TOMSK POLYTECHNIC UNIVERSITY

ТОМСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Министерство науки и высшего образования Российской Федерации
федеральное государственное автономное
образовательное учреждение высшего образования
«Национальный исследовательский Томский политехнический университет» (ТПУ)

Инженерная школа информационных технологий и робототехники
Направление подготовки 09.03.04 Программная инженерия
Отделение информационных технологий

КУРСОВОЙ ПРОЕКТ
по дисциплине Проектирование и архитектура программных систем

Тема
Проектирование программной системы для людей с сердечно-сосудистыми заболеваниями

Студент

Группа	ФИО	Подпись	Дата
8К92	Гавричкина Анастасия Витальевна		24.05.22

Преподаватель

Должность	ФИО	Ученая степень, звание	Подпись	Дата
Доцент (ОИТ, ИШИТР)	Поляков Александр Николаевич	Кандидат технических наук		24.05.22

Томск – 2022 г.

Оглавление

Оглавление 2
Введение 3
Функциональные и нефункциональные требования к системе 4
Описание требований к системе: варианты использования 5
Выявление классов. Построение и описание диаграммы классов анализа........... 8
Построение и описание диаграмм состояний 11
Построение и описание диаграммы проектных классов 13
Построение и описание диаграмм последовательности для операций проектных классов 17
Построение и описание диаграммы пакетов.. 18
Модульное и интеграционное тестирование 19
Паттерны ООП: одиночка... 27
Паттерны ООП: медиатор... 30
Паттерны ООП: фабричный метод 33
Паттерны ООП: абстрактная фабрика.. 36
Паттерны ООП: прокси... 42
Паттерны ООП: состояние.. 45
Паттерны ООП: стратегия 50
Паттерны ООП: легковес 55
Заключение... 60
Список литературы и источников... 61

Введение

С каждым днем значимость информационных систем становится все более и более неоспорима. Невозможно представить современную жизнь без электронных устройств, транспортных средств, различных Интернет-сервисов и т.д. Качество нашей жизни напрямую зависит от программного обеспечения и уровня его реализации, а также от параметров информационных систем, определяемых в процессе проектирования.

Все это стало неотъемлемой частью нашей повседневной жизни, и конечно же, особое место в ней занимают мобильные устройства, что говорит об актуальности разработки программного обеспечения для данных гаджетов.

На сегодняшний день мобильные технологии охватывают всё больше сфер деятельности человека. Рост рынка мобильных приложений показывает значимость и удобство использования мобильных устройств в различных аспектах жизни. Это касается и области здравоохранения. Возможность создавать расписание приемов лекарств по назначению врача, а также записаться на прием в медицинское учреждение в любое удобное время, будучи дома, на работе или даже стоя в пробке обеспечивает удобство только для пациентов, а также эффективность медицинского обслуживания.

В рамках данной работы было принято решение спроектировать систему, представляющую собой мобильное приложение-помощника для людей с сердечно-сосудистыми заболеваниями.

Для реализации данной системы были поставлены следующие задачи:

- Определить функциональные и нефункциональные требования к системе.
- Описать варианты ее использования.
- Проанализировав требования, построить и описать диаграмму классов анализа, а также проектных классов.
- Построить и описать конечный автомат (диаграмму состояний) объектов системы, а также диаграмму последовательности для операций, присутствующих на диаграмме проектных классов.
- Построить и описать диаграмму пакетов системы.
- Создать несколько осмысленных модульных или интеграционных тестов и добиться их успешного выполнения.
- Реализовать различные паттерны проектирования на примере классов из диаграммы проектных классов.

Функциональные и нефункциональные требования к системе

Для проектирования системы необходимо выявить требования к ней. Требования могут быть функциональными (описывать, что система должна делать) и нефункциональными (описывать ограничения, накладываемые на систему, например, как должна вести себя система, соответствие стандартам, практикам, и т. д.).

В рамках выбранной предметной области было выявлено 10 функциональных и 5 нефункциональных требований к проектируемой системе.

Функциональные требования:

1. Обеспечивает регистрацию пользователя: с обязательным указанием ФИО, пола, даты рождения, электронной почты, телефона, а также уникального логина и пароля.
2. Обеспечивает авторизацию пользователя: по логину и паролю.
3. Дает возможность сбросить пароль при невозможности пользователя вспомнить пароль от учетной записи: по номеру телефона или с помощью электронной почты.
4. Позволяет отправить заявку на прием.
5. Показывает назначенные посещения у специалиста: дату, время, ФИО врача, его должность и кабинет.
6. Дает возможность создавать расписание приема препаратов: с указанием названия лекарства, периода его приема и дозировки на день.
7. Предоставляет доступ к расписанию приема препаратов в офлайн-режиме.
8. Отправляет push-уведомления о назначенных консультациях у врача или о необходимости принятия лекарства.
9. Дает возможность просматривать советы специалистов или дополнительную информацию, связанную с сердечно-сосудистыми заболеваниями.
10. Позволяет заполнять анкету самодиагностики сердечно-сосудистых заболеваний и давать предварительный диагноз.

Нефункциональные требования:

1. Система предназначена для мобильных устройств с ОС Android.
2. Написана на языке разработки Java.
3. Для хранения данных используется СУБД SQLite.
4. Обеспечивает безопасность учетных записей пользователей: хеширование паролей.
5. Сохраняет сеанс пользователя после закрытия приложения.

Таким образом, в данном разделе был определен ряд основных функциональных и нефункциональных требований к проектируемой системе.

Описание требований к системе: варианты использования

Вариант использования описывает типичное взаимодействие между пользователем и системой и отражает представление о поведении системы с точки зрения пользователя. Для наглядного представления вариантов использования применяются диаграммы вариантов использования [1].

В рамках данной работы построение UML-диаграммы осуществлялось с помощью онлайн-редактора draw.io [2].

Ниже представлена диаграмма вариантов использования приложения для нескольких требований:

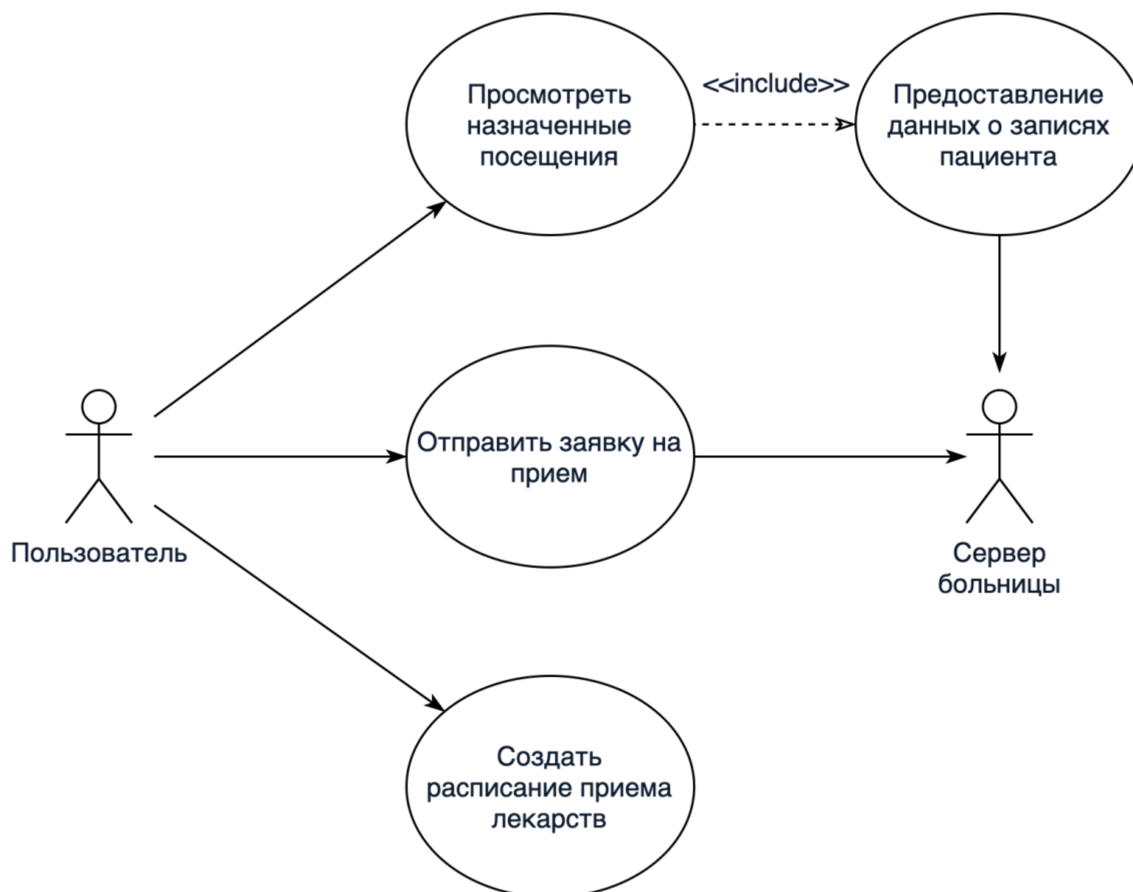


Рисунок 1. Диаграмма вариантов использования приложения

Вариант «Просмотреть назначенные посещения».

Вариант использования «Просмотреть назначенные посещения» позволяет пользователю ознакомиться со своим расписанием консультаций у врача.

Предусловие: авторизация пользователя.

Основной поток событий:

1. Вариант использования начинается, когда пользователь переходит в окно назначенных посещений.
2. Приложение по ФИО и номеру полиса запрашивает данные больницы о записях пациента на прием к специалистам.
3. Приложение отображает информацию о назначенных приемах: даты консультаций, ФИО врачей, номера кабинетов.
4. Вариант использования завершается.

Вариант «Отправить заявку на прием».

Вариант использования «Отправить заявку на прием» позволяет пользователю мобильного приложения подать заявление на приём к специалисту.

Предусловие: авторизация пользователя.

Основной поток событий:

1. Вариант использования начинается, когда пользователь переходит в окно назначенных посещений и нажимает на кнопку «Создать заявку»
2. Приложение открывает окно бланка для подачи заявки.
3. Пользователь вводит в обязательном порядке ФИО и телефонный номер.
4. Пользователь нажимает кнопку «Отправить».
5. Приложение подтверждает отправку заявки на прием.
6. Вариант использования завершается.

Альтернативный поток событий 1. Ввод некорректных данных.

4a1. Приложение информирует пользователя о некорректности введённых данных (при наличии незаполненных полей или недопустимых символов).

4a2. Вариант использования завершен.

Альтернативный поток событий 2. Ошибка при отправке заявки.

5a1. Приложение информирует пользователя о невозможности связаться с сервером больницы.

5a2. Вариант использования завершается.

Вариант «Создать расписание приема лекарств».

Вариант использования «Создать расписание приема лекарств» дает возможность пользователю создавать график приема препаратов.

Предусловие: авторизация пользователя.

Основной поток событий:

1. Вариант использования начинается, когда пользователь переходит в окно расписания приема лекарств.
2. Пользователь нажимает на кнопку «Добавить».
3. Приложение открывает окно бланка для дальнейшего формирования расписания приема лекарства.
4. Пользователь заполняет в обязательном порядке поля Название препарата, Количество приемов в день, Время приема, а также по желанию Продолжительность курса и Доза.
5. Пользователь нажимает кнопку «Сохранить».
6. Приложение подтверждает успешное создание графика приема препарата.
7. Вариант использования завершается.

Как постусловие после успешного выполнения варианта использования в окне расписания приема лекарств появляется график приема добавленного препарата: его название, время его приема, а также дозировка (если она была указана).

Альтернативный поток событий 1. Ввод некорректных данных.

5a1. Приложение информирует пользователя о некорректности введённых данных (при наличии незаполненных полей).

5a2. Вариант использования завершается.

Альтернативный поток событий 2. Ошибка при попытке создания графика приема препарата.

6a1. Приложение информирует пользователя о невозможности связаться с сервером базы данных и предлагает попробовать создать расписание позже.

6a2. Вариант использования завершается.

Таким образом, была построена UML-диаграмма вариантов использования мобильного приложения с указанием их краткого описания, а также основных и альтернативных потоков.

Выявление классов. Построение и описание диаграммы классов анализа

Класс анализа – это укрупненная абстракция, которая на концептуальном уровне (без точного определения атрибутов и операций) описывает некоторый фрагмент системы [4].

Далее представлена концептуальная UML-диаграмма классов для мобильного приложения в соответствии с требованиями проекта:

Рисунок 2. Концептуальная UML-диаграмма классов мобильного приложения

Ниже приведено краткое описание классов, их атрибутов и методов, показанных в модели.

Класс: Пользователь

Основные атрибуты:

- ID пользователя;
- Имя пользователя;
- Отчество пользователя;
- Фамилия пользователя;
- Пол;
- Дата рождения;
- Электронная почта;
- Телефон.

Основные методы:

- Изменить_личные_данные() – позволяет пользователю изменить личные данные;
- Изменить_пароль() – позволяет пользователю изменить пароль.

Класс: Доктор

Основные атрибуты:

- ID доктора;
- Имя доктора;
- Отчество доктора;
- Фамилия доктора;
- Специальность.

Класс: Препарат

Основные атрибуты:

- ID препарата;
- ID пользователя;
- Название препарата;
- Период приема препарата;
- Дозировка.

Основные методы:

- Добавить_препарат() – позволяет добавить препарат с указанием его названия, периода приема и дозировки (время приема, доза);
- Удалить_препарат() – позволяет удалить препарат;
- Изменить_препарат() – позволяет изменить информацию о препарате: название, период приема и дозировку (время приема, доза).

Класс: Посещение доктора

Основные атрибуты:

- ID посещения;
- ID пользователя;
- ID доктора;
- Дата посещения;
- Время посещения;
- Кабинет.

Основные методы:

- Отказаться_от_посещения() – позволяет отправить заявку на удаление назначенного посещения;
- Подать_заявку_на_прием() – позволяет отправлять заявку на прием в клинику, с указанием ФИО и телефона пациента.

Класс: Доступ пользователя

Основные атрибуты:

- Статус доступа.

Основные методы:

- Определить_статус() – перенаправить на авторизацию или регистрацию.

Класс: Авторизация

Основные методы:

- Авторизация() – позволяет авторизоваться в приложении;

- Сброс_пароля() – позволяет отправлять заявку на прием в клинику, с указанием ФИО и телефона пациента.

Класс: Регистрация

Основные методы:

- Регистрация() – позволяет зарегистрироваться в приложении.

В результате работы была построена концептуальная диаграмма классов системы с указанием их краткого описания, основных атрибутов и методов.

Построение и описание диаграмм состояний

Наличие у экземпляра сущности нескольких состояний, отличающихся от простой схемы «исправен – неисправен» или «активен – неактивен», служит признаком необходимости построения диаграммы состояний.

Диаграмма состояний показывает конечный автомат, состоящий из набора состояний, переходов, событий и деятельностей. Она иллюстрирует динамическое представление системы. Диаграммы этого типа особенно важны при моделировании поведения интерфейса, класса или кооперации, так как на них ясно прослеживается упорядоченное по событиям поведение объекта, что особенно удобно при моделировании реактивных систем [1].

Ниже представлена диаграмма состояний мобильного приложения, через которые оно проходит при выборе пользователем действий создания или редактирования карточки препарата, а также создания и отправки заявки на прием у специалиста:

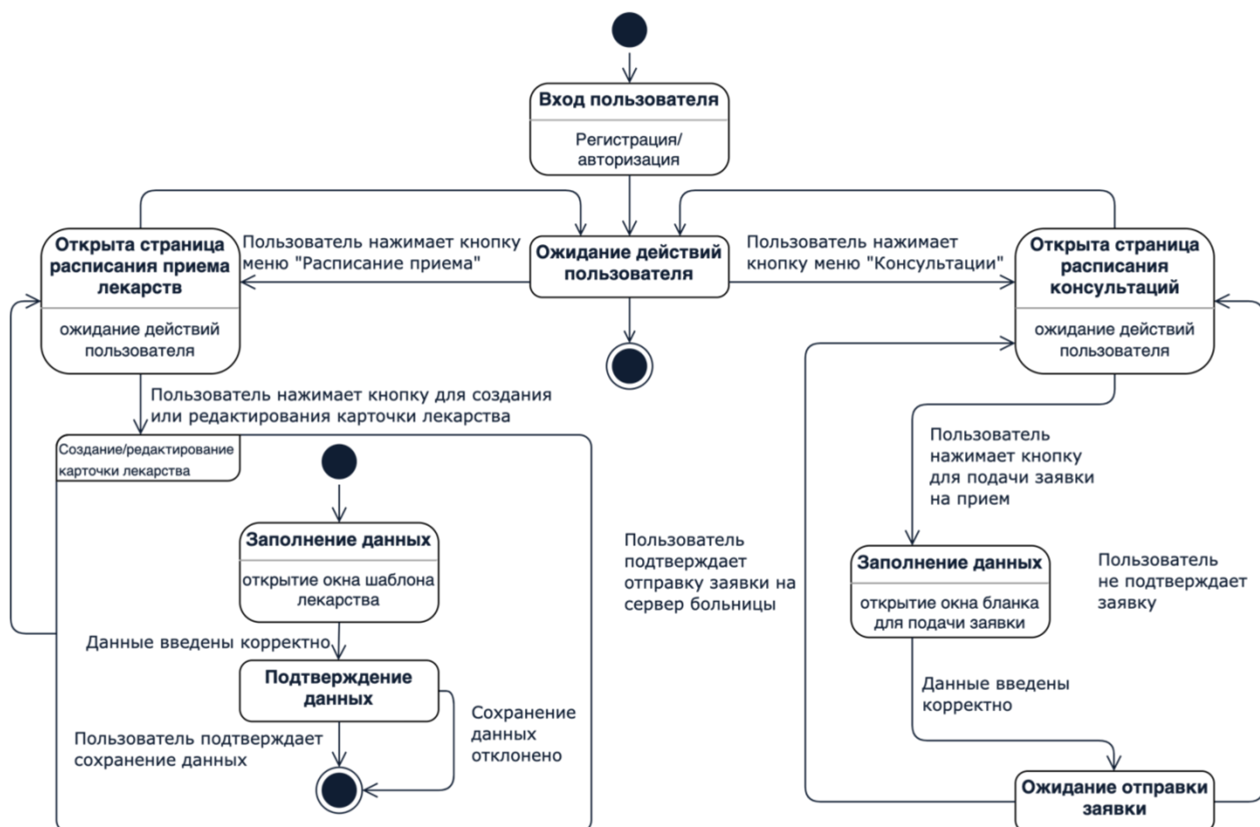


Рисунок 3. Диаграмма состояний проектируемой системы

На данной диаграмме показано, что из начального состояния (открыта стартовая страница) приложение переходит в состояние «Вход пользователя» и ожидает, пока пользователь зарегистрируется или авторизуется. После успешного выполнения действий приложение открывает главную страницу и опять ожидает действий пользователя.

Если пользователь на главной странице нажимает кнопку меню «Расписание приема», то открывается окно «Расписание приема лекарств», в котором приложение также ожидает активности пользователя. Далее, если пользователь нажмет кнопку для создания или редактирования препарата,

приложение перейдет в подсостояние «Создание/редактирование карточки лекарства». В данном состоянии приложение сначала будет ожидать заполнения данных в окне шаблона лекарства, потом перейдет в ожидание подтверждения данных. В случае если сохранение данных было отклонено, то приложение выходит из подсостояния и возвращается к состоянию ожидания действий пользователя в окне «Расписание приема лекарств». Если же сохранение данных было подтверждено, приложение также выходит из подсостояния, создав новую запись о лекарстве в базе данных.

Если пользователь на главной странице нажимает кнопку «Консультации», то будет открыто окно «Расписание консультаций», в котором приложение ожидает действий пользователя. При нажатии кнопки для подачи заявки начинается состояние ее заполнения. После введения пользователем корректных данных приложение ожидает подтверждение отправки заявки на сервер больницы. После подтверждения или отклонения отправки приложение возвращается к состоянию ожидания действий пользователя в окне «Расписание консультаций».

В описанных выше состояниях ожидания действий в окнах «Расписание приема лекарств» и «Расписание консультаций» при нажатии пользователем на соответствующую кнопку меню приложение может вернуться в состояние ожидания действий пользователя на главной, которое также является конечным состоянием, так как данная страница имеет кнопку выхода.

В результате была построена диаграмма состояний мобильного приложения для некоторых вариантов использования, а также приведено краткое описание этой диаграммы.

Построение и описание диаграммы проектных классов

Диаграмма классов – это диаграмма, которая показывает набор классов, интерфейсов, коопераций и их связи.

Диаграмма классов используется для моделирования статического представления системы. Это представление, прежде всего, поддерживает ее функциональные требования, то есть описывает услуги, которые система должна предоставлять ее конечным пользователям [1].

Ниже представлена UML-диаграмма проектных классов, которая демонстрирует общую структуру мобильного приложения.

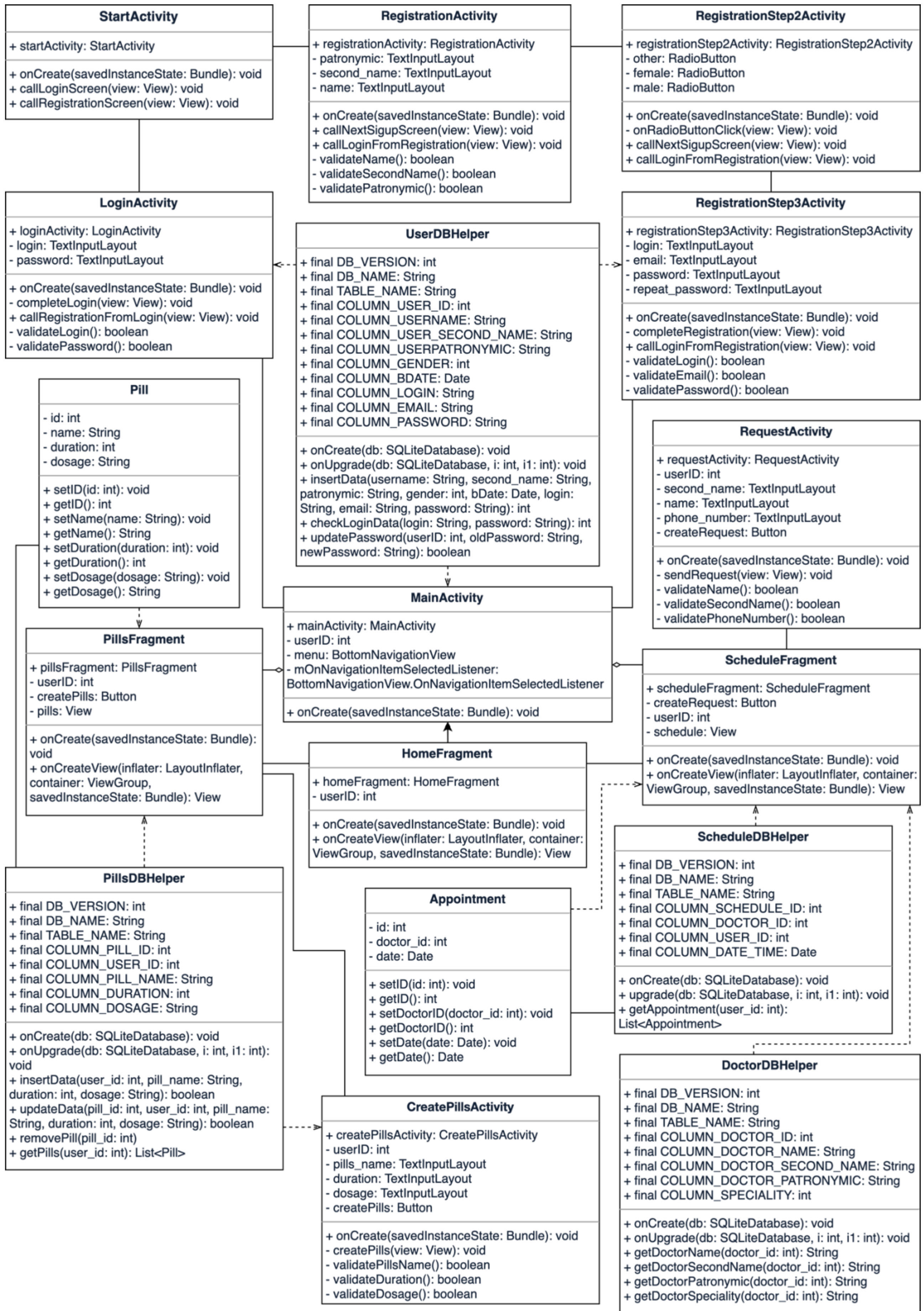


Рисунок 4. UML-диаграмма проектных классов мобильного приложения

Далее приведено краткое описание основных проектных классов.

Основные классы:

- Класс ***StartActivity*** описывает начальное окно мобильного приложения перед регистрацией или авторизацией, открывает данные окна при нажатии на соответствующие кнопки с помощью указанных методов.
- Класс ***LoginActivity*** описывает окно авторизации, также использует класс ***UserDBHelper*** для работы с базой данных (поиск по БД), открывает главное окно мобильного приложения при успешной авторизации.
- Класс ***RegistrationActivity*** описывает окно регистрации (первый шаг), открывает окно следующего шага регистрации при введении корректных данных.
- Класс ***RegistrationStep2Activity*** описывает окно регистрации (второй шаг), также открывает окно следующего шага регистрации при введении корректных данных.
- Класс ***RegistrationSte3Activity*** описывает окно регистрации (заключительный шаг), использует класс ***UserDBHelper*** для работы с БД (внесение новой записи в БД) при введении корректных данных. При успешной регистрации открывает главное окно мобильного приложения.
- Класс ***MainActivity*** описывает главное окно мобильного приложения, которое представляет собой контейнер для нескольких фрагментов. Обобщается классами ***PillsFragment***, ***HomeFragment*** и ***ScheduleFragment*** для переключения фрагментов окна с помощью меню приложения. По умолчанию при авторизации используется второй фрагмент.
- Класс ***HomeFragment*** описывает основной фрагмент главного окна мобильного приложения.
- Класс ***PillsFragment*** описывает фрагмент расписания приема лекарств главного окна мобильного приложения. Использует классы ***PillsDBHelper*** для работы с базой данных расписания приема (поиск по БД), а также класс ***Pill*** для отображения информации о препарате. Открывает окно для создания расписания приема при нажатии на соответствующую кнопку.
- Класс ***CreatePillsActivity*** описывает окно для создания или редактирования расписания приема лекарства, использует класс ***PillsDBHelper*** для работы с базой данных расписания приема (создание новой записи или поиск по БД).
- Класс ***ScheduleFragment*** описывает фрагмент расписания консультаций главного окна приложения. Использует классы ***ScheduleDBHelper*** для работы с базой данных расписания консультаций (поиск по БД), ***DoctorDBHelper*** для работы с БД докторов (поиск данных), а также класс ***Appointment*** для отображения информации о приеме. Открывает окно для подачи заявки на прием к врачу при нажатии на соответствующую кнопку.

- Класс ***RequestActivity*** описывает окно подачи заявки на прием, а также позволяет отправлять ее на сервер больницы.

Дополнительные классы:

- Класс ***UserDBHelper*** позволяет работать с базой данных пользователей, а именно, создавать записи, обновлять БД, получать из нее различные данные.
- Класс ***PillsDBHelper*** позволяет работать с базой данных приема препаратов: создавать, редактировать и удалять записи, получать различные данные.
- Класс ***Pill*** представляет лекарство как объект, описывает его свойства (название, длительность курса, дозировку).
- Класс ***ScheduleDBHelper*** позволяет работать с базой данных консультаций: по ID пользователя осуществлять поиск назначенных ему приемов и возвращать нужную информацию.
- Класс ***DoctorDBHelper*** позволяет работать с базой данных врачей, осуществлять поиск по БД и возвращать данные о врачах.
- Класс ***Appointment*** представляет консультацию как объект, описывает ее свойства (дату консультации и ее время).

Таким образом, была построена UML-диаграмма проектных классов мобильного приложения, реализующих функционал системы, с указанием их краткого описания.

Построение и описание диаграмм последовательности для операций проектных классов

Диаграмма последовательности – это диаграмма взаимодействия, куда входят упорядоченные по времени сообщения, отправляемые и принимаемые экземплярами, которые играют роли. Такая диаграмма иллюстрирует динамическое представление системы [1].

На рисунке 5 представлена UML-диаграмма последовательности действий для основного потока событий варианта использования «Записаться на прием».

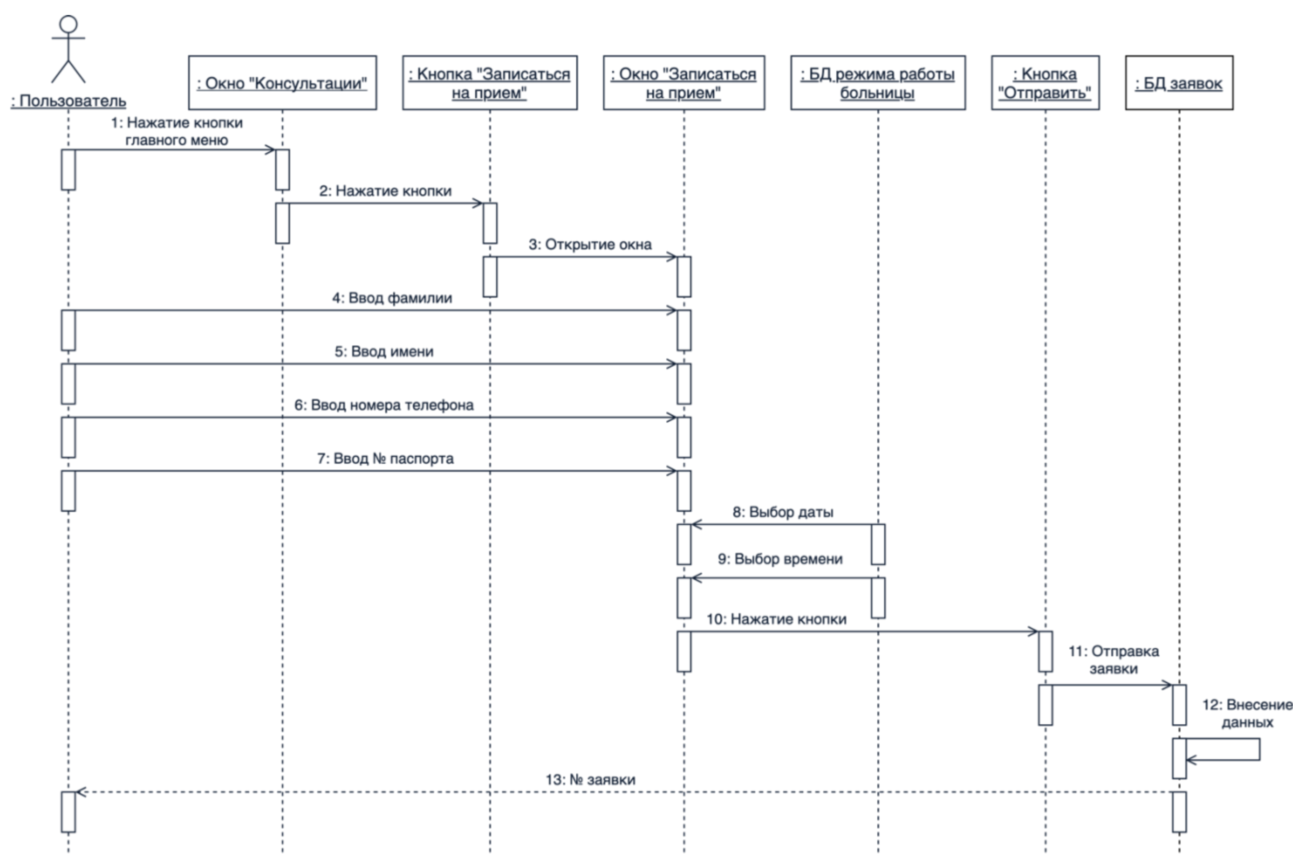


Рисунок 5. Диаграмма последовательности варианта использования «Записаться на приём»

Пользователь через главное меню приложения открывает окно «Консультации» и нажимает на кнопку «Записаться на приём», после чего открывается окно «Записаться на приём». Далее пользователь вводит свою фамилию, имя, номер телефона и номер паспорта. Затем из базы данных режима работы больницы выбираются дата и время приёма. После выбора пользователь нажимает кнопку «Отправить». Далее система создаёт заявку и вносит данные в БД заявок, и после возвращает пользователю номер заявки.

Построение и описание диаграммы пакетов

Пакет – это способ организации элементов модели в блоки, которыми можно распоряжаться как единым целым. С помощью пакетов изображаются различные представления архитектуры системы.

Диаграмма пакетов – структурная диаграмма, показывающая, как элементы модели организованы в пакеты [1].

На рисунке 6 представлена диаграмма пакетов мобильного приложения.

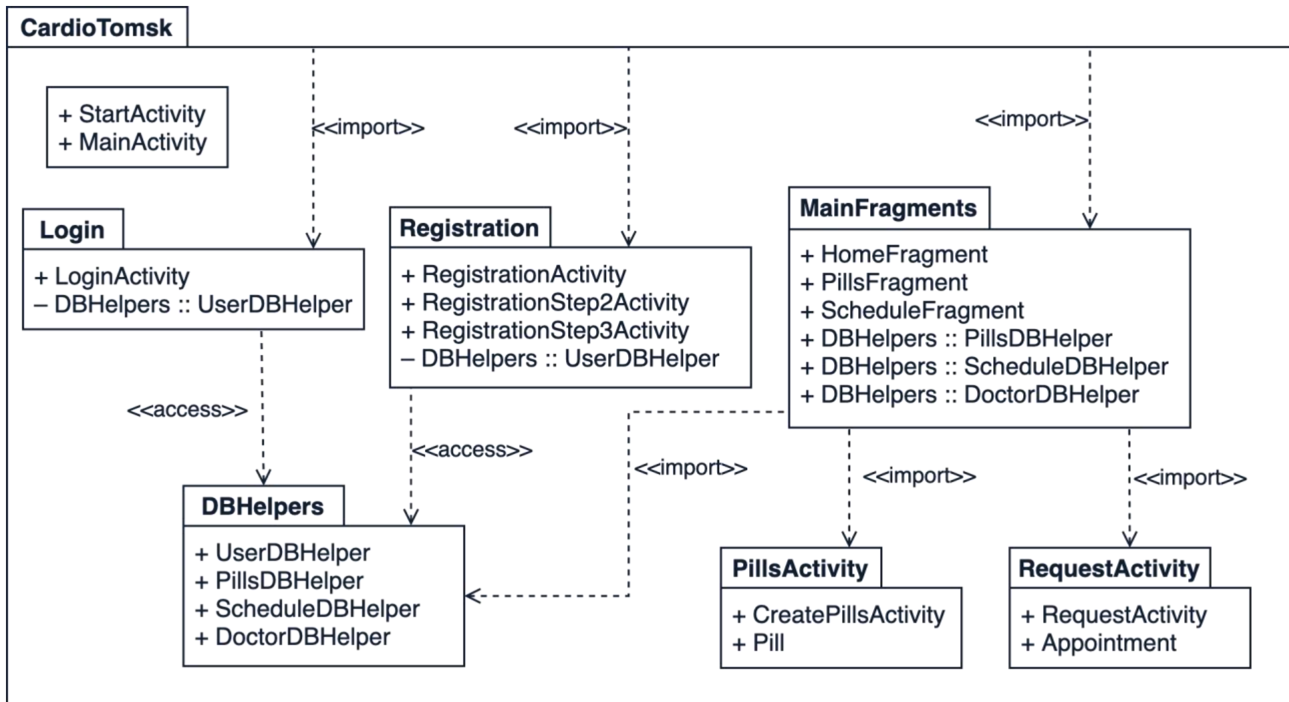


Рисунок 6. Диаграмма пакетов мобильного приложения

На данной диаграмме показано, что все классы располагаются в главном пакете *CardioTomsk*. При этом некоторые классы расположены в отдельных пакетах: *Login*, *Registration*, *MainFragments*, *PillsActivity*, *RequestActivity* и *DBHelpers*.

В пакетах *Login* и *Registration* находятся классы связанные с авторизацией и регистрацией, соответственно, а также происходит закрытое объединение с классом для работы с БД пользователей пакета *DBHelpers*.

В пакете *MainFragments* находятся классы, описывающие фрагменты главного окна приложения. В данный пакет импортируются классы для работы с базами данных докторов, лекарств и расписания консультаций из пакета *DBHelpers*, а также классы пакетов *PillsActivity* и *RequestActivity*, в которых приведены классы для работы с окнами создания лекарств и записи на прием, соответственно.

Таким образом, была построена диаграмма пакетов проектируемой системы, а также приведено их краткое описание.

Модульное и интеграционное тестирование

Модульный тест – фрагмент кода (метод), написанный разработчиком, и проверяющий правильность отдельного модуля (метода) исходного кода программы. Цель модульного тестирования – изолировать отдельные части программы и показать, что по отдельности эти части работоспособны. Модульное тестирование позже позволяет программистам проводить рефакторинг, будучи уверенными, что модуль по-прежнему работает корректно. Для отделения модулей друг от друга используют так называемые mock-объекты – фиктивные объекты, предназначенные исключительно для тестирования [5].

Использование модульного тестирования может помочь уменьшить количество ошибок в процессе разработки, что способствует сокращению цикла разработки и улучшению качества кода.

Для реализации классов были выбраны язык программирования Java, а также среда разработки Eclipse [6], в которую встроена возможность создания модульных тестов.

Далее были созданы некоторые классы, представленные на диаграмме классов, а также методы, описывающие их функционал: классы пользователя *User*, доктора *Doctor* и консультации *Appointment*. Ниже приведен код данных классов:

Класс User:

```
public class User {
  private int id;
  private String first_name;
  private String second_name;
  private String patronymic;
  private int phone;
  private String email;
  private int gender;
  private Date date_of_birth;
  private ArrayList<Appointment> appointments = new ArrayList<>();

  public User() {}
  public User(int id, String first_name, String second_name, String patronymic, int phone, String email, int gender, Date
date_of_birth) {
    this.id = id;
    this.first_name = first_name;
    this.second_name = second_name;
    this.patronymic = patronymic;
    this.phone = phone;
    this.email = email;
    this.gender = gender;
    this.date_of_birth = date_of_birth;
```

```java
}

public int getId() {
  return id;
}
public void setId(int id) {
  this.id = id;
}

public String getFirst_name() {
  return first_name;
}
public void setFirst_name(String first_name) {
  this.first_name = first_name;
}

public String getSecond_name() {
  return second_name;
}
public void setSecond_name(String second_name) {
  this.second_name = second_name;
}

public String getPatronymic() {
  return patronymic;
}
public void setPatronymic(String patronymic) {
  this.patronymic = patronymic;
}

public int getPhone() {
  return phone;
}
public void setPhone(int phone) {
  this.phone = phone;
}

public String getEmail() {
  return email;
}
public void setEmail(String email) {
  this.email = email;
```

```
    }

    public int getGender() {
      return gender;
    }
    public void setGender(int gender) {
              if (gender != 1 || gender != 2 || gender != 0) {
                        throw new NullPointerException("gender is wrong");
              }
      this.gender = gender;
    }

    public Date getDate_of_birth() {
      return date_of_birth;
    }
    public void setDate_of_birth(Date date_of_birth) {
      this.date_of_birth = date_of_birth;
    }

    public ArrayList<Appointment> getAppointments() {
      return appointments;
    }
    public void setAppointments(ArrayList<Appointment> appointments) {
      this.appointments = appointments;
    }

    public String getGenderString() {
                    switch (gender) {
                              case 0: return "Муж";
                              case 1: return "Жен";
                              case 2: return "Не указан";
                    }
                    return null;
            }

    public void addAppointment(Appointment appointment) {
      appointments.add(appointment);
    }
}
```

Класс Doctor:

```
public class Doctor {
	private int id;
	private String first_name;
	private String second_name;
	private String patronymic;
	private String office;
	private String specialization;
	private ArrayList<Appointment> appointments;


	public Doctor() {}
	public Doctor(int id, String first_name, String second_name, String patronymic, String office, String specialization, ArrayList<Appointment> appointments) {
		this.id = id;
		this.first_name = first_name;
		this.second_name = second_name;
		this.patronymic = patronymic;
		this.office = office;
		this.specialization = specialization;
		this.appointments = appointments;
	}

	public int getId() {
		return id;
	}
	public void setId(int id) {
		this.id = id;
	}

	public String getFirst_name() {
		return first_name;
	}
	public void setFirst_name(String first_name) {
		this.first_name = first_name;
	}

	public String getSecond_name() {
		return second_name;
	}
	public void setSecond_name(String second_name) {
		this.second_name = second_name;
```

```
	}

	public String getPatronymic() {
		return patronymic;
	}
	public void setPatronymic(String patronymic) {
		this.patronymic = patronymic;
	}

	public String getOffice() {
		return office;
	}
	public void setOffice(String office) {
		this.office = office;
	}

	public String getSpecialization() {
		return specialization;
	}
	public void setSpecialization(String specialization) {
		this.specialization = specialization;
	}

	public ArrayList<Appointment> getAppointments() {
		return appointments;
	}
	public void setAppointments(ArrayList<Appointment> appointments) {
		this.appointments = appointments;
	}

	public void addAppointment(Appointment appointment) {
		appointments.add(appointment);
	}
}
```

Класс Appointment:

```
public class Appointment {
	private int id;
	private User user;
	private Doctor doctor;
	private Date appointment_date_time;
```

```java
    public Appointment() {}
    public Appointment(int id, User user, Doctor doctor, Date appointment_date_time) {
        this.id = id;
        this.user = user;
        this.doctor = doctor;
        this.appointment_date_time = appointment_date_time;
    }

    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }

    public User getUser() {
        return user;
    }
    public void setUser(User user) {
        this.user = user;
    }

    public Doctor getDoctor() {
        return doctor;
    }
    public void setDoctor(Doctor doctor) {
        this.doctor = doctor;
    }

    public Date getAppointment_date_time() {
        return appointment_date_time;
    }
    public void setAppointment_date_time(Date appointment_date_time) {
        this.appointment_date_time = appointment_date_time;
    }

}
```

Далее было написано несколько методов, реализующих модульное тестирование. Ниже представлен код тестов, а также их краткое описание в комментариях к программе.

Класс TestVoids:

```
class TestVoids {

        /*тест, проверяющий возвращаемое значение, являющееся коллекцией;

         Проверка возвращаемого значения метода getAppointments,
         который должен возвращать коллекцию ArrayList<Appointment>*/
        @Test
        void testGetUserAppointmentList() {
                User user = new User();
                ArrayList<Appointment> appointments = new ArrayList<>();
                user.setAppointments(appointments);
                assertEquals(appointments, user.getAppointments());
        }



        /*тест для метода, не возвращающего значение (void);

         Проверка действия метода addAppointment, который должен
         добавлять в ArrayList объект класса Appointment*/

        @Test
        void testAddUserAppointment() {
                User user = new User();
                Appointment appointment = new Appointment();
                user.addAppointment(appointment);
                assertEquals(1, user.getAppointments().size());
        }



        /*тест, проверяющий, что метод действительно возбуждает исключение
        определённого типа при возникновении исключительной ситуации;

        Проверка возбуждения исключения при попытке добавить
        пользователю неправильное значение поля*/
        @Test
        void testGender() throws NullPointerException {
          try {
                User user = new User();
                user.setGender(3);
          } catch (Exception e) {
                assertEquals("gender is wrong", e.getMessage());
                return;
```

```
            }
            fail("Expected exception was not thrown");
        }
}
```

Далее рисунке представлен успешный результат проведения тестов.

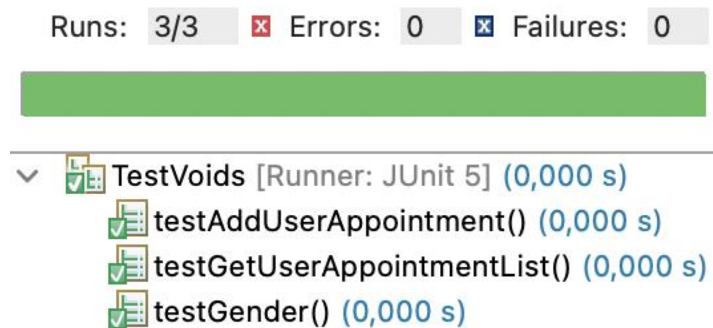


Рисунок 7. Результат проведения модульного и интеграционного

В данной главе показано, как были реализованы некоторые классы, представленные в диаграмме классов, а также проведено успешное модульное тестирование разработанных методов.

Паттерны ООП: одиночка

«Одиночка» – это один из самых простых шаблонов (паттернов) проектирования, который применяется к классу. Он гарантирует, что у класса есть только один экземпляр, и предоставляет к нему глобальную точку доступа [3, 7].

Для реализации данного паттерна было решено расширить диаграмму классов и добавить класс *Hospital*, который будет иметь один экземпляр, так как в рамках данного проекта была выбрана определенная клиника. Данный класс будет хранить информацию о клинике и позволит достать ее из любой части программы: ее название, адрес, список докторов и приемов. Ниже представлен код класса:

Класс Hospital:

```
public class Hospital {

	private static Hospital hospital;
	private static String name = "Кардиологическая клиника";
	private static String city = "Томск";
	private static String street = "Ленина";
	private static String house_number = "12";
	private static ArrayList<Doctor> doctors = new ArrayList<>();
	private static ArrayList<Appointment> appointments = new ArrayList<>();

	private Hospital() {}

	public static Hospital getHospital() {
		if (hospital == null) {
			hospital = new Hospital();
			}
		return hospital;
	}

	public String getName() {
		return name;
	}

	public String getAddress() {
		return city + ", ул. " + street + ", " + house_number;
	}

	public void addDoctor(Doctor doctor) {
		doctors.add(doctor);
```

```
    }

    public Doctor findDoctorById(int id) {
        for (Doctor doctor: doctors) {
            if (doctor.getId() == id) return doctor;
        }
        return null;
    }

    public void removeDoctorById(int id) {
        doctors.remove(Hospital.getHospital().findDoctorById(id));
    }

    public void addAppointment(Appointment appointment) {
        appointments.add(appointment);
    }

    public Appointment findAppointmentById(int id) {
        for (Appointment appointment: appointments) {
            if (appointment.getId() == id) return appointment;
        }
        return null;
    }

    public void removeAppointmentById(int id) {
        appointments.remove(Hospital.getHospital().findAppointmentById(id));
    }
}
```

Для демонстрации результата реализации паттерна «Одиночка» был создан дополнительный класс *Test*, подтверждающий, что создание нескольких объектов класса *Hospital* невозможно, о чем свидетельствует соответствие объявленных объектов.

Класс Test:

```
public class Test {

    public static void main(String[] args) {
        Hospital h1 = Hospital.getHospital();
        Hospital h2 = Hospital.getHospital();

        System.out.println("Соответствие элементов: " + h1.equals(h2));
    }
```

```
}
```

Далее представлен результат реализации данного паттерна.

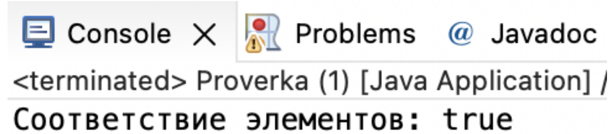


Рисунок 8. Паттерны ООП: одиночка. Результат работы программы

Таким образом, был реализован паттерн «Одиночка», в результате чего была дополнена диаграмма классов с добавлением класса Hospital, хранящего общую информацию о клинике. Данный паттерн довольно удобен в использовании в рамках выбранной предметной области.

Паттерны ООП: медиатор

«Медиатор» – паттерн поведения объектов, предоставляющий единый центр взаимодействия определенной группы объектов, которые должны быть взаимосвязаны друг с другом. Посредник обеспечивает слабую связанность системы, избавляя объекты от необходимости явно ссылаться друг на друга, и позволяя тем самым независимо изменять взаимодействия между ними [3, 8].

Для паттерна «Медиатор» было решено расширить диаграмму классов: добавить модели для реализации простого взаимодействия между пользователем и регистратурой клиники, для чего было создано несколько интерфейсов *ElectronicRegistry* и *PatientRegistry*, класс Registry, представляющий собой регистратуру, а также в ходе проектирования класс User претерпел небольшие изменения, связанные с реализацией интерфейсов. Ниже приведен код данных моделей:

Интерфейс ElectronicRegistry:

```
public interface ElectronicRegistry {
        public void createAppointment (PatientRegistry electronicRegistry, Appointment appointment, boolean status);
}
```

Данный интерфейс содержит только один метод, представляющий собой отправку запроса на создание приема.

Интерфейс PatientRegistry:

```
public interface PatientRegistry {
        public void makingAppointment(Appointment appointment, boolean status);
        public void getAppointment(Appointment appointment, boolean status);
}
```

Для создания заявки и ее получения данный интерфейс содержит два метода, которые принимают в качестве аргументов саму заявку, объект класса *Appointment*, а также статус заявки.

Далее представлен код измененного класса *User*. Теперь он реализует интерфейс *PatientRegistry*, методы которого переопределены в соответствии с действиями пользователя. Для удобства показаны только новые части кода.

Класс User:

```
public class User implements PatientRegistry{
  ...
  public static ElectronicRegistry electronicRegistry;
  @Override
  public void makingAppointment(Appointment appointment, boolean status) {
          System.out.println("Пациент: примите заявку");
  }
  @Override
  public void getAppointment(Appointment appointment, boolean status) {
```

```
            electronicRegistry.createAppointment(this, appointment, status);
    }
}
```

Ниже представлен код класса *Registry*, который также реализует интерфейс *PatientRegistry* и переписывает его методы, давая возможность по статусу заявки «одобрить» ее и присвоить ей id.

Класс Registry:

```
public class Registry implements PatientRegistry {
        public static ElectronicRegistry electronicRegistry;
        @Override
          public void makingAppointment(Appointment appointment, boolean status) {
                if (status) {
                        appointment.setId(1);
                        System.out.println("Электронная регестратура: заявка принята" + "\nНомер приема: " +
appointment.getId());
                } else System.out.println("Электронная регестратура: заявка отклонена");
          }
          @Override
          public void getAppointment(Appointment appointment, boolean status) {
                  electronicRegistry.createAppointment(this, appointment, status);
          }
}
```

Для создания чата между регистратурой и пользователем был также написан класс *Chat*, реализующий интерфейс *ElectronicRegistry* для взаимодействия между пользователями чата.

Класс Chat:

```
public class Chat implements ElectronicRegistry {
        PatientRegistry user;
        PatientRegistry registry;
        public void setUser(PatientRegistry user) {
                this.user = user;
        }
        public void setRegistry(PatientRegistry registry) {
                this.registry = registry;
        }
        @Override
        public void createAppointment(PatientRegistry electronicRegistry, Appointment appointment, boolean status) {
                user.getAppointment(appointment, status);
                registry.getAppointment(appointment, status);
        }
```

```
}
```

В целях проверки был создан дополнительный класс, показывающий реализацию паттерна «Медиатор».

Класс Test:

```
public class Test {

	public static void main(String[] args) {
		Chat chat = new Chat();

		Registry registry = new Registry();
		Registry.electronicRegistry = chat;

		User user_1 = new User();
		User.electronicRegistry = chat;
		Appointment appointment = new Appointment();

		chat.setRegistry(registry);
		chat.setUser(user_1);
		user_1.makingAppointment(appointment, false);
		registry.makingAppointment(appointment, true);

		user_1.makingAppointment(appointment, false);
		registry.makingAppointment(appointment, false);
	}
}
```

На рисунке 9 представлен результат реализации данного паттерна.

```
Пациент: примите заявку
Электронная регестратура: заявка принята
Номер приема: 1
Пациент: примите заявку
Электронная регестратура: заявка отклонена
```

Рисунок 9. Паттерны ООП: медиатор. Результат работы программы

В результате реализации паттерна была дополнена диаграмма классов с добавлением нескольких интерфейсов, а также классов для осуществления простого взаимодействия между пользователем и регистратурой клиники, что не входило в задачи по проектированию системы.

Паттерны ООП: фабричный метод

«Фабричный метод» – паттерн, порождающий классы. Определяет интерфейс для создания объекта, но оставляет подклассам решение о том, какой класс инстанцировать. Фабричный метод позволяет классу делегировать инстанцирование подклассам [3, 9].

Для паттерна «Медиатор» было решено также расширить диаграмму классов и добавить в нее модели для реализации напоминания пользователю принять лекарства. Для это было создано несколько классов с учетом данного паттерна. Ниже приведен код моделей:

Класс Medicine:

```
public class Medicine {
	private String name;
	private String dose;

	public Medicine() {}

	public Medicine(String name, String dose) {
		this.name = name;
		this.dose = dose;
	}

	public String getName() {
		return name;
	}
	public void setName(String name) {
		this.name = name;
	}

	public String getDose() {
		return dose;
	}
	public void setDose(String dose) {
		this.dose = dose;
	}

	public void takeMedicine() {
		System.out.println("Нужно принять лекарство" + ": " + name + " || " + dose);
	}
}
```

Класс *Medicine* является родительским классом, описывающим лекарство, хранящим его имя и дозировку.

Далее представлено несколько классов-наследников, *Pills* и *Syrup*, имеющих тип лекарства и переписывающих метод для напоминания *takeMedicine()*.

Класс Pills:

```
public class Pills extends Medicine{

	private String type = "таблетки";

	@Override
	public void takeMedicine() {
		System.out.println("Нужно принять " + type + ": " + this.getName() + " || " + this.getDose());
	}
}
```

Класс Syrup:

```
public class Syrup extends Medicine{
	private String type = "сироп";

	@Override
	public void takeMedicine() {
		System.out.println("Нужно принять " + type + ":    " + this.getName() + " || " + this.getDose());
	}
}
```

Также для удобства определения класса лекарства было создано перечисление его возможных типов.

Перечисление MedicineType:

```
public enum MedicineType {
	PILLS,
	SYRUP
}
```

Ниже представлен код класса *MedicineFactory*, который будет определять класс лекарства.

Класс MedicineFactory:

```
public class MedicineFactory {
	public Medicine createMedicine (MedicineType type) {
		Medicine medicine = null;

      switch (type) {
        case PILLS:
```

```
            medicine = new Pills();
             break;
          case SYRUP:
            medicine = new Syrup();
             break;
        }
        return medicine;
    }
}
```

Для демонстрации результата реализации паттерна «Фабричный метод» был создан дополнительный класс *MedicineReminder*, представленный ниже.

Класс MedicineReminder:

```
public class MedicineReminder {

      public static void main(String[] args) {
            MedicineFactory medicineFactory = new MedicineFactory();
            Medicine medicine = medicineFactory.createMedicine(MedicineType.PILLS);
            medicine.setName("Аспирин");
            medicine.setDose("1 таблетка");

            Medicine medicine2 = medicineFactory.createMedicine(MedicineType.SYRUP);
            medicine2.setName("Синекод");
            medicine2.setDose("1 чайная ложка");

            medicine.takeMedicine();
            medicine2.takeMedicine();
      }
}
```

Ниже представлен результат реализации данного паттерна.

```
Нужно принять таблетки: Аспирин || 1 таблетка
Нужно принять сироп:    Синекод || 1 чайная ложка
```

Рисунок 10. Паттерны ООП: фабричный метод. Результат работы программы

Для реализации паттерна «Фабричный метод» были спроектированы дополнительные классы, предоставляющие возможность получать напоминания о необходимости выпить лекарства. Таким образом, паттерн удачно подходит для данной предметной области.

Паттерны ООП: абстрактная фабрика

«Абстрактная фабрика» – паттерн, порождающий классы, также как и «Фабричный метод». Предоставляет интерфейс для создания семейств взаимосвязанных или взаимозависимых объектов, не специфицируя их конкретных классов. У абстрактной фабрики, как правило, есть несколько реализаций. Каждая из них отвечает за создание продуктов одной из вариаций [3, 10].

В рамках проекта довольно сложно добавить данный паттерн. Для рассмотрения примера его реализации было принято решение добавить классы и интерфейсы, в которых нет необходимости в данном проекте.

Для примера реализации паттерна добавим к записи у врача процедуру, тип которой будем определять по принадлежности доктора к определенному отделению клиники. Для это было создано несколько интерфейсов и классов с учетом данного паттерна.

Определим абстрактный интерфейс для процедур, а также метод, в котором в дальнейшем будем отображать информацию о записи пользователя на данную процедуру.

Интерфейс Procedure:

```
public interface Procedure {
	public void makingProcedure(Appointment appointment);
}
```

Далее представлено несколько классов (*OperationProcedure* и *ExaminationProcedure*), реализующих данный интерфейс. Данные классы переопределяют метод *makingProcedure()* и выводят информацию пациенте, приеме и докторе в соответствии с типом процедуры.

Класс OperationProcedure:

```
public class OperationProcedure implements Procedure {

	@Override
	public void makingProcedure(Appointment appointment) {
		System.out.println("Пациент: " + appointment.getUser().getFullName());
		System.out.println("Дата операции: " + appointment.getFormatDate());
		System.out.println("Хирург: " + appointment.getDoctor().getFullName());
		System.out.println("----------------------------\n\n");
	}
}
```

Класс ExaminationProcedure:

```
public class ExaminationProcedure implements Procedure {
	private String name = "Осмотр";

	@Override
```

```
    public void makingProcedure(Appointment appointment) {
        System.out.println("Пациент: " + appointment.getUser().getFullName());
        System.out.println("Назначенная процедура: " + name);
        System.out.println("Дата: " + appointment.getFormatDate());
        System.out.println("Кардиолог: " + appointment.getDoctor().getFullName());
        System.out.println("---------------------------\n\n");
    }
}
```

Далее определим абстрактную фабрику – интерфейс *HospitalDepartment*, который будет создавать существующие типы процедур с помощью метода *assignProcedure()*.

Интерфейс HospitalDepartment:

```
public interface HospitalDepartment {
    Procedure assignProcedure();
}
```

Допустим, что процедура осмотра может быть только в отделении кардиологии, а операция – только в хирургическом. По аналогии с процедурами, создадим различные реализации фабрики для каждого отделения.

Класс CardiologyDepartment:

```
public class CardiologyDepartment implements HospitalDepartment{

    @Override
    public Procedure assignProcedure() {
        return new ExaminationProcedure();
    }
}
```

Класс SurgeryDepartment:

```
public class SurgeryDepartment implements HospitalDepartment{

    @Override
    public Procedure assignProcedure() {
        return new OperationProcedure();
    }
}
```

Далее для реализации работы абстрактной фабрики немного изменим уже существующие классы. В классе *Doctor* добавим переменную типа абстрактной фабрики, отражающую принадлежность доктора к определенному отделению. К данной переменной добавим геттеры и сеттеры, а также создадим упрощенный

конструктор класса для облегчения примера реализации паттерна. Для удобства показаны только новые части кода.

Класс Doctor:

```
public class Doctor {
	...
	private HospitalDepartment dep;
	private static int num_id;

	public Doctor(String first_name, String second_name, String patronymic, HospitalDepartment dep) {
		this.id = num_id++;
		this.first_name = first_name;
		this.second_name = second_name;
		this.patronymic = patronymic;
		this.dep = dep;
	}

	public String getFullName() {
	    return this.second_name + " " + this.first_name + " " + this.patronymic;
	}

	public HospitalDepartment getDep() {
		return dep;
	}
	public void setDep(HospitalDepartment dep) {
		this.dep = dep;
	}
}
```

Также в классе *Appointment* добавим переменную типа абстрактной процедуры, отражающую ее тип. Как и в классе *Doctor*, были добавлены дополнительные методы. Для удобства показаны только новые части кода.

Класс Appointment:

```
public class Appointment {
	...
	private Procedure procedure;
	private static int num_id;

	public Appointment(User user, Doctor doctor, Date appointment_date_time) {
		this.id = num_id++;
		this.user = user;
		this.doctor = doctor;
		this.appointment_date_time = appointment_date_time;
```

```java
			this.appointment_date_time.setYear(appointment_date_time.getYear()-1900);
			this.appointment_date_time.setMonth(appointment_date_time.getMonth()-1);
		}

		public String getFormatDate() {
			return new SimpleDateFormat("dd.MM.yyyy || H:m").format(appointment_date_time);
		}

		public Procedure getProcedure() {
			return procedure;
		}
		public void setProcedure(Procedure procedure) {
			this.procedure = procedure;
		}

		public void makingProcedure() {
			procedure.makingProcedure(this);
		}
}
```

Далее представлен класс формы назначения процедуры, в методе которого абстрактная фабрика будет определяться по принадлежности назначенного доктора к определенному отделению. С помощью нужной реализации фабрики создаем процедуру и добавляем консультацию в коллекции.

Класс AssignForm:

```java
public class AssignForm {

		public void createAppointment(Appointment appointment) {
			HospitalDepartment dept = null;
			if (appointment.getDoctor().getDep() instanceof SurgeryDepartment) {
				dept = new SurgeryDepartment();
			} if (appointment.getDoctor().getDep() instanceof CardiologyDepartment) {
				dept = new CardiologyDepartment();
			}
			appointment.setProcedure(dept.assignProcedure());
			Hospital.addAppointment(appointment);
			appointment.getUser().addAppointment(appointment);
		}
}
```

Для демонстрации результата реализации паттерна «Абстрактная фабрика» был создан дополнительный класс *Primer*, представленный ниже. В

нем создаем несколько пользователей и докторов. Далее создаем несколько консультаций и передаем их в класс формы, описанный выше. После чего находим созданные консультации по id и проверяем правильность реализации паттерна, используя метод *makingProcedure()* класса *Appointment*, который в свою очередь вызывает одноименный метод интерфейса *Procedure*, переопределенный в соответствующих классах.

Класс Primer:

```
public class Primer {


	public static void main(String[] args) {
		AssignForm assignForm = new AssignForm();

		User patient_1 = new User("Иван", "Иванов", "Иванович");
		Doctor doctor_1 = new Doctor("Виктор", "Михайлов", "Иванович", new CardiologyDepartment());
		Appointment appointment_1 = new Appointment(patient_1, doctor_1, new Date(2022, 5, 10, 13, 30));



		User patient_2 = new User("Инна", "Петрова", "Андреевна");
		Doctor doctor_2 = new Doctor("Иван", "Орлов", "Семенович", new SurgeryDepartment());
		Appointment appointment_2 = new Appointment(patient_2, doctor_2, new Date(2022, 6, 8, 10, 00));

		assignForm.createAppointment(appointment_1);
		assignForm.createAppointment(appointment_2);

		Hospital.findAppointmentById(appointment_1.getId()).makingProcedure();
		Hospital.findAppointmentById(appointment_2.getId()).makingProcedure();
	}
}
```

Далее представлен результат реализации данного паттерна.

```
Пациент: Иванов Иван Иванович
Назначенная процедура: Осмотр
Дата: 10.05.2022 || 13:30
Кардиолог: Михайлов Виктор Иванович
----------------------------


Пациент: Петрова Инна Андреевна
Дата операции: 08.06.2022 || 10:0
Хирург: Орлов Иван Семенович
----------------------------
```

Рисунок 11. Паттерны ООП: абстрактная фабрика. Результат работы программы

Уже было указано, что данный паттерн не является удачным для применения в рамках данного проекта. Таким образом, спроектированные дополнительные классы и интерфейсы служат лишь для ознакомления с «Абстрактной фабрикой» и не требуются для реализации проекта.

Паттерны ООП: прокси

«Заместитель» – паттерн, структурирующий объекты. Данный паттерн помогает решить проблемы, связанные с контролируемым доступом к объекту. В шаблоне «прокси» мы создаем объект, имеющий оригинальный объект, чтобы связать его функциональность с внешним миром [3, 11].

Для рассмотрения его реализации также добавим несколько классов и интерфейсов. В качестве примера осуществим вывод расписания докторов из заранее созданного файла.

Определим интерфейс для чтения расписания с диска.

Интерфейс TimetableInterface:

```
public interface TimetableInterface {
	String[] getTimetable();
}
```

Далее создадим класс, реализующий данный интерфейс. Класс переопределяет метод *getTimetable()*, который читает файл с диска и возвращает массив строк.

Класс Timetable:

```
public class Timetable implements TimetableInterface {

	@Override
	public String[] getTimetable() {
		ArrayList<String> list = new ArrayList<>();
		try {
			Scanner scanner = new Scanner(new FileReader(new
File("/Users/anastasia/Documents/учеба/3 курс/2 семестр/ПИАПС/timetable.txt")));
			while (scanner.hasNextLine()) {
				String line = scanner.nextLine();
				list.add(line);
			}
		} catch (IOException e) {
			System.err.println("Error:  " + e);
		}
		return list.toArray(new String[list.size()]);
	}
}
```

Теперь определим класс-заместитель *TimetableProxy* оригинального класса *Timetable*. Метод *getTimetable()* проверяет, существует ли массив расписания. Если нет, он посылает запрос для загрузки данных с диска и сохраняет результат.

Класс TimetableProxy:

```
public class TimetableProxy implements TimetableInterface {

	private TimetableInterface timetableDoctors = new Timetable();
	private String[] timetableCache = null;

	@Override
	public String[] getTimetable() {
		if(timetableCache == null) {
			timetableCache = timetableDoctors.getTimetable();
		}
		return timetableCache;
	}

	public void clearCache() {
		timetableDoctors = null;
	}
}
```

Далее создадим клиентский класс, который будет выводить само расписание. При этом обращение к методу *getTimetable()* будет происходить не через оригинальный объект *Timetable*, а через объект-заместитель *TimetableProxy*.

Класс DisplayTimetable:

```
public class DisplayTimetable {
	private TimetableInterface timetableInterface = new TimetableProxy();

	public void printTimetable() {
		String[] timetable = timetableInterface.getTimetable();
		String[] data;
		System.out.println("Доктор\t\tКабинет\tВремя приема"
				+ "\tПн\t\tВт\t\tСр\t\tЧт\t\tПт");
		for(int i = 0; i < timetable.length; i++) {
			data = timetable[i].split(";");
			System.out.printf("%s\t%s\t%s\t\t%s"
					+ "\t%s\t%s\t%s\t%s\n",
					data[0], data[1], data[2], data[3],
					data[4], data[5], data[6], data[7]);
		}
	}
}
```

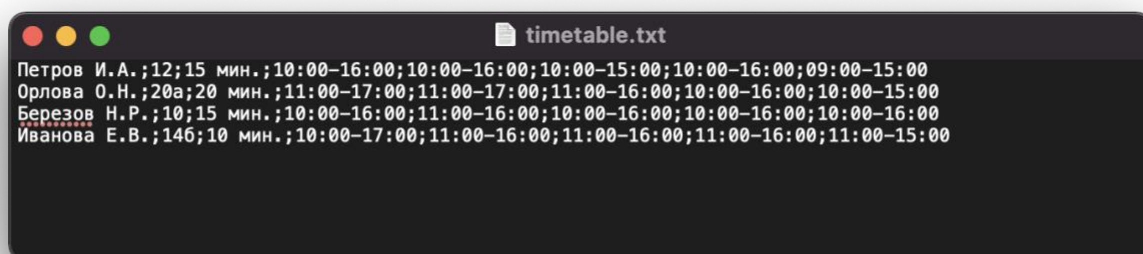
Для проверки правильности реализации работы паттерна создадим текстовый документ и тестовый класс, в котором с помощью предыдущего класса будем вызывать метод, отображающий расписание докторов.

Класс Test:

```
public class Test {

	public static void main(String[] args) {
		DisplayTimetable displayTimetable = new DisplayTimetable();
		displayTimetable.printTimetable();
	}
}
```

Ниже представлены скриншоты текстового файла, а также результат реализации паттерна.


timetable.txt

```
Петров И.А.;12;15 мин.;10:00-16:00;10:00-16:00;10:00-15:00;10:00-16:00;09:00-15:00
Орлова О.Н.;20а;20 мин.;11:00-17:00;11:00-17:00;11:00-16:00;10:00-16:00;10:00-15:00
Березов Н.Р.;10;15 мин.;10:00-16:00;11:00-16:00;10:00-16:00;10:00-16:00;10:00-16:00
Иванова Е.В.;14б;10 мин.;10:00-17:00;11:00-16:00;11:00-16:00;11:00-16:00;11:00-15:00
```

Рисунок 12. Паттерны ООП: прокси. Текстовый файл с расписанием докторов

Доктор	Кабинет	Время приема	Пн	Вт	Ср	Чт	Пт
Петров И.А.	12	15 мин.	10:00-16:00	10:00-16:00	10:00-15:00	10:00-16:00	09:00-15:00
Орлова О.Н.	20а	20 мин.	11:00-17:00	11:00-17:00	11:00-16:00	10:00-16:00	10:00-15:00
Березов Н.Р.	10	15 мин.	10:00-16:00	11:00-16:00	10:00-16:00	10:00-16:00	10:00-16:00
Иванова Е.В.	14б	10 мин.	10:00-17:00	11:00-16:00	11:00-16:00	11:00-16:00	11:00-15:00

Рисунок 13. Паттерны ООП: прокси. Результат работы программы

Осуществление данного паттерна было произведено корректно, однако можно также рассматривать вариант и с другими фабричными паттернами.

Паттерны ООП: состояние

«Состояние» – паттерн поведения объектов, задающий разную функциональность в зависимости от внутреннего состояния объекта. Позволяет объекту варьировать свое поведение в зависимости от внутреннего состояния. Поскольку поведение может меняться совершенно произвольно без каких-либо ограничений, извне создается впечатление, что изменился класс объекта [12].

Рассмотрим реализацию данного паттерна на примере изменения состояния прогресса консультации: от подачи заявки до окончания приема у врача. Для этого также расширим диаграмму классов.

Добавим интерфейс и реализующие его классы, отражающие состояние консультации: создание заявки, ее рассмотрение, отклонение или принятие, а также начало и окончание приема.

Интерфейс AppointmentState:

```
public interface AppointmentState {
	public void makingAppointment();
}
```

Класс AppointmentCreating:

```
public class AppointmentCreating implements AppointmentState{

	@Override
	public void makingAppointment() {
		System.out.println("Заявка создана...");
	}
}
```

Класс AppointmentPending:

```
public class AppointmentPending implements AppointmentState{

	@Override
	public void makingAppointment() {
		System.out.println("Заявка поступила на рассмотрение...");
	}
}
```

Класс AppointmentRejected:

```
public class AppointmentRejected implements AppointmentState{

	@Override
	public void makingAppointment() {
		System.out.println("Заявка отклонена...");
	}
}
```

Класс AppointmentApproved:

```
public class AppointmentApproved implements AppointmentState{

	@Override
	public void makingAppointment() {
		System.out.println("Заявка одобрена...");
	}
}
```

Класс AppointmentInProgress:

```
public class AppointmentInProgress implements AppointmentState{

	@Override
	public void makingAppointment() {
		System.out.println("Пациент пришел на прием...");
	}
}
```

Класс AppointmentIsOver:

```
public class AppointmentIsOver implements AppointmentState{

	@Override
	public void makingAppointment() {
		System.out.println("Прием окончен...");
	}
}
```

Далее внесем изменения в класс *Appointment*. Помимо логической переменной *isOver* (при создании объекта класса по умолчанию прием не завершен) и ее геттера, возвращающего информацию, закончен прием или нет, а также методов для отображения созданной заявки и одобренного приема, добавим атрибуты для реализации изменения его состояния.

Инициализируем атрибут *appointmentState* (при создании объекта класса по умолчанию равен объекту *AppointmentCreating*) и реализуем такой метод, как *changeAppointmentState()*. В этом методе будет определяться, представителем какого класса является упомянутый атрибут, после чего перезаписываем в него объект следующего состояния с учетом статуса заявки (одобрена или нет). Также для удобства вызова метода *makingAppointment()* интерфейса *AppointmentState*, создадим одноименный метод «обертку».

Ниже представлен фрагмент кода, содержащий только новые атрибуты и методы.

Класс Appointment:

```
public class Appointment {
	...
```

```
AppointmentState appointmentState = new AppointmentCreating();
public boolean isOver = false;

public void setAppointmentState(AppointmentState appointmentState) {
	this.appointmentState = appointmentState;
}

public boolean isOver() {
	return isOver;
}
public void setIsOver(boolean isOver) {
	this.isOver = isOver;
}



public void printCreatingAppointment() {
	System.out.println("--------------------"
			+ "\nДоктор: " + doctor.getFIO()
			+ ";\nСпециальность: " + doctor.getSpecialization()
			+ ";\nДата и время: " + getFormatDate()
			+ ";\n--------------------");
}

public void printApprovedAppointment() {
	System.out.println("--------------------"
			+ "\nПациент: " + user.getFullName()
			+ ";\nПрием назначен на: " + getFormatDate()
			+ "\nДоктор: " + doctor.getFullName()
			+ ";\nКабинет: " + doctor.getOffice()
			+ ";\n--------------------");
}

public void changeAppointmentState() {
	if (appointmentState instanceof AppointmentCreating) {
		printCreatingAppointment();
		setAppointmentState(new AppointmentPending());
	} else if (appointmentState instanceof AppointmentPending) {
		if (!this.isApproved()) {
			setAppointmentState(new AppointmentRejected());
		} else {
			setAppointmentState(new AppointmentApproved());
		}
```

```
        } else if (appointmentState instanceof AppointmentRejected) {
            isOver = true;
        } else if (appointmentState instanceof AppointmentApproved) {
            printApprovedAppointment();
            setAppointmentState(new AppointmentInProgress());
        } else if (appointmentState instanceof AppointmentInProgress) {
            setAppointmentState(new AppointmentIsOver());
        } else if (appointmentState instanceof AppointmentIsOver) {
            isOver = true;
        }
    }

    public void makingAppointment() {
        appointmentState.makingAppointment();
    }
}
```

Теперь создадим клиентский класс, показывающий работу паттерна. Создадим экземпляры классов *User*, *Doctor* и *Appointment*. С помощью сеттера передаем начальное состояние приема, в данном случае «создана заявка». И далее, пока жизненный цикл приема не закончится, просматриваем и изменяем его состояние. Также добавлена задержка, чтобы сымитировать время рассмотрения заявки и самого приема.

Ниже приведен код класса с одобренной заявкой (метод *setApproved* в качестве атрибута принимает значение *true*).

Класс TestAppointmentStates:

```
public class TestAppointmentStates {

    public static void main(String[] args) {
        User patient = new User("Олег", "Орлов", "Викторович");
        Doctor doctor = new Doctor("Екатерина", "Петрова",
                                   "Александровна", "201а", "Кардиолог");

        Appointment appointment = new Appointment(patient, doctor,
                                                  new Date(2022, 5, 22, 9, 00));
        appointment.setApproved(true);

        AppointmentState appointmentState = new AppointmentCreating();
        appointment.setAppointmentState(appointmentState);

        while(!appointment.isOver) {
            appointment.makingAppointment();
```

```
                    appointment.changeAppointmentState();
                    try {
                        Thread.sleep(1000);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }
}
```

Далее представлены скриншоты результатов работы программы при одобренной и отклоненной заявках соответственно.

```
Заявка создана...
--------------------
Доктор: Петрова Е.А.;
Специальность: Кардиолог;
Дата и время: 22.05.2022 || 09:00;
--------------------
Заявка поступила на рассмотрение...
Заявка одобрена...
--------------------
Пациент: Орлов Олег Викторович;
Прием назначен на: 22.05.2022 || 09:00
Доктор: Петрова Екатерина Александровна;
Кабинет: 201а;
--------------------
Пациент пришел на прием...
Прием окончен...
```

Рисунок 14. Паттерны ООП: состояние. Результат работы программы при одобренной заявке

```
Заявка создана...
--------------------
Доктор: Петрова Е.А.;
Специальность: Кардиолог;
Дата и время: 22.05.2022 || 09:00;
--------------------
Заявка поступила на рассмотрение...
Заявка отклонена...
```

Рисунок 15. Паттерны ООП: состояние. Результат работы программы при отклоненной заявке

В качестве примера работы паттерна «Состояние» осуществлялось изменение прогресса консультации у врача. Для этого были внесены поправки в уже существующие классы, а также добавлено несколько новых классов, представляющих собой состояние приема, что в принципе подходит для предметной области.

Паттерны ООП: стратегия

«Стратегия» – паттерн поведения объектов. Определяет семейство алгоритмов, инкапсулирует каждый из них и делает их взаимозаменяемыми. Стратегия позволяет изменять алгоритмы независимо от клиентов, которые ими пользуются [3, 13].

Реализуем данный паттерн с помощью классов, созданных при использовании шаблона «Состояние». Для этого немного изменим их и добавим в интерфейс *AppointmentState* новый метод *makingAppointment()*, однако теперь для удобства реализации он будет принимать как параметр нашу консультацию.

Интерфейс AppointmentState:

```
public interface AppointmentState {
	public void makingAppointment();
	public void makingAppointment(Appointment appointment);
}
```

Далее немного изменим классы, реализующие данный интерфейс. В зависимости от класса в новом методе также будем переопределять атрибуты консультации, такие как *isApproved* и *isOver*, то есть одобрен или закончен ли прием.

Класс AppointmentCreating:

```
public class AppointmentCreating implements AppointmentState{

	@Override
	public void makingAppointment() {
		System.out.println("Заявка создана...");
	}

	@Override
	public void makingAppointment(Appointment appointment) {
		makingAppointment();
		appointment.setIsOver(false);
	}
}
```

Класс AppointmentPending:

```
public class AppointmentPending implements AppointmentState{

	@Override
	public void makingAppointment() {
		System.out.println("Заявка поступила на рассмотрение...");
	}
```

```
	@Override
	public void makingAppointment(Appointment appointment) {
		makingAppointment();
		appointment.setIsOver(false);
	}
}
```

Класс AppointmentRejected:

```
public class AppointmentRejected implements AppointmentState{

	@Override
	public void makingAppointment() {
		System.out.println("Заявка отклонена...");
	}

	@Override
	public void makingAppointment(Appointment appointment) {
		makingAppointment();
		appointment.setApproved(false);
		appointment.setIsOver(true);
	}
}
```

Класс AppointmentApproved:

```
public class AppointmentApproved implements AppointmentState{

	@Override
	public void makingAppointment() {
		System.out.println("Заявка одобрена...");
	}

	@Override
	public void makingAppointment(Appointment appointment) {
		makingAppointment();
		appointment.setApproved(true);
		appointment.setIsOver(false);
	}
}
```

Класс AppointmentInProgress:

```
public class AppointmentInProgress implements AppointmentState{

	@Override
	public void makingAppointment() {
```

```
            System.out.println("Пациент пришел на прием...");
        }

        @Override
        public void makingAppointment(Appointment appointment) {
            if (appointment.isApproved()) {
                makingAppointment();
                appointment.setIsOver(false);
            } else {
                System.out.println("В приеме отказано...");
                appointment.setAppointmentState(new AppointmentRejected());
            }
        }
}
```

Класс AppointmentIsOver:

```
public class AppointmentIsOver implements AppointmentState{

        @Override
        public void makingAppointment() {
            System.out.println("Прием окончен...");
        }

        @Override
        public void makingAppointment(Appointment appointment) {
            if (appointment.isApproved()) {
                makingAppointment();
                appointment.setIsOver(true);
            } else {
                System.out.println("В приеме отказано...");
                appointment.setAppointmentState(new AppointmentRejected());
            }
        }
}
```

Также опять внесем изменения в класс *Appointment*. Для реализации предыдущего паттерна был добавлен атрибут *appointmentState*, а также метод-сеттер для него. Теперь создадим метод *executeAppointment()*, то есть выполнение консультации. Он похож на метод *makingAppointment()*, который был определен в предыдущем примере, и по сути также является «оберткой» для уже нового метода *makingAppointment()* интерфейса *AppointmentState*.

Ниже представлен фрагмент кода, содержащий только новые атрибуты и методы.

Класс Appointment:

```
public class Appointment {
	...
	AppointmentState appointmentState = new AppointmentCreating();

	public void setAppointmentState(AppointmentState appointmentState) {
		this.appointmentState = appointmentState;
	}
	public void executeAppointment() {
		appointmentState.makingAppointment(this);
	}
}
```

Теперь создадим клиентский класс для того, чтобы проверить работу данного паттерна. Создадим экземпляры классов *User*, *Doctor* и *Appointment*. Далее будем устанавливать новые состояния приема и вызывать метод выполнения консультации. Ниже приведен код данного класса.

Класс TestAppointmentStates:

```
public class TestAppointmentStrategy {
	public static void main(String[] args) {
		User patient = new User("Олег", "Орлов", "Викторович");
		Doctor doctor = new Doctor("Екатерина", "Петрова", "Александровна", "201а", "Кардиолог");
		Appointment appointment = new Appointment(patient, doctor, new Date(2022, 5, 22, 9, 00));

		appointment.setAppointmentState(new AppointmentCreating());
		appointment.executeAppointment();

		appointment.setAppointmentState(new AppointmentPending());
		appointment.executeAppointment();

		System.out.println("\n\n- Откажем в приеме и попытаемся \"прийти на него\" -\n");
		appointment.setAppointmentState(new AppointmentRejected());
		appointment.executeAppointment();

		appointment.setAppointmentState(new AppointmentInProgress());
		appointment.executeAppointment();

		System.out.println("\n\n- Теперь одобрим прием -\n");
		appointment.setAppointmentState(new AppointmentApproved());
		appointment.executeAppointment();

		appointment.setAppointmentState(new AppointmentInProgress());
```

```
            appointment.executeAppointment();


            appointment.setAppointmentState(new AppointmentIsOver());
            appointment.executeAppointment();
        }
}
```

Далее представлен результат работы программы.

```
Заявка создана...
Заявка поступила на рассмотрение...


– Откажем в приеме и попытаемся "прийти на него" –

Заявка отклонена...
В приеме отказано...


– Теперь одобрим прием –

Заявка одобрена...
Пациент пришел на прием...
Прием окончен...
```

Рисунок 16. Паттерны ООП: стратегия. Результат работы программы

На первый взгляд данный шаблон похож на паттерн «Состояние». Но в прошлом паттерне смена состояния происходила непосредственно в классе *Appointment*, а при использовании «Стратегии» состояние устанавливается и выполняется уже в клиентском коде, что также удобно и важно при реализации проектов подобного рода.

Паттерны ООП: легковес

«Легковес» – паттерн, структурирующий объекты. Его суть заключается в том, что если у разных объектов есть одинаковое состояние, то его можно обобщить и хранить не в каждом объекте, а в одном месте. И тогда каждый объект сможет ссылаться на общую часть, что позволит сократить расходы памяти на хранение. Часто работа данного паттерна связана с предварительным кэшированием [3, 14].

В рамках проекта довольно сложно добавить паттерн «Легковес», но как пример с помощью данного шаблона можно реализовать процедуру найма докторов по специальностям. Для этого расширим диаграмму классов проекта. Создадим несколько наследников класса *Doctor*. В конструкторах новых классов определим соответствующие специальность и отделение больницы.

Класс Surgeon:

```
public class Surgeon extends Doctor{

	public Surgeon(){
		super.setDep(new SurgeryDepartment());
		setSpecialization("Хирург");
	}
}
```

Класс Cardiologist:

```
public class Cardiologist extends Doctor{

	public Cardiologist(){
		setDep(new CardiologyDepartment());
		setSpecialization("Кардиолог");
	}
}
```

Класс Endocrinologist:

```
public class Endocrinologist extends Doctor{

	public Endocrinologist(){
		super.setDep(new CardiologyDepartment());
		setSpecialization("Эндокринолог");
	}
}
```

Также для удобства определения специальности было создано перечисление с несколькими возможными вариантами.

Перечисление Speciality:

```
public enum Speciality {
```

```
        Surgeon,
        Cardiologist,
        Endocrinologist
}
```

Теперь создадим класс, представляющий данный паттерн. Данный класс будет содержать коллекцию ключ-значений по специальности и доктору. Также у класса будет метод, позволяющий получить доктора из коллекции по его специальности. Если доктора с нужной специальностью нет в коллекции, то создаем соответствующего специалиста и добавляем в коллекцию.

Класс HospitalEmployees:

```
public class HospitalEmployees {

private static final Map<Speciality, Doctor> doctors = new HashMap<>();

        public Doctor getDoctorBySpeciality(Speciality speciality) {
                Doctor doctor = doctors.get(speciality);
                if(doctor == null) {
                        switch (speciality) {
                        case Surgeon:
                                System.out.println("Появилась должность хирурга...");
                                doctor = new Surgeon();
                                break;
                        case Cardiologist:
                                System.out.println("Появилась должность кардиолога...");
                                doctor = new Cardiologist();
                                break;
                        case Endocrinologist:
                                System.out.println("Появилась должность эндокринолога...");
                                doctor = new Endocrinologist();
                                break;
                        }
                        doctors.put(speciality, doctor);
                }
                return doctor;
        }
}
```

Также дополнительно добавим новый метод для вывода названия отделения *getDepartmentName()* в интерфейсе *HospitalDepartment* и классах, реализующих его. Приведенный ниже код интерфейса и классов для удобства содержит только новый метод.

Интерфейс HospitalDepartment:

```
public interface HospitalDepartment {
	...
	public String getDepartmentName();
}
```

Класс CardiologyDepartment:

```
public class CardiologyDepartment implements HospitalDepartment{
	...
	@Override
	public String getDepartmentName() {
		return "Кардиологическое отделение";
	}
}
```

Класс SurgeryDepartment:

```
public class SurgeryDepartment implements HospitalDepartment{
	...
	@Override
	public String getDepartmentName() {
		return "Хирургическое отделение";
	}
}
```

Ниже представлен клиентский код, реализующий работу данного паттерна. С помощью экземпляра класса *HospitalEmployees* получаем по несколько раз докторов различных специальностей и добавляем их в коллекцию докторов класса-одиночки *Hospital*. Для проверки подсчитаем количество докторов по специальностям, а также по их принадлежности к отделениям.

Класс TestAppointmentStates:

```
public class HospitalEmployeesTest {

	public static void main(String[] args) {
		HospitalEmployees hospitalEmployees = new HospitalEmployees();

	Hospital.addDoctor(hospitalEmployees.getDoctorBySpeciality(Speciality.Surgeon));
	Hospital.addDoctor(hospitalEmployees.getDoctorBySpeciality(Speciality.Surgeon));
	Hospital.addDoctor(hospitalEmployees.getDoctorBySpeciality(Speciality.Cardiologist));
	Hospital.addDoctor(hospitalEmployees.getDoctorBySpeciality(Speciality.Cardiologist));
	Hospital.addDoctor(hospitalEmployees.getDoctorBySpeciality(Speciality.Endocrinologist));
	Hospital.addDoctor(hospitalEmployees.getDoctorBySpeciality(Speciality.Endocrinologist));

		System.out.println("\n\nКоличество докторов по специальностям:\n");
```

```
        Map<String, Integer> doctorsBySpec = new HashMap<>();
        for(Doctor doctor: Hospital.getDoctors()) {
            if(doctorsBySpec.containsKey(doctor.getSpecialization())) {
                doctorsBySpec.put(doctor.getSpecialization(),
                                doctorsBySpec.get(doctor.getSpecialization()) + 1);
            } else doctorsBySpec.put(doctor.getSpecialization(), 1);
        }

        for (String key : doctorsBySpec.keySet())
            System.out.println(key + ": " +  doctorsBySpec.get(key));


        System.out.println("\n\nКоличество докторов по отделениям:\n");
        Map<String, Integer> doctorsByDep = new HashMap<>();
        for(Doctor doctor: Hospital.getDoctors()) {
            if(doctorsByDep.containsKey(doctor.getDep().getDepartmentName())) {
                doctorsByDep.put(doctor.getDep().getDepartmentName(),
                            doctorsByDep.get(doctor.getDep().getDepartmentName()) + 1);
            } else doctorsByDep.put(doctor.getDep().getDepartmentName(), 1);
        }

        for (String key : doctorsByDep.keySet())
            System.out.println(key + ": " +  doctorsByDep.get(key));
    }
}
```

Далее представлен результат работы программы.

```
Появилась должность хирурга...
Появилась должность кардиолога...
Появилась должность эндокринолога...


Количество докторов по специальностям:

Хирург: 2
Эндокринолог: 2
Кардиолог: 2


Количество докторов по отделениям:

Кардиологическое отделение: 4
Хирургическое отделение: 2
```

Рисунок 17. Паттерны ООП: легковес. Результат работы программы

Как показывает разработанный пример, данный паттерн удобен в использовании при наличии большого числа объектов, а также при

пренебрежении их идентичностью, поэтому в рамках данного проекта паттерн «Легковес» не является удачным для применения.

Заключение

В результате выполнения работы была спроектирована система, отвечающая выявленным требованиям, а также были выполнены поставленные задачи: описаны варианты использования системы, построены диаграммы классов анализа, проектных классов и пакетов, диаграмма состояний объектов системы и диаграмма последовательности.

Были созданы некоторые классы, представленные на диаграмме классов, а также реализован основной функционал системы с помощью различных методов. Для проверки корректности их работы было проведено несколько модульных и интеграционных тестов, выполненных успешно, что свидетельствует о правильности работы написанных методов.

Также для реализации системы были задействованы различные шаблоны проектирования, предложенные для рассмотрения. Особо удачными для применения в рамках данной предметной области оказались паттерны «Одиночка», «Фабричный метод», «Стратегия», а также «Медиатор». Наименее подходящими были паттерны «Абстрактная фабрика», «Прокси», «Состояние» и «Легковес», которые было достаточно сложно вписать в соответствии с выявленными требованиями к системе.

Для дальнейшего развития проекта можно рассмотреть дополнительные паттерны, такие как «Интерпретатор», «Хранитель» и пр., а также расширить и оптимизировать функционал проектируемой системы.

Список литературы и источников

1. Буч Г., Рамбо Д., Якобсон И. Язык UML. Руководство пользователя. 2-е изд.: Пер. с англ. Мухин Н. – М.: ДМК Пресс, 2006. – 496 с.: ил.
2. Draw.io | Diagrams.net [Электронный ресурс]. // URL: https://www.diagrams.net/
3. Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Приемы объектно-ориентированного проектирования. Паттерны проектирования. — СПб: Питер, 2001. — 368 с.: ил. (Серия «Библиотека программиста»)
4. StudFiles | Построение UML – модели системы. Диаграмма классов анализа. [Электронный ресурс]. // URL: https://studfile.net/preview/16480625/
5. Модульное и интеграционное тестирование. [Электронный ресурс]. // URL:https://portal.tpu.ru/SHARED/i/I/study/trpo/lab/E71E8C0825993298E0407B6D0998191A
6. Eclipse | Eclipse Foundation. [Электронный ресурс]. // URL: https://www.eclipse.org/
7. JavaRush | Паттерны проектирования: Singleton. [Электронный ресурс]. // URL: https://javarush.ru/groups/posts/2365-patternih-proektirovanija-singleton
8. YouTube | Шаблоны Java. Mediator (Посредник). [Электронный ресурс]. // URL:https://www.youtube.com/watch?v=ZnyNsrcLl2I&list=PLlsMRoVt5sTPgGbinwOVnaF1mxNeLAD7P&index=22
9. YouTube | Шаблоны Java. FactoryMethod (Фабричный метод). [Электронный ресурс]. // URL: https://www.youtube.com/watch?v=TwIjjTC5g7g&list=PLlsMRoVt5sTPgGbinwOVnaF1mxNeLAD7P&index=4
10. YouTube | Шаблоны Java. AbstractFactory (Абстрактная фабрика). [Электронный ресурс]. // URL: https://www.youtube.com/watch?v=cmyUI_ZezoU&list=PLlsMRoVt5sTPgGbinwOVnaF1mxNeLAD7P&index=5
11. JavaRush | Паттерн проектирования Proxy. [Электронный ресурс]. // URL: https://javarush.ru/groups/posts/2368-pattern-proektirovanija-proxy
12. YouTube | Шаблоны Java. State (Состояние). [Электронный ресурс]. // URL: https://www.youtube.com/watch?v=gpY98f7A_8M&list=PLlsMRoVt5sTPgGbinwOVnaF1mxNeLAD7P&index=25
13. YouTube | Шаблоны Java. Strategy (Стратегия). [Электронный ресурс]. // URL:https://www.youtube.com/watch?v=rsB2exGsR4I&list=PLlsMRoVt5sTPgGbinwOVnaF1mxNeLAD7P&index=26
14. YouTube | Шаблоны Java. Flyweight (Приспособленец). [Электронный ресурс]. // URL: https://www.youtube.com/watch?v=d9Cx-2LSGNA&list=PLlsMRoVt5sTPgGbinwOVnaF1mxNeLAD7P&index=15