IEnumerable<T> GetUser();

// Метод для вывода пользователя по имени

T GetUser(string name);

// Добавление нового пользователя

void AddUser(string login, string password, Role role);

// Метод для обновления пользователя по имени

void UpdateUserByName(string name, string newPassword, Role newRole);

// Метод для удаления пользователя

void DeleteUser(string name);

}

}

SpectacleServices

using System;

using System.Collections.Generic;

using System.Linq;

namespace App.Services

{

public class SpectacleServices : ISpectacleServices<SpectacleModel>

{

public SpectacleServices()

{

}

public IEnumerable<SpectacleModel> ShowAllSpectacles()

{

IEnumerable<SpectacleModel> spectacles = SpectacleManager.GetAll();

return spectacles;

}

public SpectacleModel ShowSpectacle(DateTime date)

{

IEnumerable<SpectacleModel> spectacles = SpectacleManager.GetAll();

SpectacleModel spectacle = spectacles.FirstOrDefault(x=> x.Date.ToString("d") == date.ToString("d"));

if (spectacle != null)

{

return spectacle;

}

else throw new ArgumentException($"Спектакль на дату `{date.ToString("d")}` не найден.");

}

public IEnumerable<SpectacleModel> ShowSpectacle(string genre)

{

IEnumerable<SpectacleModel> spectacles = SpectacleManager.GetAll();

IEnumerable<SpectacleModel> filteredSpectacles = spectacles.Where(x => x.Genre.ToLower() == genre.ToLower());

if (filteredSpectacles.Any())

{

return filteredSpectacles;

}

else

{

throw new ArgumentException($"Спектакли c жанром `{genre}` не найдены.");

}

}

public void AddNewSpectacle(string title, string author, string genre, DateTime date,

decimal vipPrise, decimal mediumPrice, decimal standartPrice)

{

SpectacleManager.Add(CreateSpectacleElement(title, author, genre, date, vipPrise, mediumPrice, standartPrice));

}

public void UpdateSpectacle(string newTitle, string newAuthor, string newGenre, DateTime date,

decimal newVipPrise, decimal newMediumPrice, decimal newStandartPrice)

{

SpectacleManager.Update(CreateSpectacleElement(newTitle, newAuthor, newGenre, date,newVipPrise, newMediumPrice, newStandartPrice));

}

public void AddGanre(string ganreName)

{

SpectacleManager.AddGenre(ganreName);

}

public List<string> GetAllGenres()

{

return SpectacleManager.GetAllGenres();

}

public int GetGenreIdByName(string name)

{

return SpectacleManager.GetGenreIdByName(name);

}

public void DeleteSpectacle(DateTime date)

{

SpectacleModel spectacleToDelete =SpectacleManager.GetAll().FirstOrDefault(x => x.Date.Equals(date));

if (spectacleToDelete != null)

{

SpectacleManager.Delete(spectacleToDelete);

}

else throw new ArgumentException($"Спектакль на дату `{date.ToString("d")}` не найден.");

}

private SpectacleModel CreateSpectacleElement(string title, string author, string genre, DateTime date,

decimal vipPrice, decimal mediumPrice, decimal standardPrice)

{

if (string.IsNullOrWhiteSpace(title))

{

throw new ArgumentException("Поле `Название` не должено быть пустым или состоять только из пробелов");

}

if (string.IsNullOrWhiteSpace(author))

{

throw new ArgumentException("Поле `Автор` не должено быть пустым или состоять только из пробелов");

}

if (string.IsNullOrWhiteSpace(genre))

{

throw new ArgumentException("Поле `Жанр` не должено быть пустым или состоять только из пробелов");

}

if (vipPrice <= 0 || mediumPrice <= 0 || standardPrice <= 0)

{

throw new ArgumentException("Цена должна быть положительной");

}

Dictionary<Categorias, decimal> thisCategories = new Dictionary<Categorias, decimal>()

{

{ Categorias.VIP, vipPrice},

{ Categorias.Medium, mediumPrice},

{ Categorias.Standart, standardPrice}

};

return new SpectacleModel

{

Title = title,

Author = author,

Genre = genre,

Date = date,

Categories = thisCategories,

FreePlace = 25

};

}

}

}

TicketServices

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace App.Services

{

public class TicketServices : ITicketServices<TicketModel>

{

private static SpectacleServices \_spectacleServices;

public TicketServices()

{

\_spectacleServices = new SpectacleServices();

}

public void AddTicket(string userName, SpectacleModel spectacleModel, Categorias category)

{

if (spectacleModel.FreePlace > 0)

{

spectacleModel.FreePlace -= 1;

SpectacleManager.Update(spectacleModel);

TicketManager.Add(CreateTicketElement(userName, spectacleModel, category));

}

else throw new ArgumentException($"На спектакль {spectacleModel.Title} нет свободных мест.");

}

public void DeletTicket(int id)

{

SpectacleModel spectacle = \_spectacleServices.ShowSpectacle(GetTicket(id).Date);

spectacle.FreePlace += 1;

SpectacleManager.Update(spectacle);

TicketManager.Delete(id);

}

public IEnumerable<TicketModel> GetTicket()

{

return TicketManager.GetAll();

}

public IEnumerable<TicketModel> GetTicket(string owner)

{

IEnumerable<TicketModel> tickets = TicketManager.GetAll().Where(x => x.Owner == owner); ;

if (tickets != null)

{

return tickets;

}

else throw new ArgumentException($"Билет пользователя {owner} не найден.");

}

public TicketModel GetTicket(int id)

{

IEnumerable<TicketModel> tickets = TicketManager.GetAll();

TicketModel ticket = tickets.FirstOrDefault(x => x.Id == id);

if (ticket != null)

{

return ticket;

}

else throw new ArgumentException($"Билет на c id={id} не найден.");

}

private TicketModel CreateTicketElement(string userName, SpectacleModel spectacleModel, Categorias category)

{

return new TicketModel

{

Owner = userName,

Date = spectacleModel.Date,

Title = spectacleModel.Title,

Category = category,

Price = spectacleModel.Categories[category]

};

}

}

}

UserServices

using System;

using System.Collections.Generic;

using System.Linq;

namespace App.Services

{

public class UserServices : IUserServices<UserModel>

{

public UserServices()

{

}

public IEnumerable<UserModel> GetUser()

{

return UserManager.GetAll();

}

public UserModel GetUser(string userLogin)

{

UserModel user = UserManager.GetAll().FirstOrDefault(x => x.Login == userLogin);

if (user != null)

{

return user;

}

throw new ArgumentException($"Пользователь '{userLogin}' не найден.");

}

public void AddUser(string login, string password, Role role)

{

UserManager.Add(CreateUserElement(login, password, role));

}

public void UpdateUserByName(string name, string newPassword, Role newRole)

{

UserModel thisUser = UserManager.GetAll().FirstOrDefault(x => x.Login == name);

if (thisUser != null)

{

UserManager.Update(CreateUserElement(name, newPassword, newRole));

}

else throw new ArgumentException($"Пользователь с логином '{thisUser.Login}' не найден.");

}

public void DeleteUser(string name)

{

UserModel userToDelete = UserManager.GetAll().FirstOrDefault(x => x.Login == name);

if (userToDelete != null)

{

UserManager.Delete(userToDelete);

}

else throw new ArgumentException($"Пользователь с логином '{userToDelete.Login}' не найден.");

}

private UserModel CreateUserElement(string login, string password, Role role)

{

return new UserModel

{

Login = login,

Password = password,

Role = role

};

}

}

}

User

using App.Services;

using System;

using System.Collections.Generic;

public abstract class User : UserModel

{

public readonly SpectacleServices \_spectacleService;

public User(SpectacleServices spectacleService)

{

\_spectacleService = spectacleService;

}

public IEnumerable<SpectacleModel> ViewSpectacle()

{

return \_spectacleService.ShowAllSpectacles();

}

public IEnumerable<SpectacleModel> ViewSpectacle(string genre)

{

return \_spectacleService.ShowSpectacle(genre);

}

public SpectacleModel ViewSpectacle(DateTime date)

{

return \_spectacleService.ShowSpectacle(date);

}

}

Registered

using App.Services;

using System;

using System.Collections.Generic;

using static System.Windows.Forms.VisualStyles.VisualStyleElement.StartPanel;

public class Registered : User

{

private TicketServices \_ticketServices;

private UserServices \_userServices;

public Registered(SpectacleServices spectacleService, UserServices userServices, TicketServices ticketServices) : base(spectacleService)

{

\_ticketServices = ticketServices;

\_userServices = userServices;

}

public void UpdateUser(string userName, string newPassword)

{

\_userServices.UpdateUserByName(userName, newPassword, Role.registered);

}

public IEnumerable<TicketModel> GetTicket()

{

return \_ticketServices.GetTicket(this.Login);

}

public IEnumerable<TicketModel> GetTicket(string owner)

{

return \_ticketServices.GetTicket(owner);

}

public TicketModel GetTicket(int id)

{

return \_ticketServices.GetTicket(id);

}

public void AddTicket(string userName, DateTime spectacleDate, Categorias category)

{

\_ticketServices.AddTicket(userName, GetThisSpectacle(spectacleDate), category);

}

public void DeletTicket(int id)

{

\_ticketServices.DeletTicket(id);

}

private SpectacleModel GetThisSpectacle(DateTime spectacleDate)

{

return \_spectacleService.ShowSpectacle(spectacleDate);

}

}

Guest

using App.Services;

using System;

public class Guest : User

{

public Guest(SpectacleServices spectacleService) : base(spectacleService)

{

}

}

Administrator

using App.Services;

using System;

using System.Collections.Generic;

public class Administrator : User

{

private UserServices \_userServices;

private TicketServices \_ticketServices;

public Administrator(SpectacleServices spectacleService, UserServices userServices, TicketServices ticketServices) : base(spectacleService)

{

\_userServices = userServices;

\_ticketServices = ticketServices;

}

public void AddSpectacle(string title, string author, string genre, DateTime date,

decimal vipPrise, decimal mediumPrice, decimal standartPrice)

{

\_spectacleService.AddNewSpectacle(title, author, genre, date, vipPrise, mediumPrice, standartPrice);

}

public void UpdateSpectacle(string newTitle, string newAuthor, string newGenre, DateTime date,

decimal newVipPrise, decimal newMediumPrice, decimal newStandartPrice)

{

\_spectacleService.UpdateSpectacle(newTitle, newAuthor, newGenre, date, newVipPrise, newMediumPrice, newStandartPrice);

}

public void DeleteSpectacle(DateTime date)

{

\_spectacleService.DeleteSpectacle(date);

}

public void AddGanre(string name)

{

\_spectacleService.AddGanre(name);

}

public List<string> GetAllGenres()

{

return \_spectacleService.GetAllGenres();

}

public int GetGenreIdByName(string name)

{

return \_spectacleService.GetGenreIdByName(name);

}

public void AddeUser(string login, string password, Role role)

{

\_userServices.AddUser(login, password, role);

}

public IEnumerable<UserModel> GetUser()

{

return \_userServices.GetUser();

}

public UserModel GetUser(string login)

{

return \_userServices.GetUser(login);

}

public void UpdateUser(string login, string password, Role role)

{

\_userServices.UpdateUserByName(login, password, role);

}

public void DeleteUser(string login)

{

\_userServices.DeleteUser(login);

}

public IEnumerable<TicketModel> GetTicket(string owner)

{

return \_ticketServices.GetTicket(owner);

}

public IEnumerable<TicketModel> GetTicket()

{

return \_ticketServices.GetTicket();

}

public TicketModel GetTicket(int id)

{

return \_ticketServices.GetTicket(id);

}

public void AddTicket(string userName, DateTime spectacleDate, Categorias category)

{

\_ticketServices.AddTicket(userName, GetThisSpectacle(spectacleDate), category);

}

public void DeletTicket(int id)

{

\_ticketServices.DeletTicket(id);

}

private UserModel GetThisUser(string userName)

{

return \_userServices.GetUser(userName);

}

private SpectacleModel GetThisSpectacle(DateTime spectacleDate)

{

return \_spectacleService.ShowSpectacle(spectacleDate);

}

}

TicketServicesXmlLoggingDecorator

using App.Services;

using System;

using System.Collections.Generic;

using System.IO;

using System.Reflection;

using System.Xml.Linq;

using static System.Windows.Forms.VisualStyles.VisualStyleElement.StartPanel;

namespace App

{

public class TicketServicesXmlLoggingDecorator : TicketServices

{

private readonly TicketServices \_ticketServices;

private readonly XDocument \_log;

private readonly string \_xmlFilePath = "C:\\Users\\Evgeni\\Desktop\\CourseWork\\App\\XMLData\\Log.xml";

public TicketServicesXmlLoggingDecorator(TicketServices inner)

{

\_ticketServices = inner;

\_log = XDocument.Load(\_xmlFilePath);

}

public void AddTicket(string userName, SpectacleModel spectacleModel, Categorias category)

{

\_ticketServices.AddTicket(userName, spectacleModel, category);

LogOperation("Покупка билета", userName, spectacleModel, category);

}

public void DeletTicket(int id)

{

LogOperation("Возврат билета", \_ticketServices.GetTicket(id));

\_ticketServices.DeletTicket(id);

}

public IEnumerable<TicketModel> GetTicket()

{

return \_ticketServices.GetTicket();

}

public IEnumerable<TicketModel> GetTicket(string owner)

{

return \_ticketServices.GetTicket(owner);

}

public TicketModel GetTicket(int id)

{

return \_ticketServices.GetTicket(id);

}

private void LogOperation(string methodName, string userName, SpectacleModel spectacleModel, Categorias category)

{

var logEntry = new XElement("Operation",

new XElement("Method", methodName),

new XElement("Owner", userName),

new XElement("SpectacleName", spectacleModel.Title),

new XElement("SpectacleDate", spectacleModel.Date.ToString("d")),

new XElement("Category", category),

new XElement("Price", spectacleModel.Categories[category]),

new XElement("Timestamp", DateTime.Now.ToString("o"))

);

\_log.Root.Add(logEntry);

\_log.Save("C:\\Users\\Evgeni\\Desktop\\CourseWork\\App\\XMLData\\log.xml");

}

private void LogOperation(string methodName, TicketModel ticketModel)

{

var logEntry = new XElement("Operation",

new XElement("Method", methodName),

new XElement("Owner", ticketModel.Owner),

new XElement("SpectacleName", ticketModel.Title),

new XElement("SpectacleDate", ticketModel.Date.ToString("d")),

new XElement("Category", ticketModel.Category),

new XElement("Price", ticketModel.Price),

new XElement("Timestamp", DateTime.Now.ToString("o"))

);

\_log.Root.Add(logEntry);

\_log.Save("C:\\Users\\Evgeni\\Desktop\\CourseWork\\App\\XMLData\\log.xml");

}

}

}

XmlToExcel

using System;

using System.Collections.Generic;

using System.Diagnostics;

using System.IO;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Xml.Linq;

using Excel = Microsoft.Office.Interop.Excel;

namespace App.Logs

{

public static class XmlToExcel

{

static private string \_filePath = "C:\\Users\\Evgeni\\Desktop\\CourseWork\\App\\XMLData\\Log.xml";

static private XDocument \_doc = XDocument.Load(\_filePath);

public static void CreateTable()

{

Excel.Application xlApp = new Excel.Application();

Excel.Workbook xlWorkbook = xlApp.Workbooks.Add();

Excel.\_Worksheet xlWorksheet = xlWorkbook.Sheets[1];

Excel.Range xlRange = xlWorksheet.UsedRange;

xlWorksheet.Cells[1, 1] = "Method";

xlWorksheet.Cells[1, 2] = "Owner";

xlWorksheet.Cells[1, 3] = "SpectacleName";

xlWorksheet.Cells[1, 4] = "SpectacleDate";

xlWorksheet.Cells[1, 5] = "Category";

xlWorksheet.Cells[1, 6] = "Price";

xlWorksheet.Cells[1, 7] = "Timestamp";

int row = 2;

foreach (XElement operation in \_doc.Descendants("Operation"))

{

string method = operation.Element("Method").Value;

string owner = operation.Element("Owner").Value;

string spectacleName = operation.Element("SpectacleName").Value;

string spectacleDate = operation.Element("SpectacleDate").Value;

string category = operation.Element("Category").Value;

string price = operation.Element("Price").Value;

string timestamp = operation.Element("Timestamp").Value;

xlWorksheet.Cells[row, 1] = method;

xlWorksheet.Cells[row, 2] = owner;

xlWorksheet.Cells[row, 3] = spectacleName;

xlWorksheet.Cells[row, 4] = spectacleDate;

xlWorksheet.Cells[row, 5] = category;

xlWorksheet.Cells[row, 6] = price;

xlWorksheet.Cells[row, 7] = timestamp;

if (method == "Покупка билета")

{

xlRange = xlWorksheet.Range[xlWorksheet.Cells[row, 1], xlWorksheet.Cells[row, 10]];

xlRange.Interior.Color = System.Drawing.ColorTranslator.ToOle(System.Drawing.Color.Green);

}

else if (method == "Возврат билета")

{

xlRange = xlWorksheet.Range[xlWorksheet.Cells[row, 1], xlWorksheet.Cells[row, 10]];

xlRange.Interior.Color = System.Drawing.ColorTranslator.ToOle(System.Drawing.Color.Red);

}

row++;

}

string resultLog = $"C:\\Users\\Evgeni\\Desktop\\CourseWork\\App\\Logs\\Log.xlsx";

xlWorkbook.SaveAs(new FileInfo(resultLog));

xlWorkbook.Close();

xlApp.Quit();

Process.Start(resultLog);

}

}

}

LoginForm

using App.Services;

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Linq;

using System.Reflection.Emit;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace App

{

public partial class LoginForm : Form

{

private MainForm \_maiForm;

private RegistrForm \_regForm;

private UserServices \_userServices;

public LoginForm(MainForm mainForm)

{

InitializeComponent();

\_userServices = new UserServices();

\_maiForm = mainForm;

\_regForm = new RegistrForm();

}

private void button1\_Click(object sender, EventArgs e)

{

LogIn();

}

private void registerButton\_Click(object sender, EventArgs e)

{

\_regForm.ShowDialog();

}

private void skipButton\_Click(object sender, EventArgs e)

{

UserModel nullUser = new UserModel() { Login = null, Password = null, Role = Role.guest};

\_maiForm.CreateUser(nullUser);

this.Close();

}

private void RegisterForm\_Load(object sender, EventArgs e)

{

userPasswordForm.PasswordChar = '\*';

exptLable.Text = "";

}

private void LogIn()

{

string userName = userNameForm.Text;

string userPassword = userPasswordForm.Text;

if (userName == "" || userPassword == "")

{

exptLable.Text = "Неправильно введен логин или пароль.";

userNameForm.Text = userPasswordForm.Text = "";

}

else

{

try

{

UserModel user = \_userServices.GetUser(userName);

if (user.Password == userPassword)

{

\_maiForm.CreateUser(user);

userNameForm.Text = userPasswordForm.Text = "";

this.Close();

}

else

{

exptLable.Text = "Неверный пароль.";

userPasswordForm.Text = "";

}

}

catch (ArgumentException exp)

{

exptLable.Text = exp.Message;

userNameForm.Text = userPasswordForm.Text = "";

}

}

}

}

}

MainForm

using App.Services;

using System;

using System.Collections.Generic;

using System.Collections.ObjectModel;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Linq;

using System.Net.NetworkInformation;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

using static System.Windows.Forms.VisualStyles.VisualStyleElement;

using System.IO;

using App.Logs;

namespace App

{

public partial class MainForm : Form

{

public SpectacleServices spectacleServices = new SpectacleServices();

public UserServices userServices = new UserServices();

public TicketServices ticketServices = new TicketServices();

public LoginForm loginForm;

public TicketBuyForm ticketBuyForm;

public UserTicketsForm userTicketsForm;

public dynamic \_user;

public MainForm()

{

InitializeComponent();

loginForm = new LoginForm(this);

ticketBuyForm = new TicketBuyForm();

userTicketsForm = new UserTicketsForm();

spectacleGridView.ReadOnly = true;

spectacleGridView.Columns.Add("Название", "Название");

spectacleGridView.Columns.Add("Свободных мест", "Свободных мест");

spectacleGridView.Columns.Add("Жанр", "Жанр");

spectacleGridView.Columns.Add("Автор", "Автор");

spectacleGridView.Columns.Add("Дата", "Дата");

spectacleGridView.Columns.Add("VIP", "VIP");

spectacleGridView.Columns.Add("Ложа", "Ложа");

spectacleGridView.Columns.Add("Галерея", "Галерея");

userGridView.ReadOnly = true;

userGridView.Columns.Add("Имя", "Имя");

userGridView.Columns.Add("Пароль", "Пароль");

userGridView.Columns.Add("Роль", "Роль");

userTicketsListBox.ReadOnly = true;

RemoveAdminPanel();

}

private void Form1\_Load(object sender, EventArgs e)

{

loginForm.ShowDialog();

spectacleGridView.CellClick += dataGridView1\_CellClick;

spectacleGridView.AutoSizeColumnsMode = DataGridViewAutoSizeColumnsMode.Fill;

spectacleGridView.RowHeadersWidthSizeMode = DataGridViewRowHeadersWidthSizeMode.AutoSizeToAllHeaders;

userInfo.MouseDown += new MouseEventHandler(InitializeContextMenu);

userRolePictureBox.MouseDown += new MouseEventHandler(InitializeContextMenu);

}

public void CreateUser(UserModel user)

{

UserFactory userFactary = new UserFactory(userServices, spectacleServices, ticketServices);

\_user = userFactary.CreateUser(user);

ShowWindowToUser(\_user);

ShowSpectacle();

}

public void ShowWindowToUser(Administrator user)

{

userInfo.Visible = true;

userInfo.Text = $"Пользователь: {\_user.Login}";

RegistrBut.Text = "Выйти";

addSpectacle.Visible = true;

permissionTabControl.TabPages[0].Text = "Управление спектаклями";

userRolePictureBox.Image = Image.FromFile("C:\\Users\\Evgeni\\Desktop\\CourseWork\\App\\img\\icons\\admin.png");

ShowUsers();

ShowAdminPanel();

}

public void ShowWindowToUser(Registered user)

{

userInfo.Visible = true;

userInfo.Text = $"Пользователь: {\_user.Login}";

RegistrBut.Text = "Выйти";

addSpectacle.Visible = false;

permissionTabControl.TabPages[0].Text = "Выбор спектаклей";

RemoveAdminPanel();

userRolePictureBox.Image = Image.FromFile("C:\\Users\\Evgeni\\Desktop\\CourseWork\\App\\img\\icons\\user.png");

}

public void ShowWindowToUser(Guest user)

{

userInfo.Visible = false;

RegistrBut.Text = "Регистрация";

addSpectacle.Visible = false;

permissionTabControl.TabPages[0].Text = "Выбор спектаклей";

RemoveAdminPanel();

userRolePictureBox.Image = null;

}

private void InitializeContextMenu(object sender, MouseEventArgs e)

{

if (\_user is Registered || \_user is Administrator)

{

userInfoContextMenu.Items.Clear();

userInfoContextMenu.Items.Add($"Имя: {\_user.Login}");

if (\_user is Registered)

{

userInfoContextMenu.Items.Add($"Роль: Пользователь");

ToolStripButton showUserTicketsButton = new ToolStripButton();

showUserTicketsButton.Text = "Мои билеты";

showUserTicketsButton.Click += new EventHandler(ticketMenuItem\_Click);

showUserTicketsButton.DisplayStyle = ToolStripItemDisplayStyle.ImageAndText;

userInfoContextMenu.Items.Add(showUserTicketsButton);

}

else if (\_user is Administrator)

{

userInfoContextMenu.Items.Add($"Роль: Администратор");

ToolStripButton showUserTicketsButton = new ToolStripButton();

showUserTicketsButton.Text = "Сформировать отчет";

showUserTicketsButton.Click += new EventHandler(ticketLog\_Click);

showUserTicketsButton.DisplayStyle = ToolStripItemDisplayStyle.ImageAndText;

userInfoContextMenu.Items.Add(showUserTicketsButton);

}

userInfoContextMenu.Name = "Информация о пользователе";

userInfoContextMenu.Show(userInfo, e.Location);

}

}

private void ticketMenuItem\_Click(object sender, EventArgs e)

{

if (\_user is Registered)

{

userTicketsForm.ShowDialog(\_user);

ShowSpectacle();

}

}

private void ticketLog\_Click(object sender, EventArgs e)

{

if (\_user is Administrator)

{

DialogResult result = MessageBox.Show("Cформировать статистический отчёт о продажах билетов?", "Формирование отчета",

MessageBoxButtons.YesNo, MessageBoxIcon.Question);

if (result == DialogResult.Yes)

{

XmlToExcel.CreateTable();

}

else if (result == DialogResult.No)

{

}

}

}

public void ShowSpectacle()

{

if (\_user is Administrator || \_user is Registered || \_user is Guest)

{

spectacleGridView.Rows.Clear();

IEnumerable<SpectacleModel> spectacles = \_user.ViewSpectacle();

foreach (SpectacleModel spectacle in spectacles)

{

spectacleGridView.Rows.Add(spectacle.Title, spectacle.FreePlace, spectacle.Genre, spectacle.Author, spectacle.Date.ToString("d"),

$"{spectacle.Categories[Categorias.VIP]}", $"{spectacle.Categories[Categorias.Medium]}",

$"{spectacle.Categories[Categorias.Standart]}");

mainExpString.Text = "";

}

}

}

public void ShowSpectacle(string ganre)

{

if (\_user is Administrator || \_user is Registered)

{

spectacleGridView.Rows.Clear();

try

{

IEnumerable<SpectacleModel> spectacles = \_user.ViewSpectacle(ganre);

foreach (SpectacleModel spectacle in spectacles)

{

spectacleGridView.Rows.Add(spectacle.Title, spectacle.FreePlace, spectacle.Genre, spectacle.Author, spectacle.Date.ToString("d"),

$"{spectacle.Categories[Categorias.VIP]}", $"{spectacle.Categories[Categorias.Medium]}",

$"{spectacle.Categories[Categorias.Standart]}");

mainExpString.Text = "";

}

}

catch (ArgumentException e)

{

mainExpString.Text = e.Message;

}

}

}

public void ShowSpectacle(DateTime date)

{

if (\_user is Administrator || \_user is Registered)

{

spectacleGridView.Rows.Clear();

try

{

SpectacleModel spectacle = \_user.ViewSpectacle(date);

spectacleGridView.Rows.Add(spectacle.Title, spectacle.Genre, spectacle.Author, spectacle.Date.ToString("d"),

$"{spectacle.Categories[Categorias.VIP]}", $"{spectacle.Categories[Categorias.Medium]}",

$"{spectacle.Categories[Categorias.Standart]}");

mainExpString.Text = "";

}

catch (ArgumentException e)

{

mainExpString.Text = e.Message;

}

}

}

private void ShowUsers()

{

if (\_user is Administrator)

{

userGridView.Rows.Clear();

IEnumerable<UserModel> users = \_user.GetUser();

foreach (UserModel user in users)

{

userGridView.Rows.Add(user.Login, user.Password, user.Role == Role.registered ? "Пользователь" : "Администратор");

}

}

}

private void RemoveAdminPanel()

{

if (permissionTabControl.TabPages.Count == 2) {

permissionTabControl.TabPages.Remove(AdminTabPanel);

}

}

private void ShowAdminPanel() {

if (permissionTabControl.TabPages.Count == 1)

{

permissionTabControl.TabPages.Add(AdminTabPanel);

}

}

private void searchSpectacleByGenre\_TextChanged(object sender, EventArgs e)

{

if (!string.IsNullOrEmpty(searchSpectacleByGenre.Text))

{

ShowSpectacle(searchSpectacleByGenre.Text);

}

else ShowSpectacle();

}

private void searchSpectacleByDate\_ValueChanged(object sender, EventArgs e)

{

DateTime selectedDate = searchSpectacleByDate.Value;

ShowSpectacle(selectedDate);

}

private void dataGridView1\_CellContentClick(object sender, DataGridViewCellEventArgs e)

{

}

private void dataGridView1\_CellClick(object sender, DataGridViewCellEventArgs e)

{

if (\_user is Registered || \_user is Administrator)

{

if (e.RowIndex >= 0)

{

DataGridViewRow row = spectacleGridView.Rows[e.RowIndex];

if (row.Cells[3].Value != null)

{

SpectacleModel thisSpectacle = \_user.ViewSpectacle(DateTime.Parse(row.Cells[4].Value.ToString()));

ticketBuyForm.ShowDialog(thisSpectacle, \_user);

}

else ticketBuyForm.ShowDialog(\_user);

}

else if (\_user is Administrator)

{

ticketBuyForm.ShowDialog(\_user);

}

ShowSpectacle();

}

}

private void userGridView\_CellClick(object sender, DataGridViewCellEventArgs e)

{

if (\_user is Administrator)

{

if (e.RowIndex >= 0)

{

DataGridViewRow row = userGridView.Rows[e.RowIndex];

if (row.Cells[0].Value != null)

{

UserModel thisUser = \_user.GetUser(row.Cells[0].Value.ToString());

userLoginBox.Text = thisUser.Login;

userPasswordBox.Text = thisUser.Password;

userRoleBox.Text = thisUser.Role == Role.registered ? "Пользователь" : "Администратор";

delUserButton.Visible = true;

ShowTicketsList(thisUser.Login);

}

}

else if (\_user is Administrator)

{

userLoginBox.Text = "";

userPasswordBox.Text = "";

userRoleBox.Text = "";

delUserButton.Visible = false;

}

}

}

private void addSpectacle\_Click(object sender, EventArgs e)

{

if (\_user is Administrator)

{

ticketBuyForm.ShowDialog(\_user);

ShowSpectacle();

}

}

private void RegistrBut\_Click(object sender, EventArgs e)

{

loginForm.ShowDialog();

}

private void userLoginBox\_TextChanged(object sender, EventArgs e)

{

if (\_user is Administrator)

{

try

{

UserModel newUser = \_user.GetUser(userLoginBox.Text);

userPasswordBox.Text = newUser.Password;

userRoleBox.Text = newUser.Role == Role.registered ? "Пользователь" : "Администратор";

addUserButton.Text = "Изменить";

delUserButton.Visible = true;

ShowTicketsList(newUser.Login);

}

catch (ArgumentException userNull)

{

userPasswordBox.Text = "";

userRoleBox.Text = "";

addUserButton.Text = "Добавить";

delUserButton.Visible = false;

userTicketsLable.Visible = false;

userTicketsListBox.Clear();

}

}

}

private void ShowTicketsList(string userLogin)

{

if (\_user is Administrator || \_user is Registered)

{

userTicketsLable.Visible = true;

userTicketsLable.Text = $"Билеты пользователя `{userLogin}`:";

userTicketsListBox.Clear();

IEnumerable<TicketModel> tickets = \_user.GetTicket(userLogin);

foreach (TicketModel ticket in tickets)

{

userTicketsListBox.Text += ($"Название:{ticket.Title}\nДата: {ticket.Date.ToString("d")}\nКатегория: {ticket.Category}\nЦена: {ticket.Price} р.\n\n");

}

}

}

private void AddUserButton\_Click(object sender, EventArgs e)

{

if (\_user is Administrator)

{

if (addUserButton.Text == "Добавить")

{

\_user.AddeUser(userLoginBox.Text, userPasswordBox.Text, userRoleBox.Text == "Пользователь" ? Role.registered : Role.admin);

ShowUsers();

}

else if (addUserButton.Text == "Изменить")

{

\_user.UpdateUser(userLoginBox.Text, userPasswordBox.Text, userRoleBox.Text == "Пользователь" ? Role.registered : Role.admin);

ShowUsers();

}

}

}

private void delUserButton\_Click(object sender, EventArgs e)

{

if (\_user is Administrator)

{

try

{

if (userLoginBox.Text != \_user.Login)

{

\_user.DeleteUser(userLoginBox.Text);

ShowUsers();

}

else MessageBox.Show("Невозможно удалить данного пользователя!");

}

catch (ArgumentException delUserExp)

{

MessageBox.Show(delUserExp.Message);

}

}

}

private void changeTicketsButton\_Click(object sender, EventArgs e)

{

if (\_user is Administrator || \_user is Registered)

{

userTicketsForm.ShowDialog(\_user, userLoginBox.Text);

ShowTicketsList(userLoginBox.Text);

ShowSpectacle();

}

}

}

}

RegistrForm

using App.Services;

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Text.RegularExpressions;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace App

{

public partial class RegistrForm : Form

{

private UserServices userServices;

public RegistrForm()

{

InitializeComponent();

userServices = new UserServices();

}

private void logInButtog\_Click(object sender, EventArgs e)

{

if(UserDataValid(newUserNameForm.Text, newUserPasswordForm.Text))

{

Registr();

}

}

private void RegistrForm\_Load(object sender, EventArgs e)

{

exptRegisterLable.Text = "";

newUserNameForm.Text = "";

newUserPasswordForm.PasswordChar = '\*';

newUserPasswordForm.Text = "";

}

private void Registr()

{

try

{

userServices.AddUser(newUserNameForm.Text, newUserPasswordForm.Text, Role.registered);

this.Close();

}

catch (ArgumentException exp)

{

exptRegisterLable.Text = exp.Message;

newUserNameForm.Text = newUserPasswordForm.Text = "";

}

}

public bool UserDataValid(string username, string password)

{

if (string.IsNullOrEmpty(username) || string.IsNullOrEmpty(password))

{

exptRegisterLable.Text = "Заполните поле логина и пароля";

return false;

}

string usernameRegex = @"^.{4,}$";

string passwordRegex = @"^.{4,}$";

if (!Regex.IsMatch(username, usernameRegex) || !Regex.IsMatch(password, passwordRegex))

{

exptRegisterLable.Text = "Длина логина и пароля должна быть не менее 4 символов";

return false;

}

if (username.Contains(" ") || password.Contains(" "))

{

exptRegisterLable.Text = "Логин и пароль не должны содержать пробелов";

return false;

}

string invalidCharsRegex = @"[;']";

if (Regex.IsMatch(username, invalidCharsRegex) || Regex.IsMatch(password, invalidCharsRegex))

{

exptRegisterLable.Text = "Логин и пароль содержат недопустимые символы `;'`";

return false;

}

exptRegisterLable.Text = "";

return true;

}

}

}

TicketBuyForm

using App.Services;

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Linq;

using System.Reflection.Emit;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

using static System.Windows.Forms.VisualStyles.VisualStyleElement.Button;

using static System.Windows.Forms.VisualStyles.VisualStyleElement.Window;

namespace App

{

public partial class TicketBuyForm : Form

{

private SpectacleModel \_spectacleModel;

private TicketServices \_ticketServices;

private TicketServicesXmlLoggingDecorator \_ticketServicesLog;

private dynamic \_owner;

public TicketBuyForm()

{

\_ticketServices = new TicketServices();

\_ticketServicesLog = new TicketServicesXmlLoggingDecorator(\_ticketServices);

InitializeComponent();

}

private void SpectacleTitle\_Click(object sender, EventArgs e)

{

}

private void groupBox1\_Enter(object sender, EventArgs e)

{

}

private void TicketBuyForm\_Load(object sender, EventArgs e)

{

this.ClientSize = new Size(311, 334);

AdminBuyPanel.Location = new Point(0, 0);

RegisterBuyPanel.Location = new Point(0, 0);

}

public new DialogResult ShowDialog(SpectacleModel thisSpectacle, Administrator user)

{

ClearAdminFields();

AdminBuyPanel.Visible = true;

RegisterBuyPanel.Visible = false;

\_spectacleModel = thisSpectacle;

\_owner = user;

newSpectacleName.Text = \_spectacleModel.Title;

newAuthorName.Text = \_spectacleModel.Author;

GanreBox.Text = \_spectacleModel.Genre;

newDateName.Value = \_spectacleModel.Date;

GanreBox.Items.Clear();

GanreBox.Items.AddRange(\_owner.GetAllGenres().ToArray());

newVIPPrice.Value = \_spectacleModel.Categories[Categorias.VIP];

newMediumPrice.Value = \_spectacleModel.Categories[Categorias.Medium];

newStandartPrice.Value = \_spectacleModel.Categories[Categorias.Standart];

return base.ShowDialog();

}

public new DialogResult ShowDialog(Administrator user)

{

ClearAdminFields();

AdminBuyPanel.Visible = true;

RegisterBuyPanel.Visible = false;

\_owner = user;

GanreBox.Items.Clear();

GanreBox.Items.AddRange(\_owner.GetAllGenres().ToArray());

changeTiket.Text = "Добавить";

changeTiket.Location = new Point((this.ClientSize.Width - changeTiket.Width) / 2, 297);

delSpectacle.Visible = false;

return base.ShowDialog();

}

public new DialogResult ShowDialog(SpectacleModel thisSpectacle, Registered user)

{

ClearUserFields();

AdminBuyPanel.Visible = false;

RegisterBuyPanel.Visible = true;

\_spectacleModel = thisSpectacle;

\_owner = user;

SpectacleTitle.Text = \_spectacleModel.Title;

SpectacleAuthor.Text = $"Автор: {\_spectacleModel.Author}";

SpectacleGenre.Text = $"Жанр: {\_spectacleModel.Genre}";

SpectacleDate.Text = $"Дата: {\_spectacleModel.Date.ToString("d")}";

CategoriesRadio1.Text = $"VIP - {\_spectacleModel.Categories[Categorias.VIP]}";

CategoriesRadio2.Text = $"Ложа - {\_spectacleModel.Categories[Categorias.Medium]}";

CategoriesRadio3.Text = $"Галерея - {\_spectacleModel.Categories[Categorias.Standart]}";

return base.ShowDialog();

}

public new void ShowDialog(SpectacleModel thisSpectacle, Guest user)

{

}

public new void ShowDialog(User user)

{

}

public new void ShowDialog(SpectacleModel thisSpectacle, User user)

{

}

private void ClearUserFields()

{

SpectacleTitle.Text = string.Empty;

SpectacleAuthor.Text = string.Empty;

SpectacleGenre.Text = string.Empty;

SpectacleDate.Text = string.Empty;

CategoriesRadio1.Text = string.Empty;

CategoriesRadio2.Text = string.Empty;

CategoriesRadio3.Text = string.Empty;

}

private void ClearAdminFields()

{

newSpectacleName.Text = string.Empty;

newAuthorName.Text = string.Empty;

GanreBox.Text = string.Empty;

newDateName.Value = DateTime.Today;

newVIPPrice.Value = 0;

newMediumPrice.Value = 0;

newStandartPrice.Value = 0;

}

private void BuyTicket\_Click(object sender, EventArgs e)

{

try{

Categorias category = new Categorias();

if (CategoriesRadio1.Checked) category = Categorias.VIP;

else if (CategoriesRadio2.Checked) category = Categorias.Medium;

else if (CategoriesRadio3.Checked) category = Categorias.Medium;

else throw new ArgumentException("Ни одина категория не выбрана");

\_ticketServicesLog.AddTicket(\_owner.Login, \_spectacleModel, category);

this.Close();

}

catch(ArgumentException exp)

{

MessageBox.Show(exp.Message);

}

}

private void changeTiket\_Click(object sender, EventArgs e)

{

try

{

\_owner.GetGenreIdByName(GanreBox.Text);

}

catch (ArgumentException exp)

{

DialogResult addGanreMessage = MessageBox.Show($"{exp.Message}\nДобавить жанр?", "Подтверждение", MessageBoxButtons.YesNo, MessageBoxIcon.Question);

if (addGanreMessage == DialogResult.Yes)

{

\_owner.AddGanre(GanreBox.Text);

}

return;

}

try {

if (changeTiket.Text == "Изменить")

{

var q = newDateName.Value;

\_owner.UpdateSpectacle(newSpectacleName.Text, newAuthorName.Text, GanreBox.Text, newDateName.Value,

newVIPPrice.Value, newMediumPrice.Value, newStandartPrice.Value);

}

else if (changeTiket.Text == "Добавить")

{

\_owner.AddSpectacle(newSpectacleName.Text, newAuthorName.Text, GanreBox.Text, newDateName.Value,

newVIPPrice.Value, newMediumPrice.Value, newStandartPrice.Value);

}

this.Close();

}

catch(ArgumentException exp)

{

MessageBox.Show(exp.Message);

}

}

private void GanreBox\_SelectedIndexChanged(object sender, EventArgs e)

{

}

private void delSpectacle\_Click(object sender, EventArgs e)

{

try

{

\_owner.DeleteSpectacle(newDateName.Value);

this.Close();

}

catch (ArgumentException exp)

{

MessageBox.Show(exp.Message);

}

}

private void newDateName\_ValueChanged(object sender, EventArgs e)

{

try

{

\_owner.ViewSpectacle(newDateName.Value);

changeTiket.Text = "Изменить";

changeTiket.Location = new Point(7, 297);

delSpectacle.Visible = true;

}

catch

{

changeTiket.Text = "Добавить";

changeTiket.Location = new Point((this.ClientSize.Width - changeTiket.Width) / 2, 297);

delSpectacle.Visible = false;

}

}

}

}

UserTicketsForm

using App.Services;

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace App

{

public partial class UserTicketsForm : Form

{

private TicketServices \_ticketServices;

private TicketServicesXmlLoggingDecorator \_ticketServicesLog;

private string \_userName;

private TicketModel \_activeTicket;

public UserTicketsForm()

{

InitializeComponent();

\_ticketServices = new TicketServices();

\_ticketServicesLog = new TicketServicesXmlLoggingDecorator(\_ticketServices);

userTicketsGridView.ReadOnly = true;

userTicketsGridView.Columns.Add("№", "№");

userTicketsGridView.Columns.Add("Название", "Название");

userTicketsGridView.Columns.Add("Дата", "Дата");

userTicketsGridView.Columns.Add("Категория", "Категория");

userTicketsGridView.Columns.Add("Цена", "Цена");

userTicketsGridView.Columns[0].Width = 30;

}

private void UserTicketsForm\_Load(object sender, EventArgs e)

{

}

public new DialogResult ShowDialog(Administrator user, string userName)

{

\_userName = userName;

userTicketGroupBox.Text = $"Билеты пользователя {\_userName}";

ShowTickets();

return base.ShowDialog();

}

public new DialogResult ShowDialog(Registered user)

{

\_userName = user.Login;

userTicketGroupBox.Text = $"Билеты пользователя {\_userName}";

ShowTickets();

return base.ShowDialog();

}

public new DialogResult ShowDialog(User user)

{

return base.ShowDialog();

}

public new void ShowTickets()

{

userTicketsGridView.Rows.Clear();

IEnumerable<TicketModel> tickets = \_ticketServicesLog.GetTicket(\_userName);

foreach (TicketModel ticket in tickets)

{

userTicketsGridView.Rows.Add(ticket.Id, ticket.Title, ticket.Date.ToString("d"), ticket.Category, ticket.Price);

}

}

private void userTicketsGridView\_CellClick(object sender, DataGridViewCellEventArgs e)

{

if (e.RowIndex >= 0)

{

DataGridViewRow row = userTicketsGridView.Rows[e.RowIndex];

\_activeTicket = \_ticketServicesLog.GetTicket(Int32.Parse(row.Cells[0].Value.ToString()));

}

}

private void delUserTicketButton\_Click(object sender, EventArgs e)

{

if (\_activeTicket != null)

{

DialogResult result = MessageBox.Show("Вы уверены, что хотите удалить билет на спектакль \"" + \_activeTicket.Title + "\"?", "Подтверждение удаления", MessageBoxButtons.YesNo, MessageBoxIcon.Question);

if (result == DialogResult.Yes)

{

\_ticketServicesLog.DeletTicket(\_activeTicket.Id);

ShowTickets();

}

}

else

{

MessageBox.Show("Выберите спектакль для удаления.", "Ошибка", MessageBoxButtons.OK, MessageBoxIcon.Error);

}

}

}

}

SpectacleManagerTests

using System;

using System.Collections.Generic;

using System.Collections.ObjectModel;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace CourseWorkTests

{

[TestClass]

public class SpectacleManagerTests

{

private readonly SpectacleModel \_testSpectacle = new SpectacleModel

{

Title = "Test Spectacle",

Author = "Test Author",

Genre = "Драма",

Date = DateTime.Now,

Categories = new Dictionary<Categorias, decimal>()

{

{ Categorias.VIP,100},

{ Categorias.Medium, 50 },

{ Categorias.Standart, 25 }

},

FreePlace = 25

};

[TestMethod]

public void Add\_AddsNewSpectacle()

{

// Arrange

int initialCount = SpectacleManager.GetAll().Count();

// Act

SpectacleManager.Add(\_testSpectacle);

var result = SpectacleManager.GetAll();

// Assert

Assert.AreEqual(initialCount + 1, result.Count());

Assert.IsTrue(result.Any(x => x.Title == \_testSpectacle.Title));

SpectacleManager.Delete(\_testSpectacle);

}

[TestMethod]

public void Update\_UpdatesSpectacle()

{

// Arrange

var updatedSpectacle = \_testSpectacle;

updatedSpectacle.Author = "Updated Author";

SpectacleManager.Add(\_testSpectacle);

// Act

SpectacleManager.Update(updatedSpectacle);

var result = SpectacleManager.GetAll();

// Assert

Assert.IsTrue(result.Any(x => x.Title == \_testSpectacle.Title && x.Author == updatedSpectacle.Author));

SpectacleManager.Delete(\_testSpectacle);

}

[TestMethod]

public void Delete\_DeletesSpectacle()

{

// Arrange

SpectacleManager.Add(\_testSpectacle);

var initialCount = SpectacleManager.GetAll().Count();

// Act

SpectacleManager.Delete(\_testSpectacle);

var result = SpectacleManager.GetAll();

// Assert

Assert.AreEqual(initialCount - 1, result.Count());

Assert.IsFalse(result.Any(x => x.Title == \_testSpectacle.Title));

}

[TestMethod]

public void TestDataValidate\_ValidData\_ReturnsTrue()

{

// Arrange

var item = new SpectacleModel()

{

Title = "Title",

Author = "Author",

Genre = "Genre",

Date = DateTime.Now,

Categories = new Dictionary<Categorias, decimal>()

{

{ Categorias.VIP,100},

{ Categorias.Medium, 50 },

{ Categorias.Standart, 25 }

},

};

// Act

var result = SpectacleManager.DataValidate(item);

// Assert

Assert.IsTrue(result);

}

}

}

TicketManagerTests

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace CourseWorkTests

{

[TestClass]

public class TicketManagerTests

{

private readonly SpectacleModel \_testSpectacle = new SpectacleModel

{

Title = "Test Spectacle",

Author = "Test Author",

Genre = "Драма",

Date = DateTime.Now,

Categories = new Dictionary<Categorias, decimal>()

{

{ Categorias.VIP,100},

{ Categorias.Medium, 50 },

{ Categorias.Standart, 25 }

},

FreePlace = 25

};

[TestMethod]

public void GetAll\_ReturnsTicketModels()

{

// Arrange

// Act

var result = TicketManager.GetAll();

// Assert

Assert.IsNotNull(result);

Assert.IsInstanceOfType(result, typeof(IEnumerable<TicketModel>));

}

[TestMethod]

public void Add\_ValidTicketModel\_AddsTicketToXml()

{

// Arrange

SpectacleManager.Add(\_testSpectacle);

int initialCount = TicketManager.GetAll().Count();

var validTicket = new TicketModel

{

Owner = "Valid Owner",

Date = DateTime.Now,

Category = Categorias.Medium,

Price = 100

};

// Act

TicketManager.Add(validTicket);

// Assert

Assert.AreEqual(initialCount + 1, TicketManager.GetAll().Count());

SpectacleManager.Delete(\_testSpectacle);

}

[TestMethod]

public void Delete\_ExistingTicketId\_RemovesTicketFromXml()

{

// Arrange

SpectacleManager.Add(\_testSpectacle);

int ticketIdToDelete = TicketManager.GetAll().First().Id;

int initialCount = TicketManager.GetAll().Count();

// Act

TicketManager.Delete(ticketIdToDelete);

// Assert

Assert.AreEqual(initialCount - 1, TicketManager.GetAll().Count());

Assert.IsNull(TicketManager.GetAll().SingleOrDefault(t => t.Id == ticketIdToDelete));

SpectacleManager.Delete(\_testSpectacle);

}

[TestMethod]

[ExpectedException(typeof(ArgumentException))]

public void Delete\_NonExistingTicketId\_ThrowsArgumentException()

{

// Arrange

int nonExistingTicketId = -1;

// Act

TicketManager.Delete(nonExistingTicketId);

// Assert

// ExpectedException

}

}

}

UserManagerTests

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Xml.Linq;

namespace CourseWorkTests

{

[TestClass]

public class UserManagerTests

{

XDocument xmlDoc = XDocument.Load("C:\\Users\\Evgeni\\Desktop\\CourseWork\\App\\XMLData\\users.xml");

[TestMethod]

public void GetAll\_ReturnsAllUsers()

{

// Arrange

var expectedUsers = new List<UserModel>

{

new UserModel { Login = "user1", Password = "password1", Role = Role.admin },

new UserModel { Login = "user2", Password = "password2", Role = Role.registered },

new UserModel { Login = "user3", Password = "password3", Role = Role.guest }

};

var xmlDoc = new XDocument(new XElement("users",

expectedUsers.Select(u => new XElement("user",

new XElement("login", u.Login),

new XElement("password", u.Password),

new XElement("role", u.Role)

))));

// Act

var actualUsers = UserManager.GetAll();

// Assert

Assert.IsTrue(actualUsers.ToList().Count() > 0);

}

[TestMethod]

public void Add\_ValidUser\_AddsUserToXmlDoc()

{

// Arrange

var newUser = new UserModel { Login = "user4", Password = "password4", Role = Role.admin };

// Act

UserManager.Add(newUser);

// Assert

var actualUser = UserManager.GetAll().FirstOrDefault(x => x.Login == newUser.Login);

Assert.IsNotNull(actualUser);

UserManager.Delete(newUser);

}

[TestMethod]

public void Update\_ValidUser\_UpdatesUserInXmlDoc()

{

// Arrange

var existingUser = new UserModel { Login = "user1", Password = "password1", Role = Role.admin };

var updatedUser = new UserModel { Login = "user1", Password = "newPassword", Role = Role.registered };

UserManager.Add(updatedUser);

// Act

UserManager.Update(updatedUser);

// Assert

var actualUser = UserManager.GetAll().FirstOrDefault(x => x.Login == updatedUser.Login);

Assert.IsNotNull(actualUser);

UserManager.Delete(updatedUser);

}

[TestMethod]

public void Delete\_UserExists\_RemovesUser()

{

// Arrange

var user = new UserModel

{

Login = "existinguser",

Password = "password",

Role = Role.admin

};

UserManager.Add(user);

// Act

UserManager.Delete(user);

// Assert

Assert.IsFalse(UserManager.GetAll().Any(u => u.Login == user.Login));

}

}

}

# ПРИЛОЖЕНИЕ В

(обязательное)

**Руководство системного программиста**

Программа предназначена для реализации продаж и учета проданных билетов на спектакли в театре. В программе можно производить поиск по спектаклям, зарегистрировать нового пользователя. Зарегистрированный пользователь может добавить и удалить билет. Администратор может добавить и удалить пользователя, изменять тип билета и его цену, добавлять новый спектакль и удалить его, так же формировать отчеты. Приложение было разработана на языке *C#* с использованием *XML*-файлов*.* Для доступа к данным было использовано *LINQ.*

Программный комплекс состоит из трех составных частей а именно:

– библиотека классов;

– модульное тестирование;

– *Windows form* приложение.

Для корректной работы приложения необходима следующая конфигурация ЭВМ:

– *Windows* 10 с *.NET Framework* 6;

– объем оперативной памяти не менее 2гб;

– объем видеокарты должен быть не менее 128 Мбайт;

– объем свободного места на жестком диске должен быть не менее 20мб;

– разрешение экрана 800 x 600.

Настройка приложения. Для запуска приложения требуется открыть файл с расширением *.exe.* Длявзаимодействия с приложением нужна клавиатура и мышь.

Проверка приложения. Для проверки работоспособности программы требуется запустить *Windows form* приложение, а так же имеются модульные тесты. Чтобы их запустить нужно перейти во вкладку «Тест» и нажать на кнопку «Запуск всех тестов».

Сообщения системному программисту. При тестировании программы, после ввода некорректных данных в поля логина и пароля, будет выдана ошибка. Для исправления необходимо ввести логин и пароль зарегистрированного пользователя в данной программе. Если администратор захочет создать новый отчет и такой файл уже будет существовать, возникнет ошибка. Чтобы это исправить, администратору нужно удалить файл отчетов или подтвердить изменение файла.

# ПРИЛОЖЕНИЕ Г

(обязательное)

**Руководство программиста**

Программа предназначена для театров. В программе реализуется продажи и учет проданных билетов на спектакли в театре. В программе можно производить поиск по спектаклям, зарегистрировать нового пользователя. Зарегистрированный пользователь может добавить и удалить билет. Администратор может добавить и удалить пользователя, изменять тип билета и его цену, добавлять новый спектакль и удалить его, так же формировать отчеты. Приложение было разработана на языке *C#* с помощью *XML-*файлов*.* Для доступа к данным было использовано *LINQ.* Для написания запросов было использовано *LINQ to object.*

Минимальные требования для запуска программы:

– *Windows* 10 с *.NET Framework* 6;

– объем оперативной памяти не менее 2гб;

– объем видеокарты должен быть не менее 128 Мбайт;

– объем свободного места на жестком диске должен быть не менее 20мб;

– разрешение экрана 800 x 600.

Программу можно запустить несколькими способами:

1. С помощью файла с раcширением .*exe*;
2. С помощью *Visual Studio.*

В качестве входных данных служит считывание информации из *XML*-файлов.

В качестве выходных данных служат отчеты, которые может формировать администратор.

Сообщения. Если в поля логина и пароля введены некорректные данные, приложение выдаст ошибку. Для исправления ситуации необходимо ввести логин и пароль зарегистрированного пользователя. Для поиска спектаклей необходимо заполнить хотя бы одно из полей.

# ПРИЛОЖЕНИЕ Д

(обязательное)

**Руководство пользователя**

Уданной программы не большая область применения, программа предназначена в основном для театров. В программе реализуется поиск по билетам, просмотр представлений, регистрация нового пользователя, заказ и отмена билетов. Администратор же может изменять типы билетов, цену, удалять пользователей и представления, добавлять пользователей и представление. Так же формировать отчеты за определенную дату. Чтобы использовать данную программу, пользователю нужно уметь пользоваться компьютером и мышью. Чтобы пользоваться данной программой достаточно прочитать Руководство пользователя.

Программа упрощает заказ билетов, формирование отчетов.

Условия при котором обеспечивается работа программы:

– *Windows* 10 с *.NET Framework* 6;

– объем оперативной памяти не менее 2гб;

– объем видеокарты должен быть не менее 128 Мбайт;

– объем свободного места на жестком диске должен быть не менее 20мб;

– разрешение экрана 800 x 600;

– установленная база данных *MySQL*.

Программы необходимые для работоспособности, сперва стоит загрузить *.NET Framework* 6, а потом установить базу данных *MySQL*.

Для того чтобы проверить работоспособность программы, пользователю достаточно запустить файл с расширением .exe. В программе необходимо правильно ввести данные и протестировать кнопки.

Если программа откажет в работе, ее нужно перезапустить, для этого можно нажать на крестик в правом верхнем углу или воспользоваться сочетанием клавиш *Alt + F4.*

Для использования данной программы необходимы небольшие знания в области работы с компьютером, необходимо прочитать руководство пользователя, а также убедиться, что конфигурация компьютера соответствует минимальным системным требованиям.

# ПРИЛОЖЕНИЕ Е

(обязательное)

**Иерархическая схема классов приложения**