МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ   
федеральное государственное автономное образовательное учреждение высшего образования «САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ АЭРОКОСМИЧЕСКОГО ПРИБОРОСТРОЕНИЯ»

КАФЕДРА КОМПЬЮТЕРНЫХ ТЕХНОЛОГИЙ И ПРОГРАММНОЙ ИНЖЕНЕРИИ

КУРСОВОЙ ПРОЕКТ  
ЗАЩИЩЕН С ОЦЕНКОЙ

РУКОВОДИТЕЛЬ

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Ст.преподаватель |  |  |  | Е.О. Шумова |
| должность, уч. степень, звание |  | подпись, дата |  | инициалы, фамилия |

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА  
К КУРСОВОМУ ПРОЕКТУ

Разработка приложения для организации взаимодействия объектов  
при заданных критериях

по дисциплине: ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

РАБОТУ ВЫПОЛНИЛ

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| СТУДЕНТ ГР. |  | Z1432K |  |  |  | Г. М. Штанчаев |
|  |  |  | подпись, дата |  | инициалы, фамилия |

Санкт-Петербург 2023

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ   
федеральное государственное автономное образовательное учреждение высшего образования «САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ АЭРОКОСМИЧЕСКОГО ПРИБОРОСТРОЕНИЯ»

КАФЕДРА КОМПЬЮТЕРНЫХ ТЕХНОЛОГИЙ И ПРОГРАММНОЙ ИНЖЕНЕРИИ

**Задание  
на курсовой проект по дисциплине  
«Объектно-ориентированное программирование»**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Студенту группы |  | Z1432K |  | Г. М. Штанчаев |
|  | № группы |  | инициалы, фамилия |

Тема «Разработка приложения для организации взаимодействия объектов  
при заданных критериях»

Исходные данные: Программа для обеспечения работы кинотеатра

Проект должен содержать:

* анализ предметной области
* разработку классов
* разработку тестового приложения
* оформление пояснительной записки по результатам выполнения проекта
* создание презентации к проекту

Срок сдачи законченного проекта \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Руководитель проекта \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ст.преп. Е.О.Шумова

Дата выдачи задания 01.09.2023 г.

# Содержание

[Содержание 3](#_Toc150693461)

[Введение 4](#_Toc150693462)

[1. Постановка задачи 5](#_Toc150693463)

[1.1. Анализ предметной области 5](#_Toc150693464)

[1.2. Формулировка функциональных требований 5](#_Toc150693465)

[2. Проектирование классов 7](#_Toc150693466)

[2.1. Классы сущностей 7](#_Toc150693467)

[2.2. Управляющие классы 18](#_Toc150693468)

[2.3. Используемые паттерны проектирования 20](#_Toc150693469)

[2.4. Диаграмма классов 21](#_Toc150693470)

# Введение

Основной задачей проекта является реализация приложения для обеспечения работы кинотеатра. Программа должна иметь интерфейс пользователя, позволяющий оператору кинотеатра управлять фильмами, сеансами, кинозалами, режиссерами, актерами, клиентами, продавать билеты и анализировать популярность фильмов. Также должна присутствовать возможность сохранения и загрузки всех данных из базы данных.

В цели проекта можно добавить изучение взаимодействия классов в объектно- ориентированном программировании и изучение паттернов проектирования. Основная часть пояснительной̆ записки содержит все необходимые разделы для описания процесса создания приложения.

В разделе «Постановка задачи» анализируется предметная область и формируется техническое.

В разделе «Проектирование классов» отображается процесс создания основной структуры программы, создание классов и их связей.

В разделе «Разработка приложения» в первую очередь подробно описывается пользовательский интерфейс, далее следует процесс разработки реализации классов и тестового приложения.

Раздел «Тестирование» содержит в себе демонстрацию работы описанного приложения в виде рисунков, содержащих скриншоты.

Раздел «Приложение» хранит полный код всех созданных программ.

# 1. Постановка задачи

## 1.1. Анализ предметной области

Предметной областью курсового проекта является программа для обеспечения работы кинотеатра. В данной предметной области необходимо разработать программу, позволяющую управлять фильмами и сеансами, продавать билеты, вести учет клиентов и статистику посещений сеансов. Программа должна иметь графический интерфейс и возможность хранить данные в базе.

Используемые сущности:

* Человек
  + Режиссер
  + Актер
  + Клиент
* Фильм
* Зал
* Билет
* Сеанс
* Кресло (в зале)
* Дата
* Время
* Статистика
* Сохранение и загрузка (для работы с БД)

## 1.2. Формулировка функциональных требований

Программа должна иметь графический интерфейс, предоставляющий оператору кинотеатра следующий функционал:

* Добавлять фильмы
* Добавлять сеансы
* Добавлять актеров и режиссеров
* Добавлять клиентов и управлять ценовой политикой для постоянных клиентов (выдавать скидки)
* Распределять места на сеансе
* Продавать и печатать билеты
* Смотреть статистику по фильмам и сеансам
* Отслеживать популярность фильмов определенных режиссеров
* Сохранять данные в базу и загружать из нее

Технические требования:

* Реализация на языке C++ с помощью среды Qt
* Графический интерфейс реализованный на фреймворке Qt

# 2. Проектирование классов

## 2.1. Классы сущностей

2.1.1. Класс «Hall»

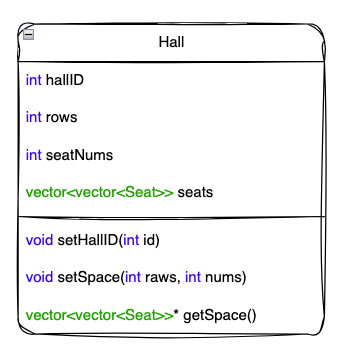


Рисунок . Класс "Hall"

Класс «Hall» описывает кинозал.

Описание полей:

* hallID – идентификатор (или номер) зала
* rows – количество рядов в зале
* seatNums – количество кресел в одном ряду
* seats – двумерный массив кресел

Описание методов:

* setHallID – сеттер для id
* setSpace – задает количество рядов и сидений в ряду
* getSpace() – возвращает двумерный массив кресел

2.1.2. Класс «Session»

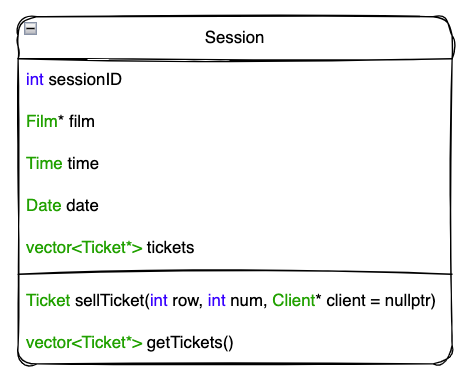


Рисунок . Класс "Session"

Класс «Session» описывает сущность сеанса. Этот класс наследуется от класса «Hall».

Описание полей:

* sessionID – идентификатор сеанса
* film – указать на фильм
* time – время начала сеанса
* date – дата проведения сеанса
* tickets – массив билетов на сеанс

Описание методов:

* sellTicket() – продажа билета (бронирует место на сеансе и прикрепляет билет к клиенту, если он зарегистрирован в базе)
* getTickets() – возвращает массив указателей на все билеты сеанса

2.1.3. Класс «Film»

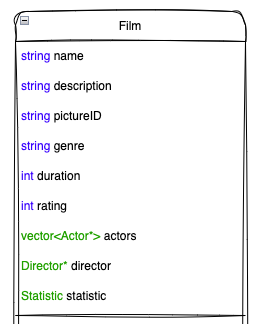
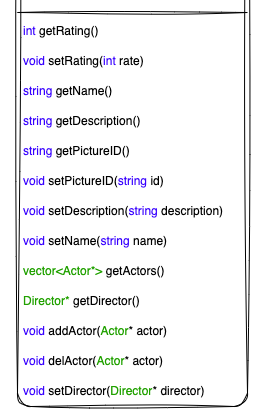
 

Рисунок . Класс "Film"

Класс «Film» описывает сущность фильма.

Описание полей:

* name – название фильма
* description – описание фильма
* pictureID – идентификатор изображения в хранилище
* genre – жанр
* duration – длительность в минутах
* rating – рейтинг фильма
* actors – указатели на актерев, играющих в фильме
* director – указатель на режисера фильма
* statistic – статистика по фильму

Описание методов:

* getRating() – выдает рейтинг фильма
* setRating() – задать рейтинг фильму
* getName() – получить название фильма
* setName() – задать название
* getPictureID() – получить идентификатор изображения
* setPictureID() – задать идентификатор
* getDescription() – получить описание
* setDescription() – задать описание
* setDirector() – задать режиссера фильма
* getDirector() – получить режиссера
* getActors() – возвращает массив указателей на актеров, играющих в фильме
* addActor() – добавляет актера
* dellActor() – удаляет актера из фильма

2.1.4. Класс «Seat»

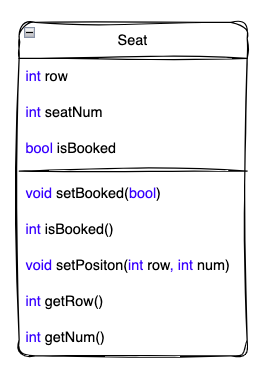


Рисунок . Класс "Seat"

Класс «Seat» описывает сущность кресла в кинозале.

Описание полей:

* row – ряд
* seatNum – номер кресла
* isBooked – занято ли место

Описание методов:

* setBooked() – задать занято или свободно
* isBooked() – возвращает true если занято, иначе - false
* setPosition() – задать ряд и номер кресла
* getRow() – получить ряд
* getNum() – получить номер места

2.1.5. Класс «Ticket»



Рисунок . Класс "Ticket"

Класс «Ticket» описывает сущность билета.

Описание полей:

* cost – стоимость билета
* session – указатель на сеанс
* seat – указатель на место (на сеансе)

Описание методов:

* setCost() – задать стоимость
* getCost() – получить стоимость
* setSession() – задать сеанс
* getSession() – получить сеанс
* setSeat() – задать кресло
* getSeat() – поучить кресло

2.1.6. Класс «Statistic»

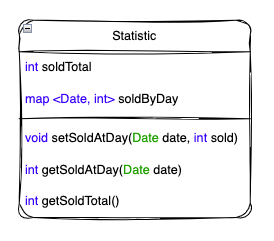


Рисунок . Класс "Statistic"

Класс «Statistic» описывает статистику.

Описание полей:

* soldTotal – продано всего
* soldByDay – продано по дням. Словарь, в котором ключ – дата, а значение – количество проданных билетов в эту дату

Описание методов:

* setSoldAtDay() – добавляет значение в словарь. Также увеличивает soldTotal
* getSoldAtDay() – получить количество проданных билетов в определенный день
* getSoldTotal() – получить количество проданных билетов за все время

2.1.7. Класс «Date»



Рисунок . Класс "Date"

Класс «Date» описывает дату.

Описание полей:

* day – день
* month – месяц
* year – год

Описание методов:

* setDate() – задать дату
* getDay() – получить день
* getMonth() – получить месяц
* getYear() – получить год
* getDate() - получить всю дату строкой в формате «ДД.ММ.ГГГГ»

2.1.7. Класс «Time»

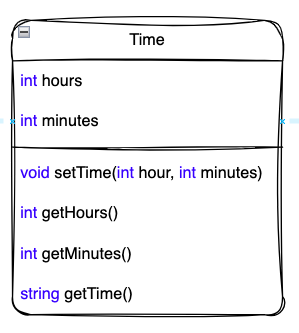


Рисунок . Класс "Time"

Класс «Time» описывает время.

Описание полей:

* hours – часы
* minutes – минуты

Описание методов:

* setTime() – задать время
* getHours() – получить часы
* getMinutes() – получить минуты
* getTime() – получить время строкой в формате «ЧЧ:ММ»

2.1.8. Класс «Client»

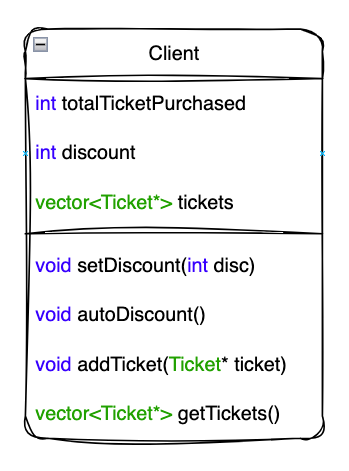


Рисунок . Класс "Client"

Класс «Client» описывает сущность клиента. Наследуется от класса «Human»

Описание полей:

* totalTicketPurchased – количество купленных билетов
* discount – скидка в процентах
* tickets – массив указателей на купленные билеты

Описание методов:

* setDiscount() – задать скидку
* autoDiscount() – рассчитать скидку автоматически (по количеству купленных билетов)
* addTicket() – добавить билет в список купленных данным клиентом
* getTickets() – выдает все билеты клиента

2.1.9. Класс «Actor»

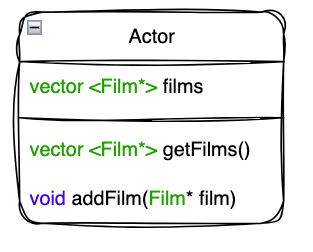


Рисунок . Класс "Actor"

Класс «Actor» описывает сущность актера. Наследуется от класса «Human»

Описание полей:

* films – массив указателей на фильмы, в которых играет актер

Описание методов:

* addFilm() – добавить фильм в список
* getFilms() – выдает список всех фильмов с этим актером

2.1.10. Класс «Director»

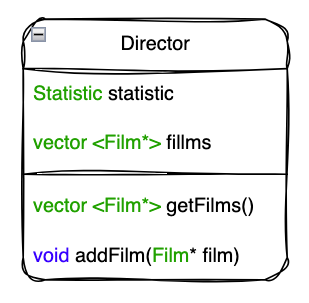


Рисунок . Класс "Director"

Класс «Director» описывает сущность режиссера. Наследуется от класса «Human»

Описание полей:

* films – массив указателей на фильмы, снятые этим режиссером
* statistic – статистика по всем фильмам этого режиссера

Описание методов:

* addFilm() – добавить фильм в список
* getFilms() – выдает список всех фильмов этого режиссера

2.1.11. Класс «Human»

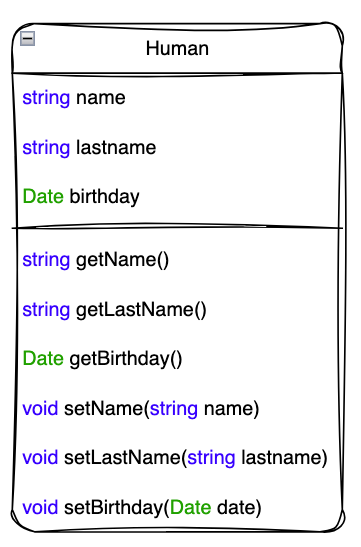


Рисунок . Класс "Human"

Класс «Human» описывает сущность человека.

Описание полей:

* name – имя
* lastname – фамилия
* birthday – дата рождения

Описание методов:

* getName() – получить имя
* getLastname() – получить фамилию
* getBirthday() – получить дату рождения
* setName() – задать имя
* setLastname() – задать фамилию
* setBirthday() – задать день рождения

## 2.2. Управляющие классы

2.2.1. Класс «SaveAndLoad»

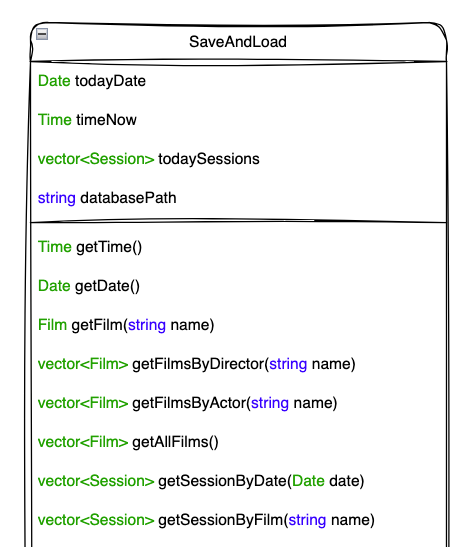


Рисунок . Класс "SaveAndLoad" (1)

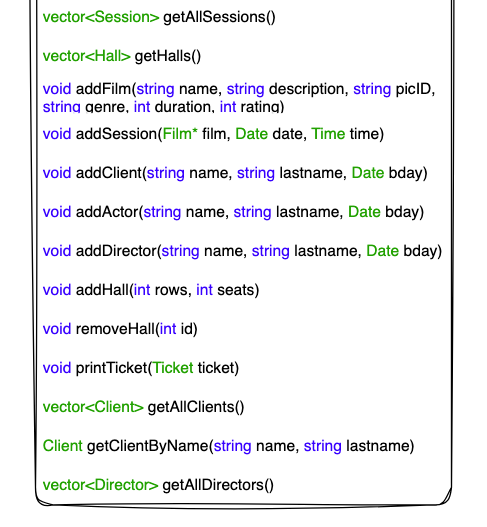


Рисунок . Класс "SaveAndLoad" (2)

Класс «SaveAndLoad» является управляющим классом, реализующим удобный API для работы с базой(ами) данных и хранилищем.

Описание полей:

* todayDate – сегодняшняя дата
* timeNow – время сейчас
* todaySessions – массив сегодняшних сеансов
* databasePath – путь к базе данных

Описание методов:

* getTime() – получить время (сейчас)
* getDate() – получить сегодняшнюю дату
* getFilm() – получить фильм по названию
* getFilmsByDirector() – получить фильмы по режиссеру
* getFilmsByActor() – получить фильмы с участием актера
* getAllFilms() – получить все фильмы
* getSessionByDate() – получить сеансы по дате
* getSessionsByFilm() – получить сеансы по фильму
* getAllSessions() – получить список всех сеансов
* getHalls() – получить
* addFilm() – добавить фильм
* addSession() – добавить сеанс
* addClient() – добавить клиента
* addActor() – добавить актера
* addDirector() – добавить режиссера
* addHall() – добавить зал
* removeHall() – удалить зал
* printTicket() – распечатать билет (для выдачи клиенту)
* getAllClients() – получить список всех клиентов
* getClientByName() – получить клиента по имени
* getAllDirectors() – получить список всех режиссеров

## 2.3. Используемые паттерны проектирования

2.3.1. Singleton

Реализован в классе «SaveAndLoad». Данный класс в программе глобален и должен существовать в одном экземпляре, поэтому паттерн «одиночка» хорошо подходит. Цель использования данного паттерна в том, чтобы реализовать интерфейс сохранения и загрузки данных, абстрагируя логику работы программы, от работы с базой данных. Такой подход позволит масштабировать программу. Например, в будущем, при расширении сети кинотеатров, можно вынести базу в облако чтобы все кинотеатры работали с одной БД. А в программе нам нужно будет поправить только реализацию методов данного класса.

## 2.4. Диаграмма классов

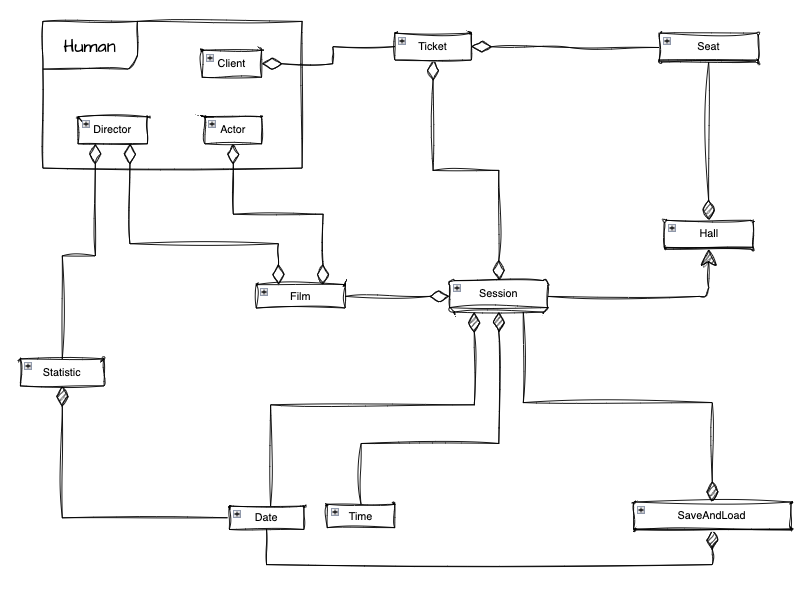


Рисунок . Диаграмма классов

# 3. Разработка приложения

## 3.1. Разработка интерфейса приложения

3.1.1. Главное меню программы

Главное меню программы содержит кнопки (QPushButton):

* Сеансы сегодня
* Фильмы сегодня
* Сеансы
* Фильмы
* Клиенты
* Актеры
* Режиссеры
* Кинозалы

Нажатие на каждую кнопку открывает интерфейс управления соответствующим разделом.

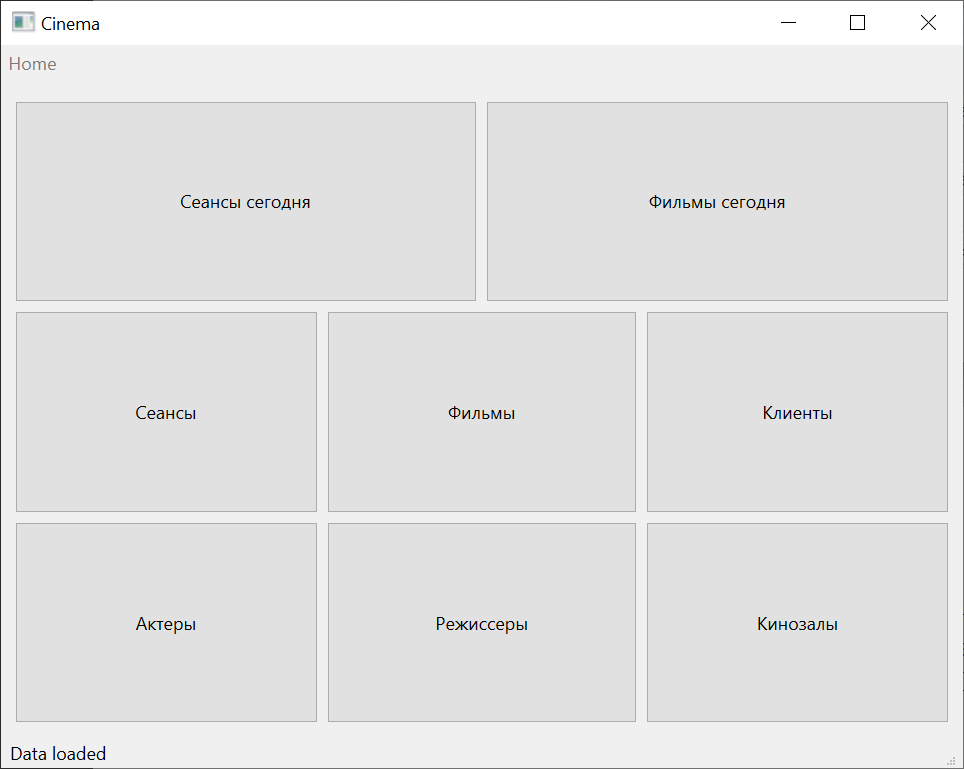


Рисунок . Главное меню программы

В дизайнере это окно спроектировано следующим образом:

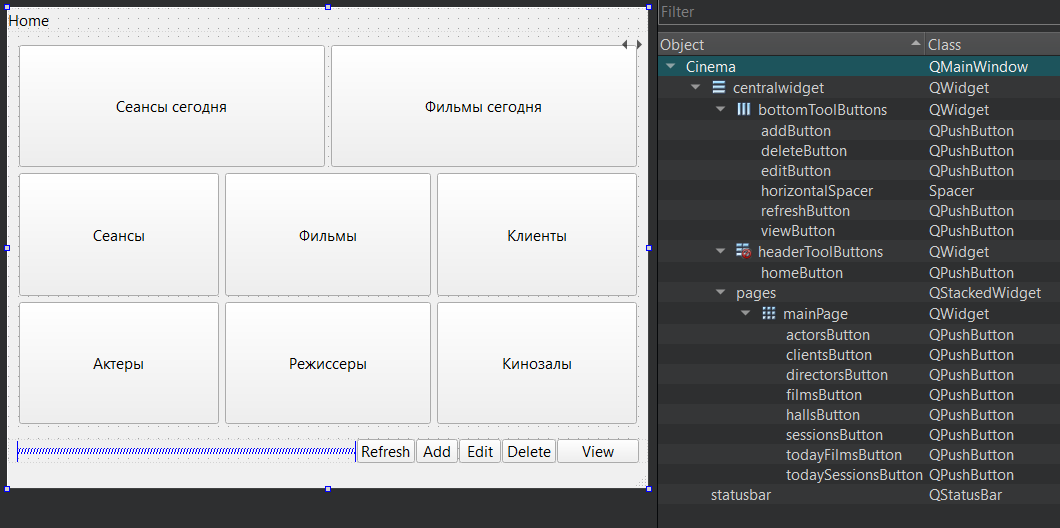


Рисунок . Главное меню программы [Design]

Есть блок *pages* (QStackedWidget), который содержит страницу с главным меню (mainPage). При нажатии на кнопку с разделом динамически создается новая страница, добавляется в *pages* и делается отображаемой.

Также есть кнопки в нижней панели (*bottomToolButtons*). Данная панель скрыта на главной странице, но отображается во всех разделах.

Кнопка *Home* в верхней позволяет переместиться с открытого раздела на главную страницу.

3.1.2. Страницы разделов

Страницы разделов создаются динамически. Верстка описывается только в коде, без использования среды дизайнера. Сделаны они по похожему принципу, отличается только содержание виджетов.

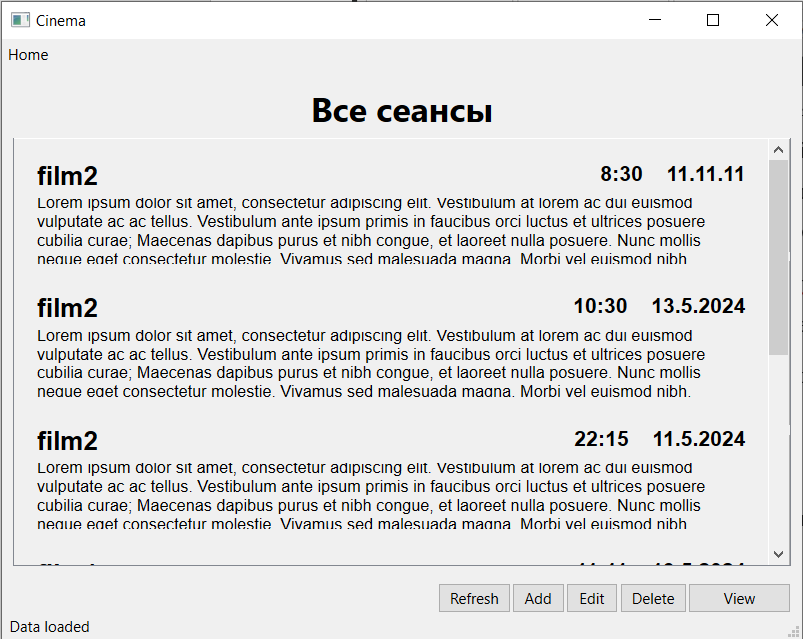


Рисунок . Страница сеансов

Для генерации страницы создан отдельный класс SessionsPage, который наследуется от QWidget. В данном классе реализован метод (в виде публичного слота) render\_page, который принимает в качестве параметров вектор сессий и строку с заголовком страницы

void **SessionsPage**::**render\_page**(const **vector**<**Session**\*>& sessions, **QString** title)

{

Сначала страница очищается от всего что уже есть в ней. Это нужно в случаях, когда страница уже создана, но были внесены изменения и необходимо ее обновить (например при нажатии кнопки Refresh или после удаления элемента кнопкой Delete).

Также создается Layout для равномерного заполнения страницы и добавляется заголовок, если его передали в параметрах.

*// Очистка страницы*

    for (auto child : this->**children**()) {

        delete child;

    }

    sessionsPageLayout = new **QVBoxLayout**(this);

*// Заголовок если есть*

    if (title != "") {

        sessionsLabel = new **QLabel**(title, this);

        sessionsLabel->**setFont**(**QFont**("Segoe UI", 20, **QFont**::Bold));

        sessionsLabel->**setAlignment**(**Qt**::**Alignment**(**Qt**::AlignCenter));

        sessionsPageLayout->**addWidget**(sessionsLabel);

    }

Далее создается QScrollArea и в ней QVBoxLayout. Также создается QButtonGroup. Все виджеты с сеансами реализованы в виде QPushButton (с включенным checkable) и добавляются в QButtonGroup. Таким образом пользователь в списке сеансов может выбрать один из них и производить с ним действия, которые реализованы кнопками в нижней панели.

*// Scroll Area*

    sessionsScrollArea = new **QScrollArea**(this);

    sessionsScrollArea->**setWidgetResizable**(true);

    sessionsScrollAreaWidget = new **QWidget**();

    sessionsScrollArea->**setWidget**(sessionsScrollAreaWidget);

    sessionsPageLayout->**addWidget**(sessionsScrollArea);

*// Layout для равномерного заполнения*

    sessionsScrollAreaLayout = new **QVBoxLayout**(sessionsScrollAreaWidget);

    sessionsButtonsGroup = new **QButtonGroup**(this);

Далее создаются виджеты всех сеансов.

    for (auto session : sessions) {

*// Создание виджетов и добавление их в ScrollArea*

*// session Widget*

        QPushButton \*sessionWidget = new **QPushButton**();

        sessionWidget->**setFixedHeight**(100);

        sessionWidget->**setFlat**(true);

        sessionWidget->**setCheckable**(true);

        QGridLayout\* sessionLayout = new **QGridLayout**(sessionWidget);

*// session name*

        QLabel\* sessionNameLabel = new **QLabel**(**QString**::**fromStdString**(session->**getFilm**()->**getName**()));

        sessionNameLabel->**setFont**(**QFont**("Arial", 16, **QFont**::Bold));

*// session description*

        QLabel\* sessionDescLabel = new **QLabel**(**QString**::**fromStdString**(session->**getFilm**()->**getDescription**()));

        sessionDescLabel->**setFont**(**QFont**("Arial", 10));

        sessionDescLabel->**setMaximumWidth**(860);

        sessionDescLabel->**setWordWrap**(true);

*// session date and time*

        QLabel\* sessionDateTimeLabel = new **QLabel**((session->**getTime**().**getTime**() + "    " + session->**getDate**().**getDate**()).**c\_str**());

        sessionDateTimeLabel->**setFont**(**QFont**("Arial", 13, **QFont**::Bold));

        sessionDateTimeLabel->**setAlignment**(**Qt**::**Alignment**(**Qt**::AlignRight));

*// sessionDateTimeLabel->setStyleSheet("QLabel { color : green; }");*

*// Компановка*

        sessionLayout->**addWidget**(sessionNameLabel, 1, 1);

        sessionLayout->**addWidget**(sessionDescLabel, 2, 1, 2, 3);

        sessionLayout->**addWidget**(sessionDateTimeLabel, 1, 3);

        sessionsScrollAreaLayout->**addWidget**(sessionWidget);

        sessionsButtonsGroup->**addButton**(sessionWidget, session->**getId**());

}

В конце страницы добавляется QSpacerItem, чтобы виджеты сеансов не растягивались.

*// Spacer для заполнения нижней части, чтоб виджеты фильмов не растягивались на всю высоту*

    sessionsScrollSpaser = new **QSpacerItem**(100, 20, **QSizePolicy**::Minimum, **QSizePolicy**::Expanding);

    sessionsScrollAreaLayout->**addItem**(sessionsScrollSpaser);

}

Таким же образом реализованы страницы других разделов, меняется только содержание самих виджетов элементов.

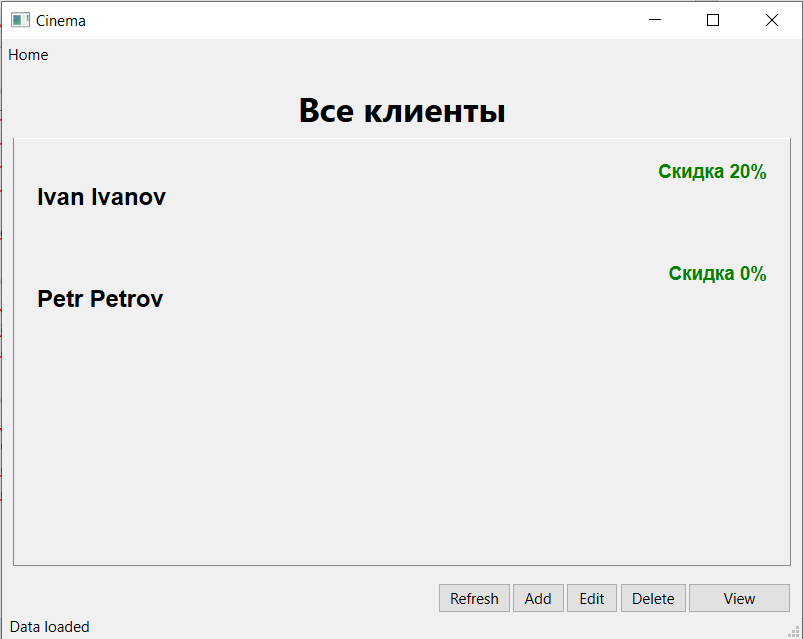


Рисунок . Страница клиентов

3.1.3. Страница сеанса

В разделе сеансов при нажатии на кнопку View открывается модальное окно с выбранным сеансом в режиме просмотра.

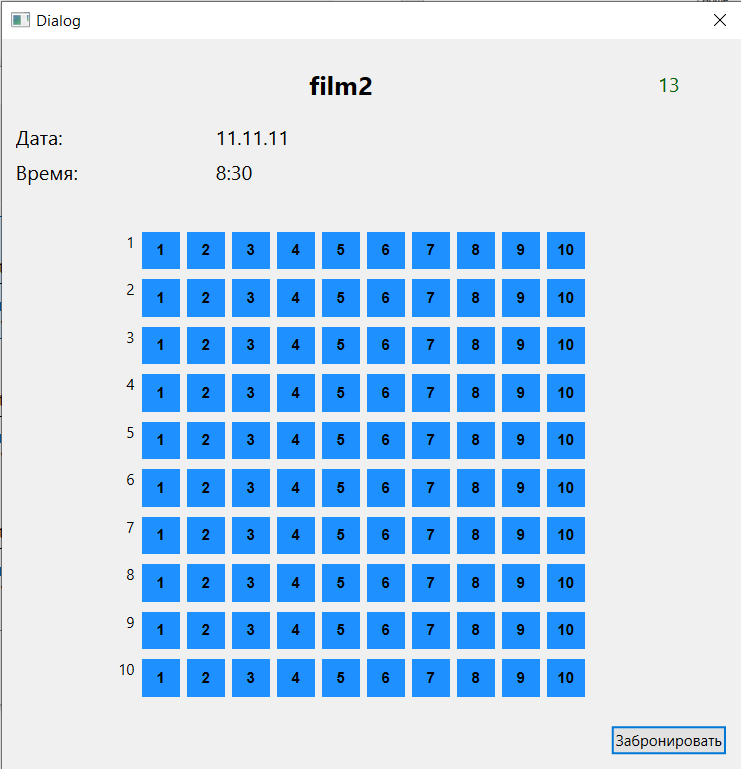


Рисунок . Просмотр сеанса

В нем можно выбрать место в ряду и забронировать его. При нажатии на кнопку открывается модальное окно, в которое можно вписать id клиента, который покупает билет.

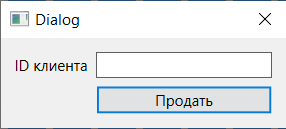


Рисунок . Выбор клиента

Забронированное место в зале отображается серым цветом и становится неактивным (его нельзя выбрать).

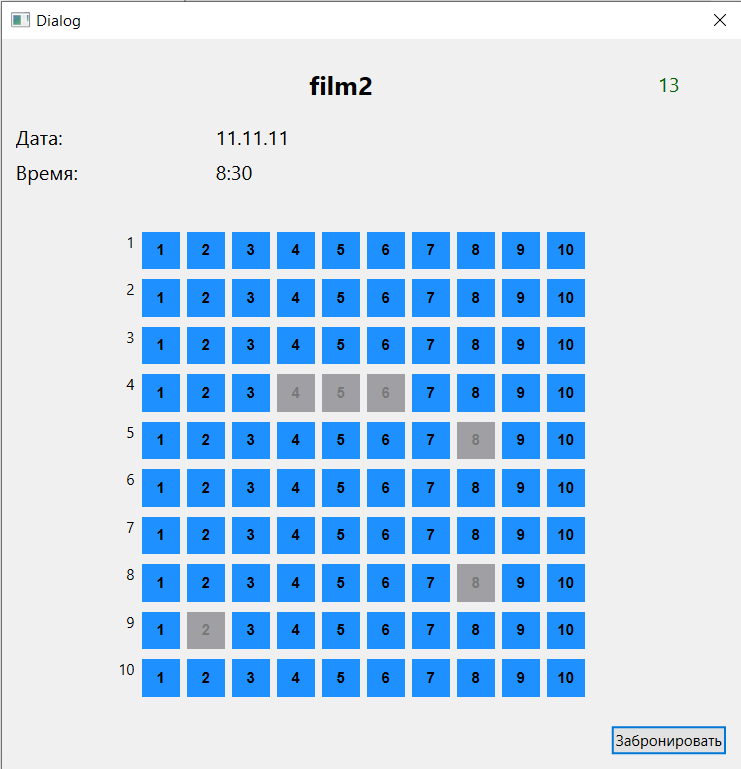


Рисунок . Забронированные места

Интерфейс данного окна частично реализован с помощью среды дизайнера.

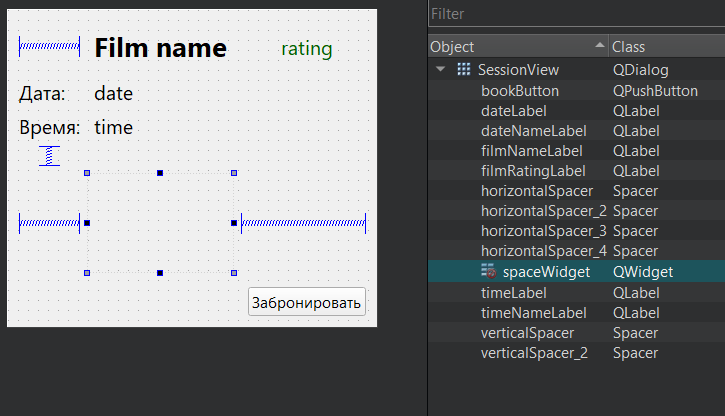


Рисунок . Окно сеанса [Design]

Текстовые поля (QLabel) меняются при создании страницы. А spaceWidget заполняется местами, которые генерируются динамически.

Рендер мест в окне реализован с помощью следующего слота

void **SessionView**::**render\_space**()

{

    spaceWidget = ui->spaceWidget;

*// Очистка страницы*

    for (auto child : spaceWidget->**children**()) {

        delete child;

    }

    spaceWidget->**setMinimumHeight**(session->**getSpace**()->**size**() \* 40);

    spaceLayout = new **QGridLayout**(spaceWidget);

    spaceButtonGroup = new **QButtonGroup**(spaceWidget);

    int row\_i = 1;

    for (auto row : \*session->**getSpace**()) {

        QLabel\* rowLabel = new **QLabel**(**QString**("%1").**arg**(row\_i));

        rowLabel->**setAlignment**(**Qt**::**Alignment**(**Qt**::AlignVCenter));

        rowLabel->**setAlignment**(**Qt**::**Alignment**(**Qt**::AlignRight));

        rowLabel->**setFixedSize**(30, 30);

        spaceLayout->**addWidget**(rowLabel, row\_i, 1);

        row\_i++;

        for (auto seat : row) {

            QPushButton\* seatButton = new **QPushButton**();

            seatButton->**setFixedSize**(30, 30);

            seatButton->**setCheckable**(true);

            seatButton->**setFlat**(true);

*// free - blue, booked - gray*

            QPalette palette = seatButton->**palette**();

            palette.**setColor**(seatButton->**backgroundRole**(), **qRgb**(30, 144, 255));

            if (seat->**isBooked**()) {

                palette.**setColor**(seatButton->**backgroundRole**(), **Qt**::gray);

                seatButton->**setDisabled**(true);

            }

            seatButton->**setAutoFillBackground**(true);

            seatButton->**setPalette**(palette);

            QVBoxLayout\* seatButtonLayout = new **QVBoxLayout**(seatButton);

            QLabel\* seatButtonLabel = new **QLabel**(**QString**("%1").**arg**(seat->**getNum**()));

            seatButtonLabel->**setFont**(**QFont**("Arial", 9, **QFont**::Bold));

            seatButtonLabel->**setAlignment**(**Qt**::**Alignment**(**Qt**::AlignCenter));

            seatButtonLayout->**addWidget**(seatButtonLabel);

            seatButtonLayout->**setContentsMargins**(0,0,0,0);

            spaceLayout->**addWidget**(seatButton, seat->**getRow**(), seat->**getNum**()+1);

            spaceButtonGroup->**addButton**(seatButton);

        }

    }

}

Окно, которое открывается при продаже билета реализовано следующим образом

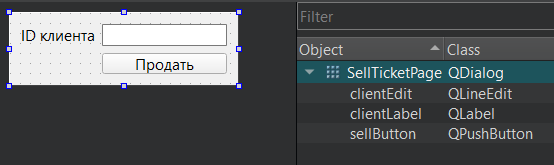


Рисунок . Окно продажи билета [Design]

Для кнопки *Продать* реализован слот, который регистрирует продажу в системе (с помощью класса-одиночки SaveAndLoad) и отправляет сигнал *sold()*

void **SellTicketPage**::**on\_sellButton\_clicked**()

{

    Client\* client = nullptr;

    if  (!ui->clientEdit->**text**().**isEmpty**()) {

        int clientId = ui->clientEdit->**text**().**toInt**();

        client = sal.**getClientById**(clientId);

    }

    sal.**sellTicket**(session, row, num, client);

    emit **sold**();

    this->**close**();

}

Это окно создается в слоте предыдущего класса (который описывает окно просмотра сеанса) при нажатии на кнопку *Забронировать*. Также тут соединяется сигнал sold() и слот render\_space(), для того, чтобы после продажи билета места, страница обновилась и места на ней стали неактивными.

void **SessionView**::**on\_bookButton\_clicked**()

{

*// calc row and num from index*

    int index = -spaceButtonGroup->**checkedId**() - 1;

    if (index > 0) {

        int row = (index - 1) / session->**getHall**()->**getNums**() + 1;

        int num = index - (row - 1) \* session->**getHall**()->**getNums**();

        sellTicketPage = new **SellTicketPage**(session, row, num);

**connect**(sellTicketPage, **SIGNAL**(**sold**()), this,  **SLOT**(**render\_space**()));

        sellTicketPage->**setModal**(true);

        sellTicketPage->**show**();

        this->**render\_space**();

    }

}

3.1.4. Страницы просмотра разделов

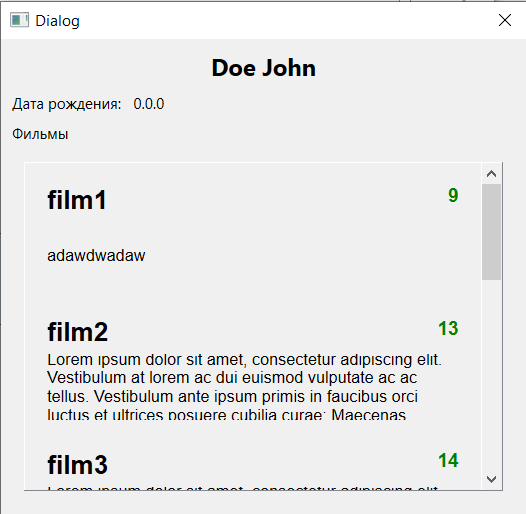


Рисунок . Страница просмотра актера

Страница просмотра актера реализована следующим образом

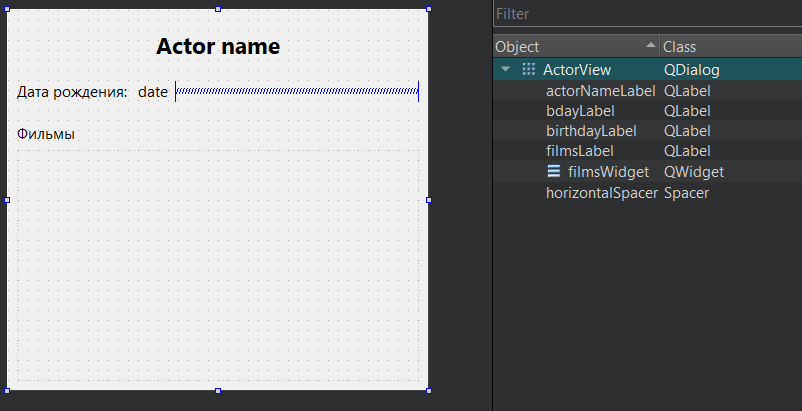


Рисунок . Страница просмотра актера [Design]

Текстовые поля (QLabel) изменяются при создании страницы. А в виджет с фильмами (filmsWidget) добавляется страница со списком фильмов, в которых играл этот актер (о том, как устроена эта страница см 3.1.2.).

Также и для режиссера

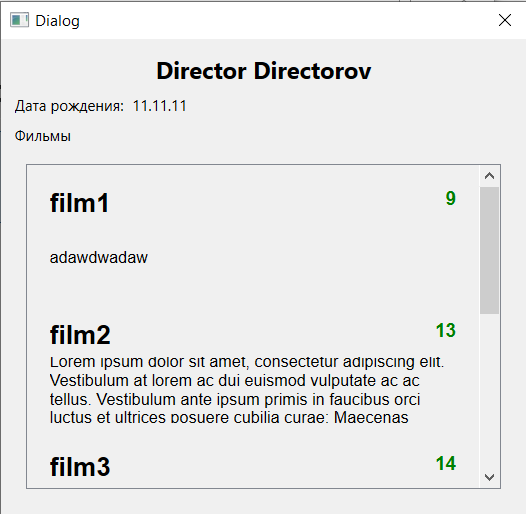


Рисунок . Страница просмотра режиссера

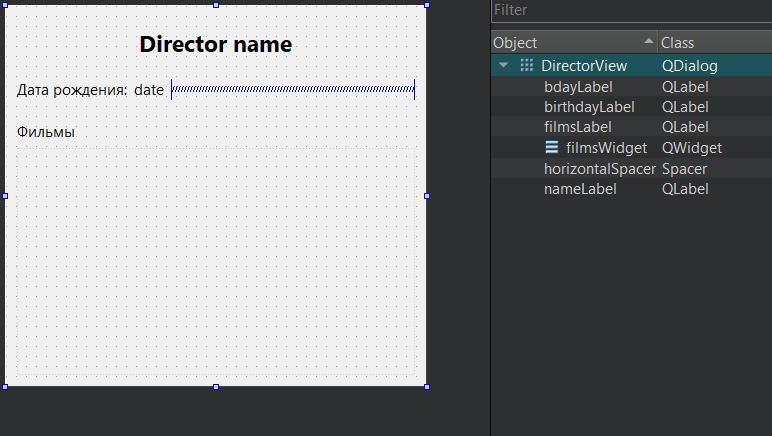


Рисунок . Страница просмотра режиссера [Design]

На странице просмотра фильмов отображается сеансы и актеры.

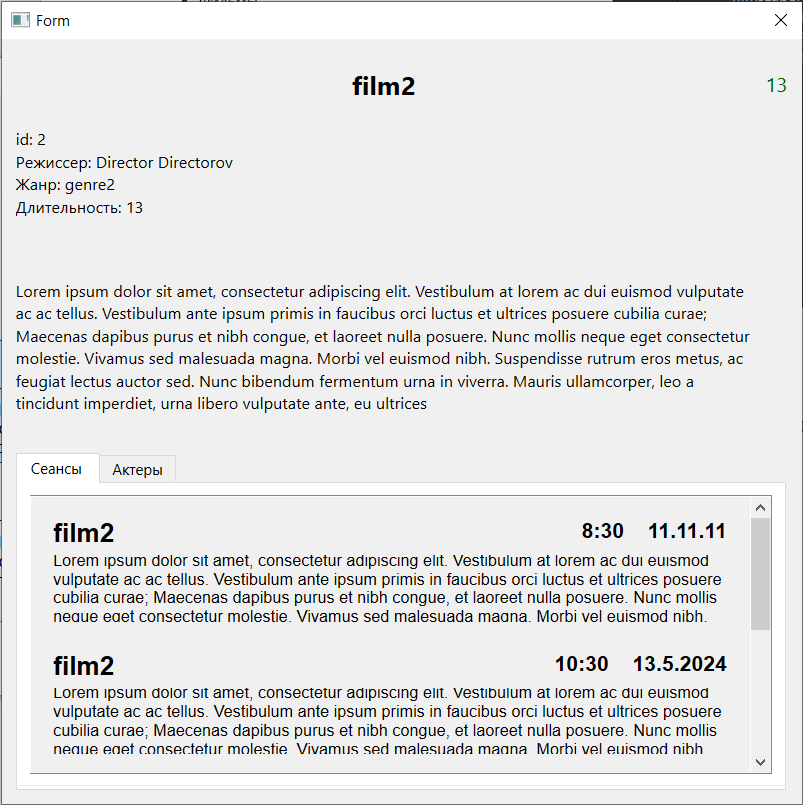


Рисунок . Страница просмотра фильма (сеансы)



Рисунок . Страница просмотра фильма (актеры)

Реализовано это следующим образом

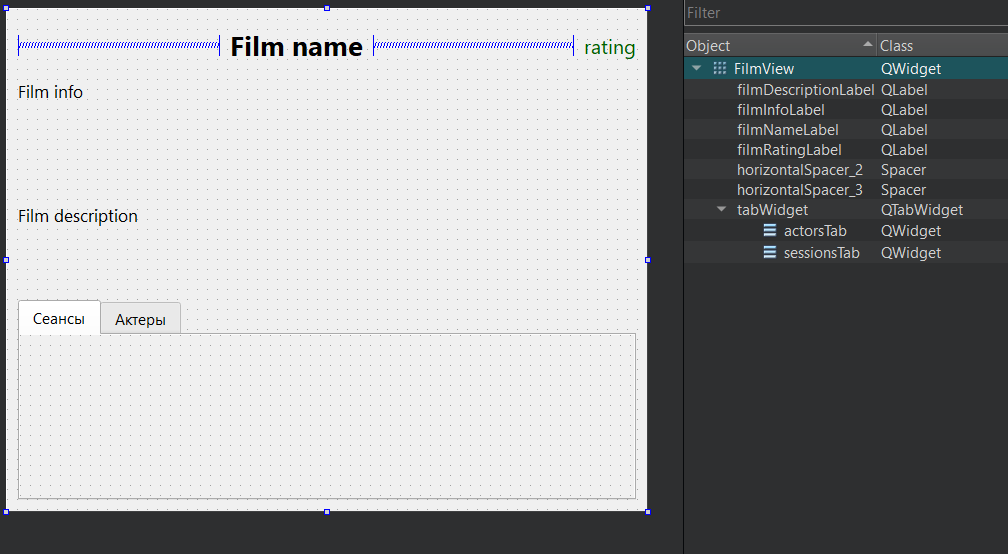


Рисунок . Страница просмотра фильма [Design]

В QTabWidget есть две вкладки – с актерами и сеансами. В них добавляются страницы со списком соответствующих элементов (см 3.1.2.).

3.1.5. Страницы редактирования

При нажатии на кнопку Edit открывается редактирование выбранного элемента.

Для сеанса это выглядит так

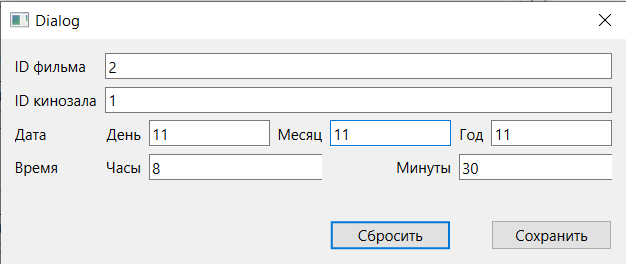


Рисунок . Редактирование сеанса

Для фильма

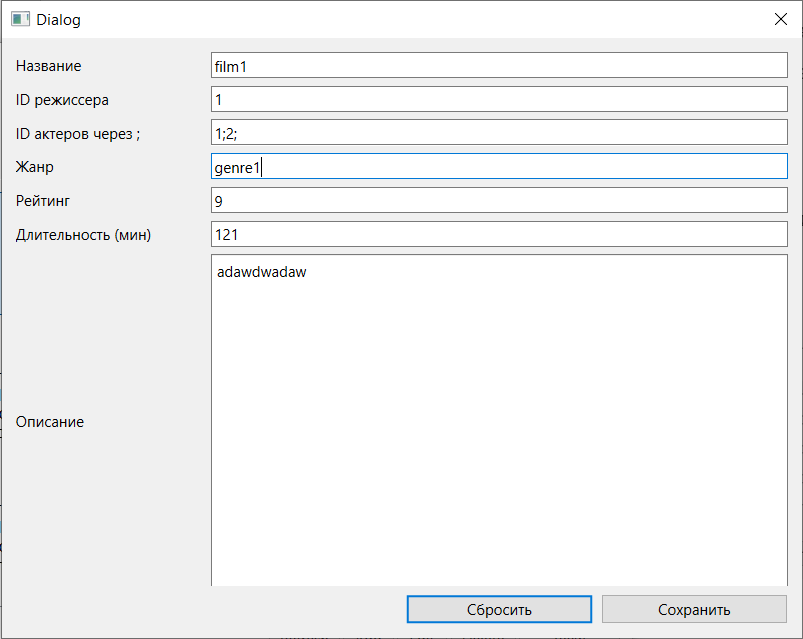


Рисунок . Редактирование фильма

Для актера и режиссера

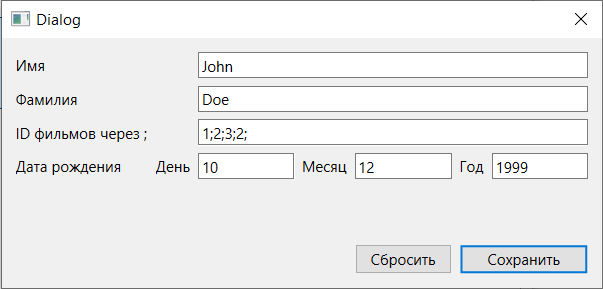


Рисунок . Редактирование актера

Для клиента

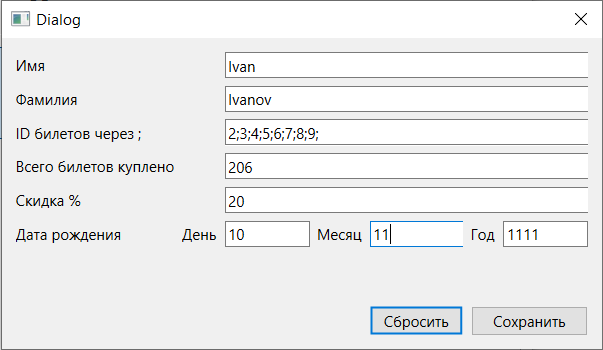


Рисунок . Редактирование клиента

Для кинозала

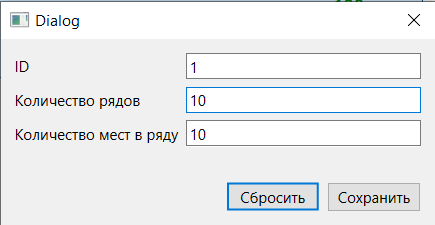


Рисунок . Редактирование кинозала

Реализованы они следующим образом

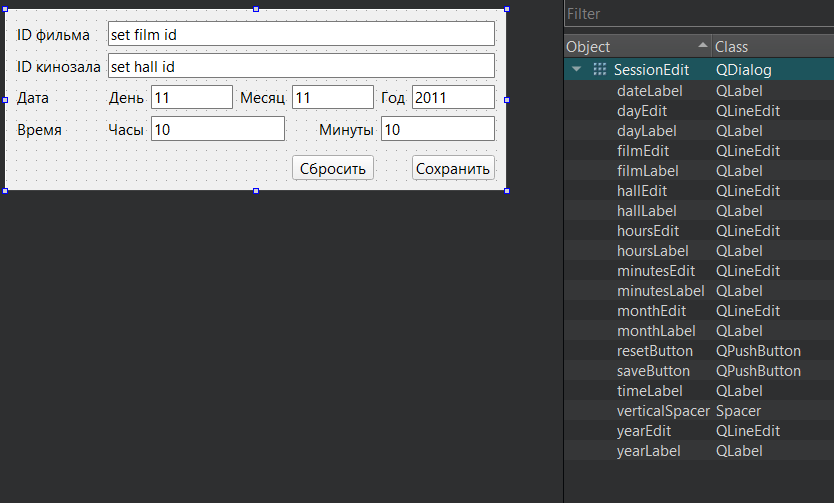


Рисунок . Редактирование сеанса [Design]

Для кнопки *Сбросить* реализован следующий слот

void **SessionEdit**::**on\_resetButton\_clicked**()

{

    if (session->**getFilm**() != nullptr)

        ui->filmEdit->**setText**(**QString**("%1").**arg**(session->**getFilm**()->**getId**()));

    else

        ui->filmEdit->**setText**("set film id");

    if (session->**getHall**() != nullptr)

        ui->hallEdit->**setText**(**QString**("%1").**arg**(session->**getHall**()->**getId**()));

    else

        ui->hallEdit->**setText**("set hall id");

    ui->dayEdit->**setText**(**QString**("%1").**arg**(session->**getDate**().**getDay**()));

    ui->monthEdit->**setText**(**QString**("%1").**arg**(session->**getDate**().**getMonth**()));

    ui->yearEdit->**setText**(**QString**("%1").**arg**(session->**getDate**().**getYear**()));

    ui->hoursEdit->**setText**(**QString**("%1").**arg**(session->**getTime**().**getHours**()));

    ui->minutesEdit->**setText**(**QString**("%1").**arg**(session->**getTime**().**getMinutes**()));

}

Он же вызывается при создании страницы для заполнения полей.

Для кнопки *Сохранить* следующий слот

void **SessionEdit**::**on\_saveButton\_clicked**()

{

    this->session->**setHall**(sal.**getHallById**(ui->hallEdit->**text**().**toInt**()));

    this->session->**setFilm**(sal.**getFilmById**(ui->filmEdit->**text**().**toInt**()));

    this->session->**setDate**(**Date**(ui->dayEdit->**text**().**toInt**(),

                                ui->monthEdit->**text**().**toInt**(),

                                ui->yearEdit->**text**().**toInt**()));

    this->session->**setTime**(**Time**(ui->hoursEdit->**text**().**toInt**(),

                                ui->minutesEdit->**text**().**toInt**()));

}

Таким же образом реализованы и другие окна редактирования объектов.

Добавление новых объектов (кнопка Add) тоже использует эти окна, только вместо существующего объекта, в редактирование подставляется новый «пустой» объект, который в результате сохраняется в системе.

## 3.2. Реализация классов

3.2.1. Реализация классов сущностей

Для каждого класса сущности определены конструкторы, гетеры и сеттеры. Помимо них, для некоторых классов было необходимо реализовать некоторые функциональные методы:

* Для класса Film помимо добавления и удаления актера из вектора, есть метод очищающий вектор полностью – clearActors()

void **Film**::**addActor**(**Actor**\* actor) {

    if (actor) {

        this->actors.**push\_back**(actor);

    }

}

void **Film**::**clearActors**() {

    this->actors.**clear**();

}

void **Film**::**delActor**(**Actor**\* actor) {

**vector**<**Actor**\*>::iterator f;

    f = **std**::**find**(this->actors.**begin**(), this->actors.**end**(), actor);

    if (f != this->actors.**end**()) {

        this->actors.**erase**(f);

    }

}

* Также для классов Director и Actor есть метод очищающий вектор фильмов

void **clearFilms**()

{

    this->films.**clear**();

}

Необходимость очистить вектор возникает, когда пользователь в графическом интерфейсе переопределяет список фильмов у актера или режиссера или список актеров в фильме.

* У класса Client помимо очищения вектора билетов, определен также параметр autoDiscount, который пересчитывает скидку на основе количества купленных билетов

void **Client**::**autoDiscount**() {

    if (this->totalTicketPurchased >= 10)   **setDiscount**(5);

    if (this->totalTicketPurchased >= 20)   **setDiscount**(10);

    if (this->totalTicketPurchased >= 50)   **setDiscount**(15);

    if (this->totalTicketPurchased >= 100)  **setDiscount**(20);

    if (this->totalTicketPurchased >= 250)  **setDiscount**(30);

    if (this->totalTicketPurchased >= 500)  **setDiscount**(40);

    if (this->totalTicketPurchased >= 1000) **setDiscount**(50);

}

* Для класса Date переопределен оператор сравнения «==» для облегчения поиска сеансов в определенную дату

bool **Date**::operator==(**Date**& date) {

    return this->**getDay**() == date.**getDay**()

           and this->**getMonth**() == date.**getMonth**()

           and this->**getYear**() == date.**getYear**();

}

* Для класса Session реализован метод, который инициализирует «пространство» в зале. Пространство у сеанса – это двумерный вектор сидений (Seat). У каждого сидения есть состояние – занято/свободно. При инициализации создаются объекты сидений и им проставляется статус «свободно»

void **Session**::**\_initSpace**() {

    for (int row = 1; row <= hall->**getRows**(); row++) {

**std**::vector<Seat\*> r;

        for (int num = 1; num <= hall->**getNums**(); num++) {

            Seat \*seat = new **Seat**(row, num, false);

            r.**push\_back**(seat);

        }

        this->space.**push\_back**(r);

    }

}

А также метод для бронирования места

void **Session**::**bookSeat**(int row, int num, **Ticket**\* ticket)

{

    this->**getSeatByNum**(row, num)->**setBooked**(true);

    if (ticket) {

        this->tickets.**push\_back**(ticket);

    }

}

3.2.2 Реализация управляющих классов

В программе предусмотрен только один управляющий класс – SaveAndLoad. Это класс одиночка, который реализует удобный API для работы с базой данных и управляет всеми объектами в системе.

Так как этот класс используется для реализации паттерна Singleton, у него удалены конструкторы и деструкторы:

**SaveAndLoad**() = default;

**~SaveAndLoad**() = default;

**SaveAndLoad**(const **SaveAndLoad**&) = delete;

**SaveAndLoad**&    operator =(const **SaveAndLoad**&)  = delete;

    void\*           operator new(**std**::size\_t)       = delete;

    void\*           operator new[](**std**::size\_t)     = delete;

    void            operator delete(void\*)          = delete;

    void            operator delete[](void\*)        = delete;

И реализован метод init(), который создает и возвращает статический объект этого класса. Тем самым гарантируя, что объект в программе будет только один

**SaveAndLoad**& **SaveAndLoad**::**init**() {

    static SaveAndLoad obj;

    return obj;

}

Также реализованы методы загрузки и сохранения данных в БД

    bool **loadData**();

    bool **loadFilms**();

    bool **loadActors**();

    bool **loadHalls**();

    bool **loadClients**();

    bool **loadSessions**();

    bool **loadTickets**();

    bool **loadDirectors**();

    bool **loadFilmsActors**();

    bool **saveData**();

    bool **saveFilms**();

    bool **saveActors**();

    bool **saveHalls**();

    bool **saveClients**();

    bool **saveSessions**();

    bool **saveTickets**();

    bool **saveDirectors**();

    bool **saveFilmsActors**();

loadData() и saveData() вызывают поочередно методы для загрузки/сохранения сущностей.

Методы сохранения данных реализованы одинаково, отличаются лишь поля

bool **SaveAndLoad**::**saveFilms**() {

    QSqlQuery query = **QSqlQuery**(this->**getDB**());

    query.**exec**("DELETE FROM films");

    query.**prepare**("INSERT INTO films (id, name, description, genre, duration, rating, director) "

               "VALUES (:id, :name, :description, :genre, :duration, :rating, :director)");

    for (auto film : this->films){

        query.**bindValue**(":id", film->**getId**());

        query.**bindValue**(":name", film->**getName**().**c\_str**());

        query.**bindValue**(":description", film->**getDescription**().**c\_str**());

        query.**bindValue**(":genre", film->**getGenre**().**c\_str**());

        query.**bindValue**(":duration", film->**getDuration**());

        query.**bindValue**(":rating", film->**getRating**());

        if (film->**getDirector**())

            query.**bindValue**(":director", film->**getDirector**()->**getId**());

        query.**exec**();

    }

    return true;

}

bool **SaveAndLoad**::**loadFilms**() {

    QSqlQuery query = **QSqlQuery**(this->**getDB**());

    query.**exec**("SELECT \* FROM films");

    while (query.**next**()) {

        int id              = query.**record**().**field**("id").**value**().**toInt**();

        string name         = query.**record**().**field**("name").**value**().**toString**().**toStdString**();

        string description  = query.**record**().**field**("description").**value**().**toString**().**toStdString**();

        string genre        = query.**record**().**field**("genre").**value**().**toString**().**toStdString**();

        int duration        = query.**record**().**field**("duration").**value**().**toInt**();

        int rating          = query.**record**().**field**("rating").**value**().**toInt**();

        int directorId      = query.**record**().**field**("director").**value**().**toInt**();

        Director\* director = this->**getDirectorById**(directorId);

        this->**addFilm**(name, description, genre, duration, rating, director, id);

    }

    return true;

}

Далее реализованы гетеры для сущностей

**vector**<**Film**\*>       **getFilmsByDirector**(**string** name);

**vector**<**Film**\*>       **getFilmsByActor**(**string** name);

**vector**<**Film**\*>       **getFilmsByDate**(**Date** date);

**vector**<**Film**\*>       **getAllFilms**();

**vector**<**Session**\*>    **getSessionByDate**(**Date** date);

**vector**<**Session**\*>    **getSessionByFilm**(**string** name);

**vector**<**Session**\*>    **getSessionByFilm**(**Film**\* film);

**vector**<**Session**\*>    **getAllSessions**();

**vector**<**Hall**\*>       **getHalls**();

**vector**<**Client**\*>     **getAllClients**();

**vector**<**Ticket**\*>     **getAllTickets**();

**Client**\*             **getClientByName**(**string** name, **string** lastname);

**vector**<**Director**\*>   **getAllDirectors**();

**vector**<**Actor**\*>      **getAllActors**();

**Film**\*               **getFilmById**(int id);

**Session**\*            **getSessionById**(int id);

**Client**\*             **getClientById**(int id);

**Hall**\*               **getHallById**(int id);

**Ticket**\*             **getTicketById**(int id);

**Director**\*           **getDirectorById**(int id);

**Actor**\*              **getActorById**(int id);

Получение списка объектов, по какому-то фильтру (например сеансов по дате) происходит реализовано следующим образом:

**vector**<**Session**\*> **SaveAndLoad**::**getSessionByDate**(**Date** date) {

    vector<Session\*> sessions;

    for (auto session : this->**getAllSessions**()) {

        if (session->**getDate**() == date)

            sessions.**push\_back**(session);

    }

    return sessions;

}

Получение единичных объектов примерно также:

**Client**\* **SaveAndLoad**::**getClientByName**(**string** name, **string** lastname) {

    for (auto client : this->**getAllClients**()) {

        if (client->**getName**() == name and client->**getLastname**() == lastname)

            return client;

    }

    return nullptr;

}

Далее методы для добавления и удаления объектов

**Film**\* **addFilm**(**string** name, **string** description, **string** genre, int duration,

int rating, **Director**\* director = nullptr, int id = -1);

**Session**\* **addSession**(**Film**\* film, **Hall**\* hall, **Time** time, **Date** date, int id=-1);

**Client**\* **addClient**(**string** name, **string** lastname, **Date** bday, int id = -1);

**Actor**\* **addActor**(**string** name, **string** lastname, **Date** bday, int id = -1);

**Director**\* **addDirector**(**string** name, **string** lastname, **Date**& bday, int id = -1);

**Hall**\* **addHall**(int rows, int seats, int id = -1);

void **delFilm**(int id);

void **delSession**(int id);

void **delActor**(int id);

void **delClient**(int id);

void **delDirector**(int id);

void **delHall**(int id);

Реализованы они тоже одинаково для всех сущностей

**Client**\* **SaveAndLoad**::**addClient**(**string** name, **string** lastname, **Date** bday, int id) {

    if (id == -1) {

        id = this->clients.**size**() + 1;

    }

    Client\* newClient = new **Client**(name, lastname, bday, id);

    this->clients.**push\_back**(newClient);

    return newClient;

}

void **SaveAndLoad**::**delClient**(int id)

{

    Client\* client = this->**getClientById**(id);

    auto d = **std**::**find**(this->clients.**begin**(), this->clients.**end**(), client);

    if (d != this->clients.**end**()) {

        this->clients.**erase**(d);

        delete client;

    }

}

# 4. Тестирование

При запуске программы открывается главное меню

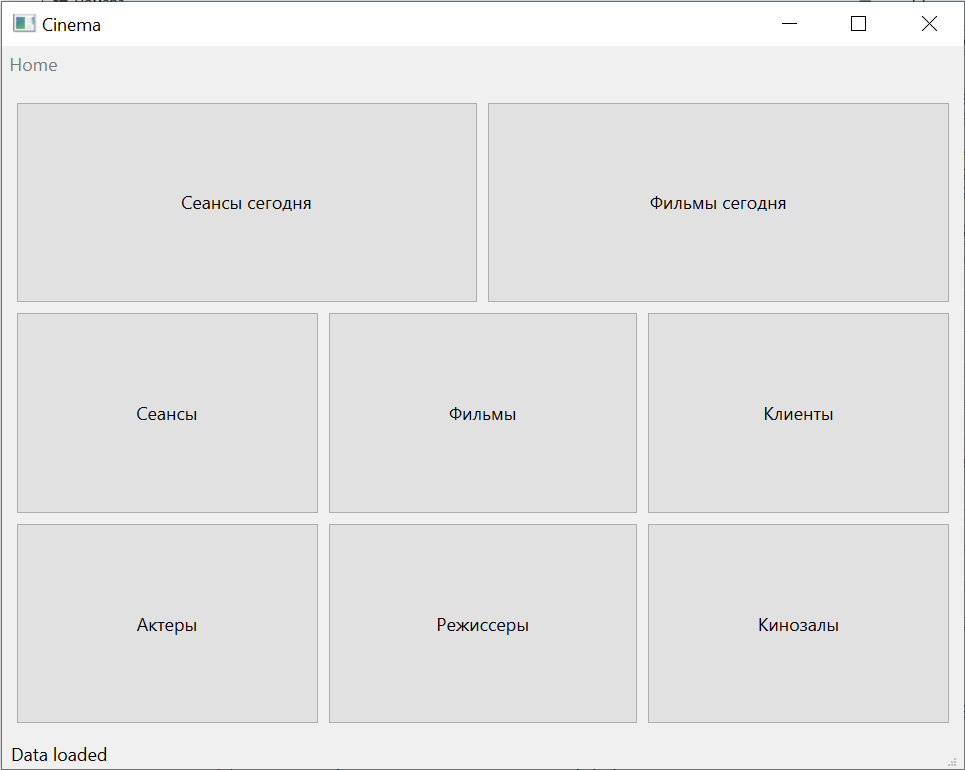


Рисунок . Главное меню

Пройдемся по основным разделам:

* Актеры

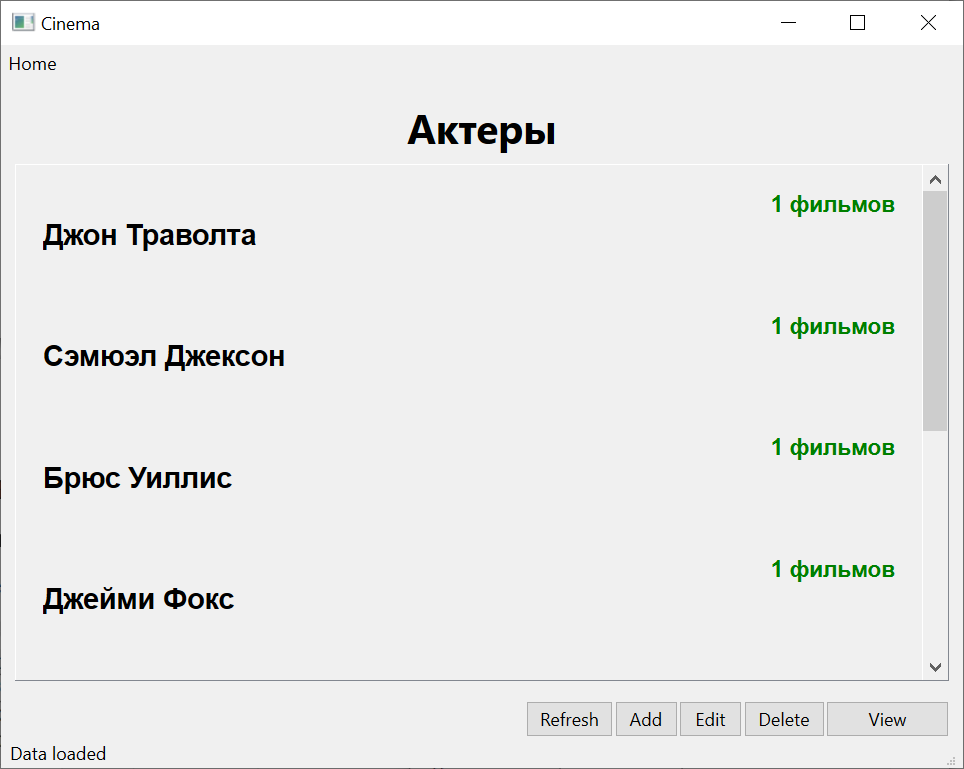


Рисунок . Раздел актеров

Просмотр актера

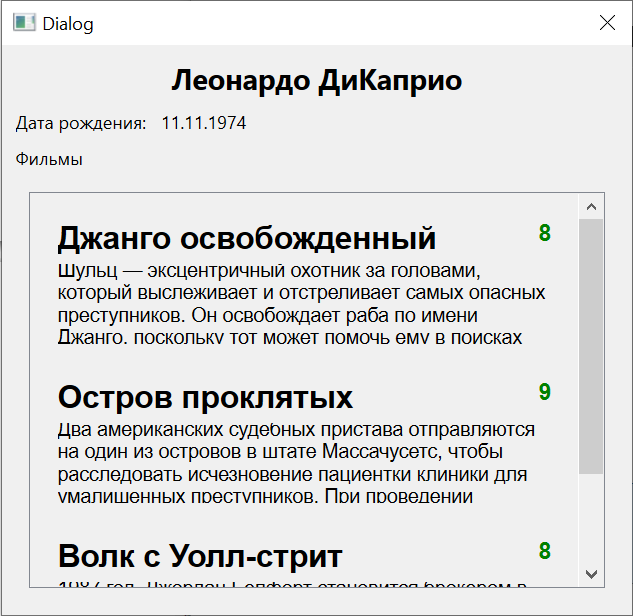


Рисунок . Просмотр актера

Попробуем изменить ему имя

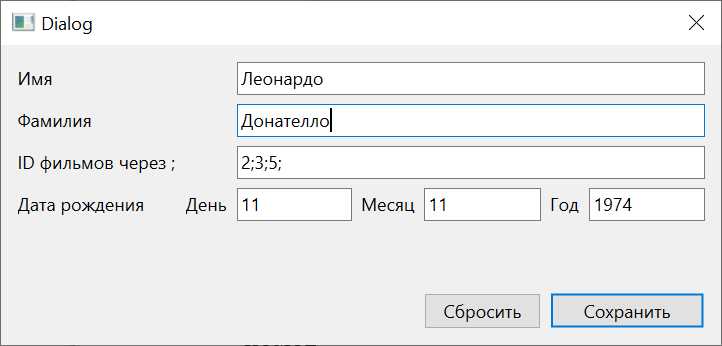


Рисунок . Изменение актера

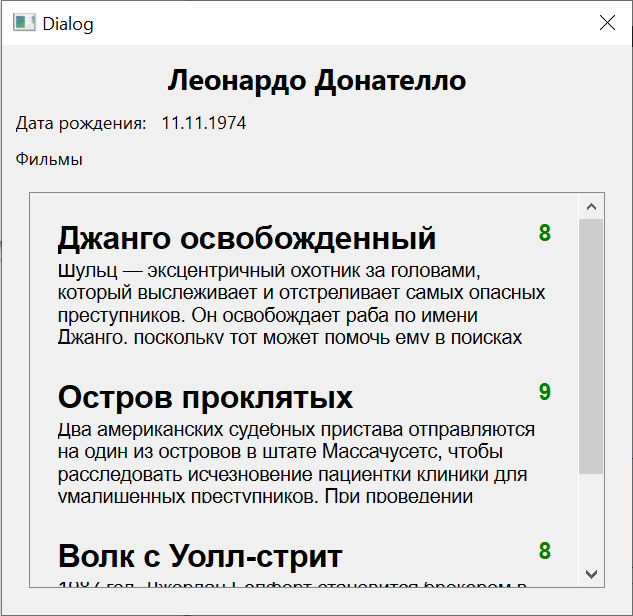


Рисунок . Результат изменения

* Фильмы

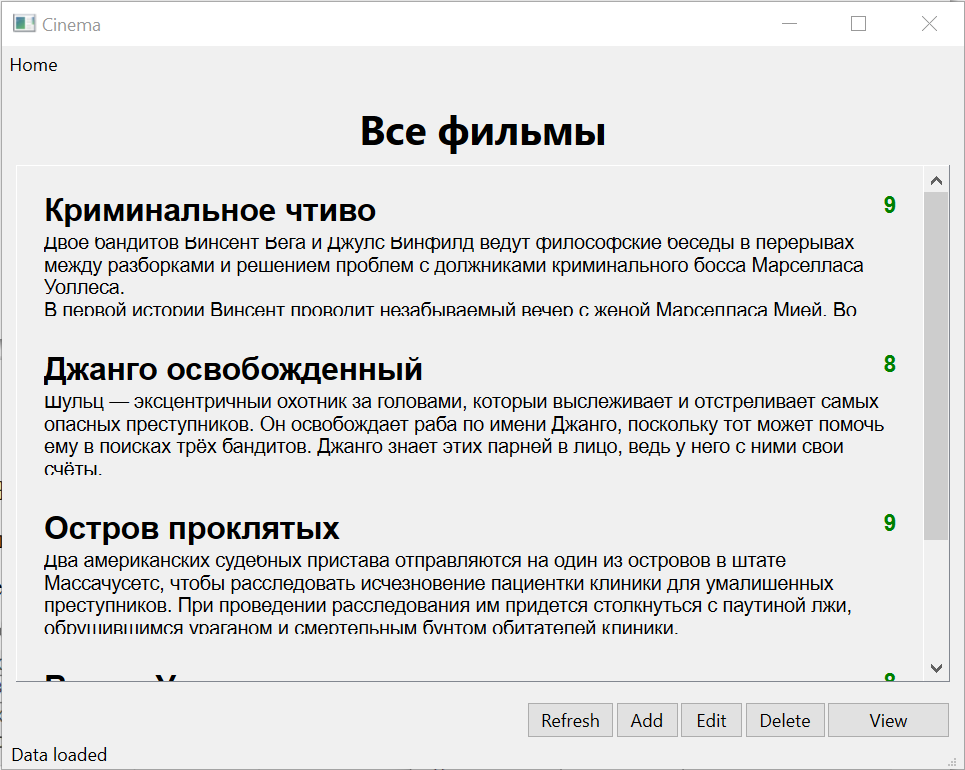


Рисунок . Раздел фильмов

Просмотр информации о фильме

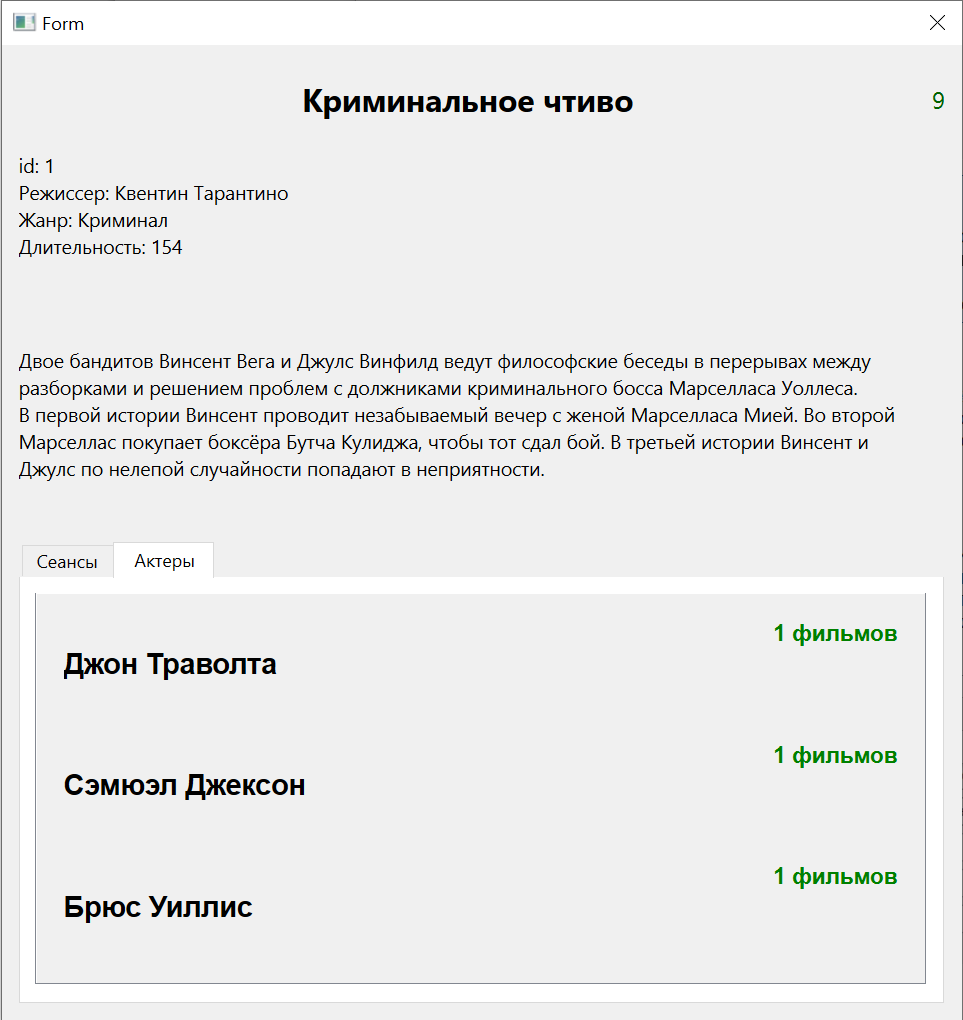


Рисунок . Просмотр фильма

Попробуем изменить его описание

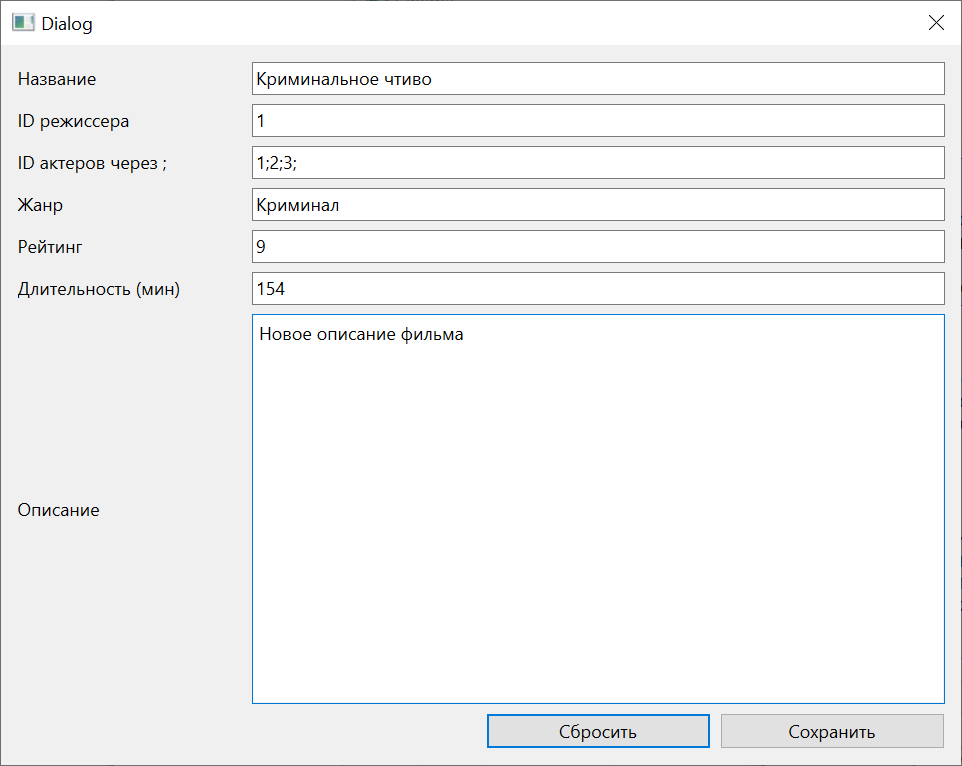


Рисунок . Изменение фильма

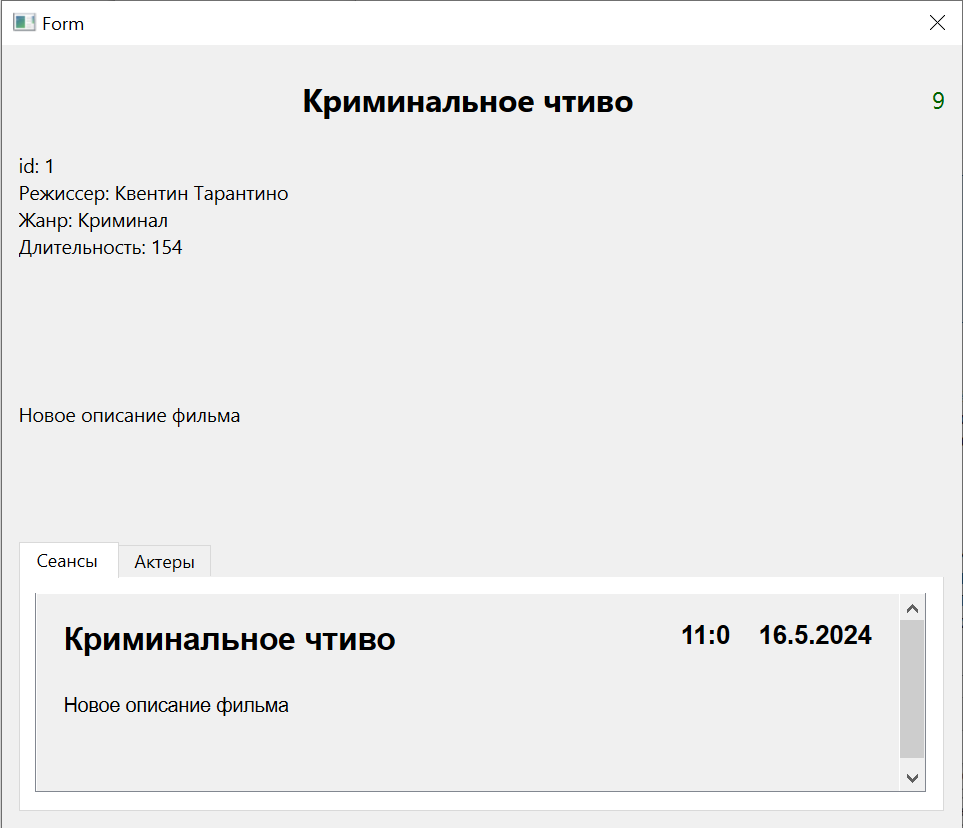


Рисунок . Результат изменения описания

* Сеансы

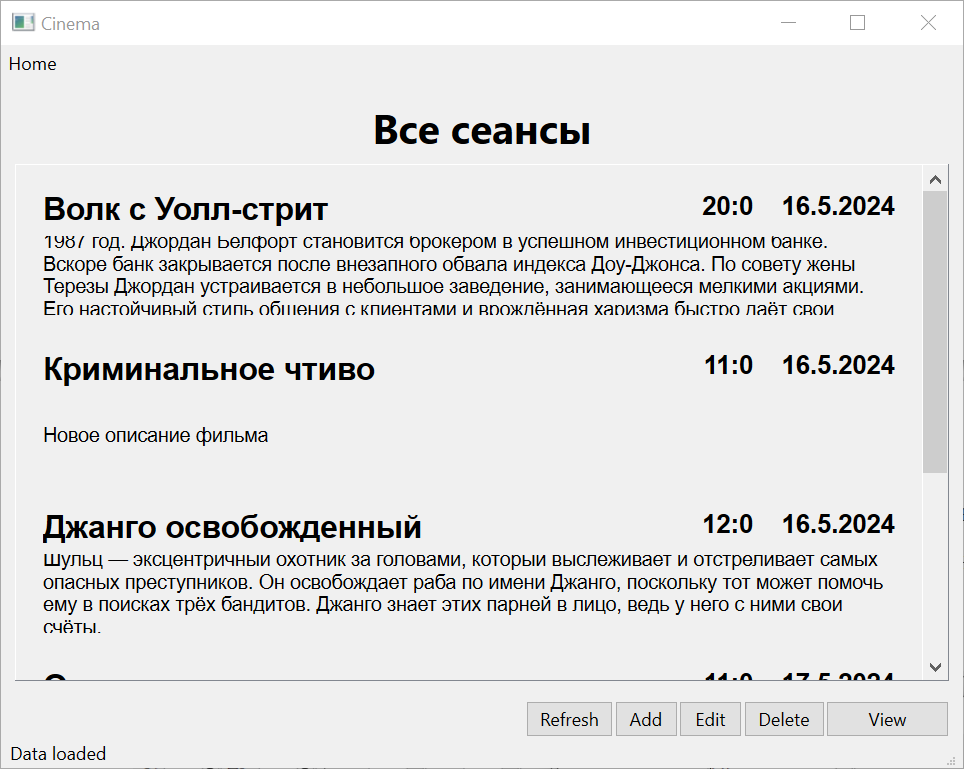


Рисунок . Раздел сеансов

Откроем страницу сеанса

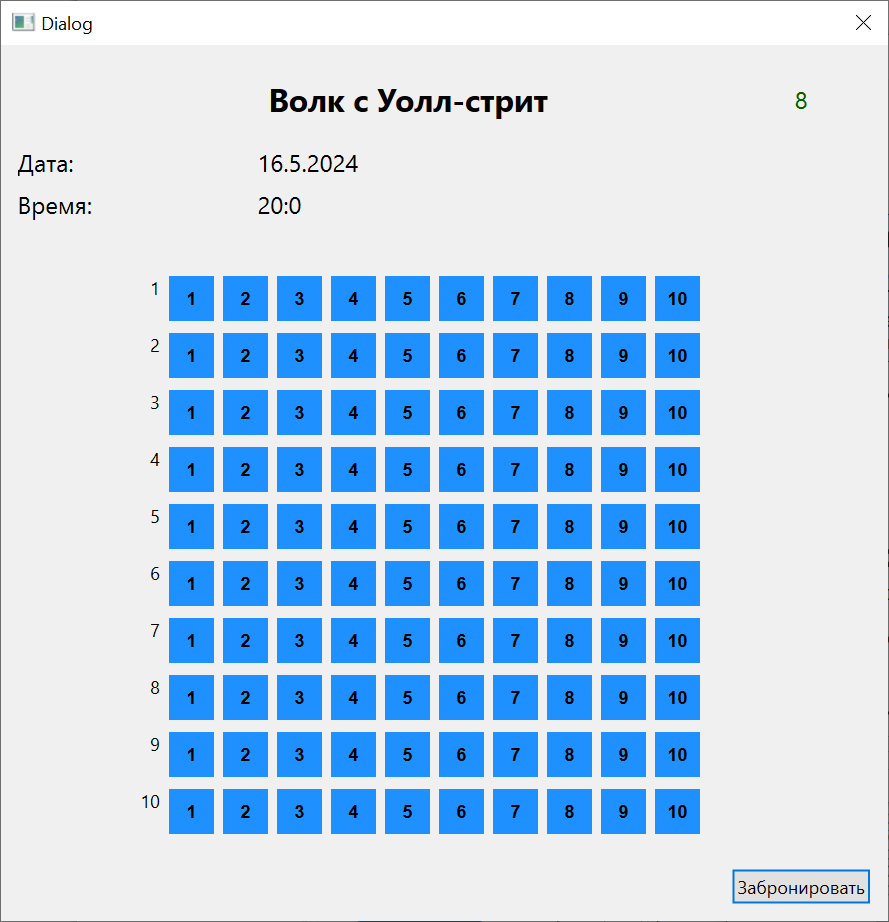


Рисунок . Страница сеанса

Попробуем забронировать место

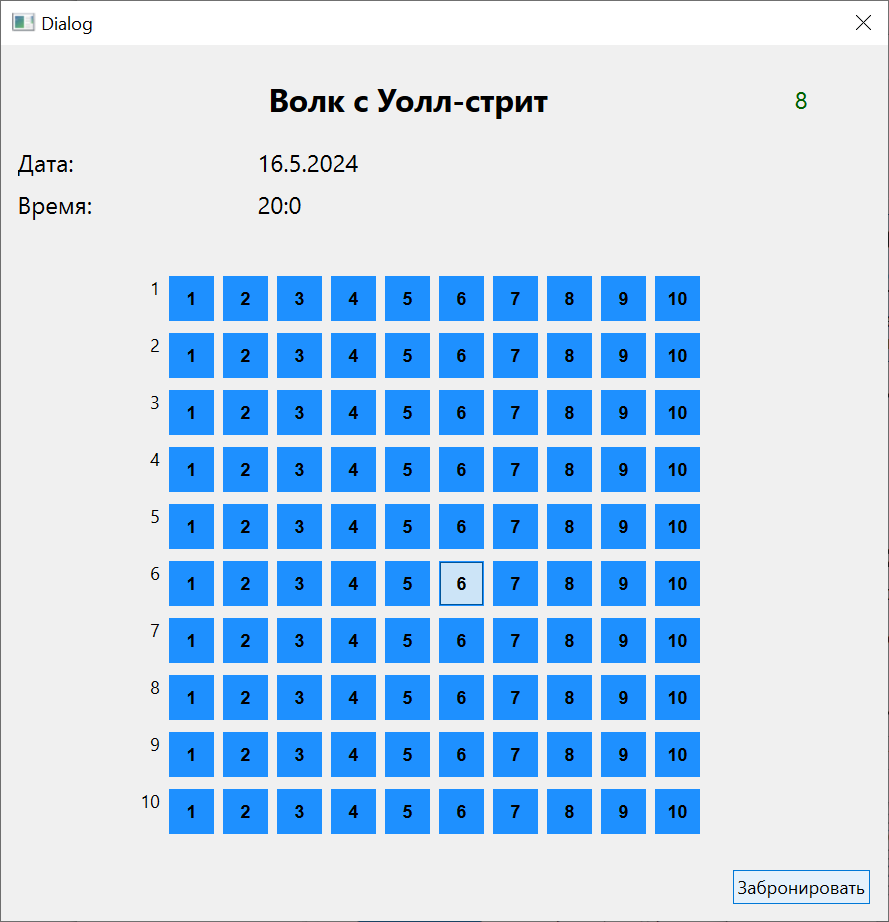


Рисунок . Выбор места

Программа предлагает выбрать клиента, но можем оставить строку пустой в случае, если клиент не зарегистрирован

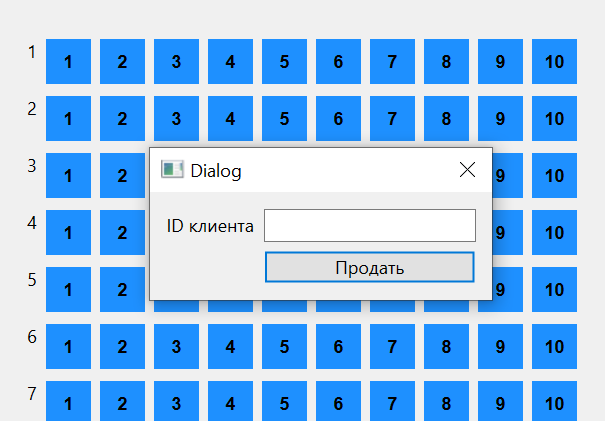


Рисунок . Выбор клиента

Место забронировано

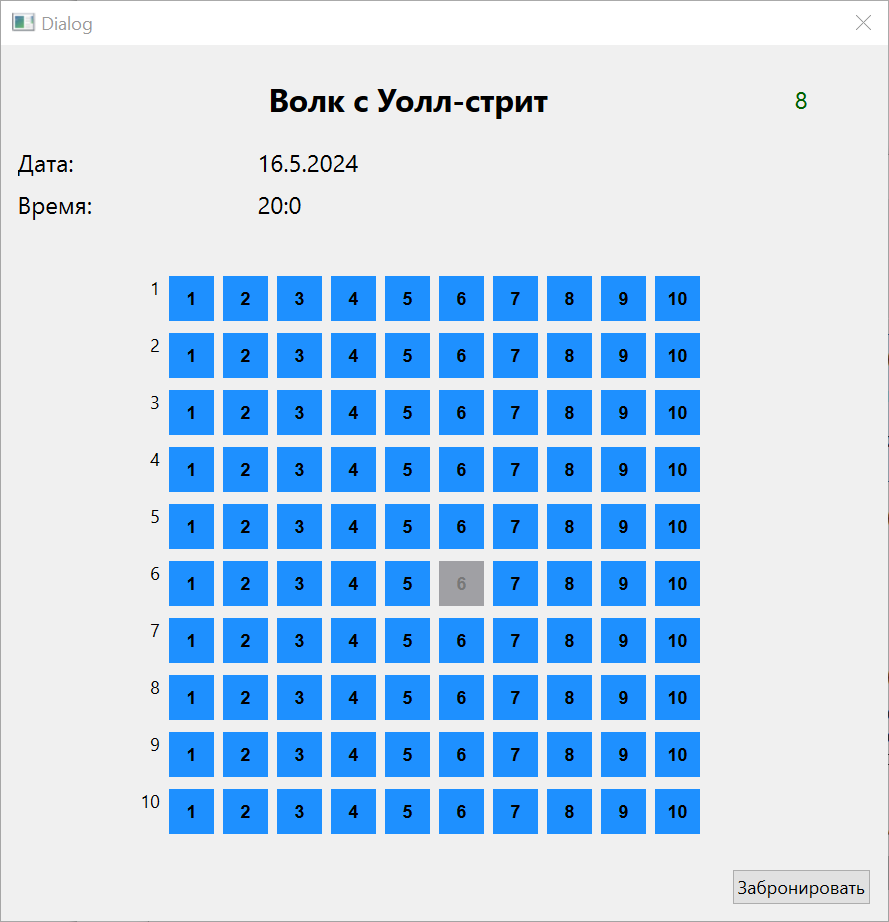


Рисунок . Результат бронирования

# Заключение

В процессе выполнения курсового проекта было спроектировано и реализовано программное решение в соответствии с выбранной предметной областью. Применены принципы ООП (создана иерархия классов), а также шаблон проектирования – Singleton.

В процессе выполнения проекта приобретены и улучшены навыки создания приложений с пользовательским графическим интерфейсом.

Как можно было заметить, логика работы с БД реализована так, что при загрузке программы, через класс SaveAndLoad загружаются все объекты из БД. А дальше при добавлении, удалении и получении сущностей, работа происходит с объектами внутри программы. Такой метод не очень оптимизировано использует ресурсы компьютера. Если данных будет много, то они будут попросту забивать оперативную память.

Однако, чтобы исправить ситуацию нужно только переопределить логику методов add\* get\* и del\* так, чтобы они делали запрос к БД, а методы save\* и load\* можно не использовать. Благодаря тому, что используется паттерн Singleton, изменений в других частях программы вносить не придется т.к. везде идет обращение к классу-одиночке.