

МИНИСТЕРСТВО науки и высшего ОБРАЗОВАНИЯ РОссИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ

УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ

(национальный исследовательский университет)»

Кафедра 319 «Системы интеллектуального мониторинга»

**КУРСОВАЯ РАБОТА**

по дисциплине

«Объектно-ориентированное программирование»

Тема:

Игра «Шахматы»

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Студент | | |  | Калёнов В.И. |
| Группа | | |  | М3О-235Б-22 |
| Проверил | | |  | Поречный А.С. |
| Оценка |  | Дата защиты « \_ » 2023 г. | | |

**МОСКВА 2023**

МИНИСТЕРСТВО науки и высшего ОБРАЗОВАНИЯ РОссИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ

УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ

(национальный исследовательский университет)»

Кафедра 319 «Системы интеллектуального мониторинга»

**З А Д А Н И Е**

на курсовую работу по дисциплине

«Объектно-ориентированное программирование»

Студент \_\_\_\_\_\_\_\_М3О-235Б-22 Калёнов Вячеслав Игоревич\_\_\_\_\_\_\_\_\_\_

(№ группы, Ф. И. О.)

Тема \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Игра «Шахматы»\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Перечень вопросов, подлежащих разработке в курсовой работе

Реализация алгоритма на языке Java с использованием принципов ООП.\_\_\_\_\_\_\_\_\_\_\_\_\_

Чтение и разработка UML-диаграмм.\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Чтение и разработка дерева функций\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Разработка приложения с использованием системой управления версиями\_\_\_\_\_\_\_\_\_\_\_\_

Рекомендуемая литература

1. Шилдт, Г. Java. Полное руководство, 10-е изд.: Пер. с англ. – СПб. ООО "Альфакнига', 2018. – 1488 с. ISBN 978-5-6040043-6-4 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

2. Мейер, Б. Объектно-ориентированное конструирование программных систем: Русская Редакция; 2015. – 768 с. ISBN 5-7502-0255-0\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

3. Лафоре, Р. Структуры данных и алгоритмы в Java. / Классика Computers Science. 2-е изд. – СПб.:Питер, 2013. – 704 с. ISBN 978-5-496-007405 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

4. Мартин, Р. Чистый код: создание, анализ и рефакторинг. / Библиотека программиста. – СПб.: Питер, 2015. – 464 с. ISBN 978-5-496-00487-9\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

5. Буч, Г., Рамбо, Д., Якобсон, И. Язык UML. / Руководство пользователя, М.:ДМК Пресс, 2006. – 496с. ISBN 5-94074-334-X\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Задание выдано «22» сентября 2023 г.

Проверил \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*Поречный А.С.* \_\_\_\_

(Ф. И. О., подпись)

Студент \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(подпись)

Содержание

[СОДЕРЖАНИЕ 3](#_Toc5585)

[ВВЕДЕНИЕ 5](#_Toc30847)

[1. ТЕОРЕТИЧЕСКАЯ ЧАСТЬ 7](#_Toc24982)

[1.1. Дерево функций 7](#_Toc30111)

[1.2. UML диаграммы 7](#_Toc907)

[1.2.1. Разновидности UML диаграмм 7](#_Toc5086)

[1.2.2. Виды связи в UML диаграмме классов 8](#_Toc11280)

[1.3. SQL. Виды SQL. Способы нормализации баз данных 10](#_Toc25003)

[1.3.1. Описание базы данных 10](#_Toc31032)

[1.3.2. Нормализация баз данных 11](#_Toc12449)

[1.3.3. Виды отношений между таблицами 12](#_Toc23208)

[1.3.4. Описание языка SQL 13](#_Toc19585)

[1.4. Система управления версиями 14](#_Toc17690)

[1.4.1. Виды систем управления версиями 14](#_Toc22290)

[1.4.2. Стратегии слияния разных версий кода 14](#_Toc15181)

[1.4.3. Основные команды Git 15](#_Toc22503)

[1.5. Объектно-ориентированное программирование 16](#_Toc7980)

[1.5.1. Абстракция 16](#_Toc18258)

[1.5.2. Инкапсуляция 17](#_Toc5486)

[1.5.3. Полиморфизм 18](#_Toc22043)

[1.5.4. Наследование 18](#_Toc22277)

[2. ОСНОВАНАЯ ЧАСТЬ 20](#_Toc32681)

[2.1. Основные части кода 20](#_Toc26373)

[2.1.1. Класс Piece 20](#_Toc8900)

[2.1.2. Класс Board 21](#_Toc6398)

[2.1.3. Класс Game 23](#_Toc32027)

[2.1.4. Класс CheckmateGameStateChecker 24](#_Toc29397)

[2.1.5. Класс Database 24](#_Toc2089)

[2.2. Дерево функций 31](#_Toc3735)

[2.3. UML диаграмма последовательностей 31](#_Toc27196)

[2.4. Диаграмма классов 32](#_Toc1007)

[3. ТЕСТИРОВАНИЕ 33](#_Toc1185)

[3.1. Описание Junit 33](#_Toc24087)

[3.2. Тестовые классы 33](#_Toc12357)

[3.2.1. Класс BoardTest 33](#_Toc2683)

[3.2.2. Класс CheckmateGameCheckerTest 35](#_Toc1098)

[3.2.3. DatabaseTest 38](#_Toc19827)

[3.2.4. Класс PiecesAvailableMovesTest 40](#_Toc7782)

[3.3. Результат тестирования 46](#_Toc30133)

[ЗАКЛЮЧЕНИЕ 47](#_Toc25342)

[СПИСОК ЛИТЕРАТУРЫ 48](#_Toc3185)

[ПРИЛОЖЕНИЕ А 49](#_Toc7006)

[ПРИЛОЖЕНИЕ Б 51](#_Toc28470)

[ПРИЛОЖЕНИЕ В 52](#_Toc19075)

[ПРИЛОЖЕНИЕ Г 53](#_Toc27149)

[ПРИЛОЖЕНИЕ Д 54](#_Toc5450)

[ПРИЛОЖЕНИЕ Е 55](#_Toc23413)

[ПРИЛОЖЕНИЕ Ж 56](#_Toc16642)

[ПРИЛОЖЕНИЕ З 57](#_Toc31004)

Введение

Объектно-ориентированное программирование представляет собой методологию разработки программных систем, основанную на концепции объектов и классов. Основной принцип ООП заключается в том, что программа строится из отдельных объектов, каждый из которых является экземпляром определенного класса и имеет свои собственные свойства и методы.

ООП позволяет абстрагировать сложные системы, повышает их удобство в использовании, облегчает поддержку и расширение кода. Введение ООП в программирование привнесло новые возможности для создания модульных, гибких и масштабируемых приложений.

Java является объектно-ориентированным языком программирования. В Java всё является объектом, за исключением примитивных типов данных. Классы в Java используются для создания объектов, которые содержат данные и методы для их обработки. Наследование позволяет создавать новые классы на основе существующих, что упрощает повторное использование кода и уменьшает его дублирование.

Java также поддерживает инкапсуляцию, что означает, что данные объекта могут быть скрыты от внешнего доступа и доступны только через методы класса. Это обеспечивает безопасность данных и упрощает использование объектов.

Полиморфизм в Java позволяет объектам иметь различное поведение в зависимости от контекста. Например, один и тот же метод может иметь разную реализацию в разных классах.

База данных — это организованная и структурированная совокупность данных, которая хранится и управляется с помощью специализированной программы — системы управления базами данных (СУБД). Базы данных предлагают надежное и эффективное решение для хранения, управления и доступа к данным. Они обеспечивают централизованное хранилище для хранения информации и предлагают гибкость и масштабируемость при обработке данных.

Важным аспектом баз данных является их структура. Они могут использовать различные модели организации данных, такие как реляционная модель, иерархическая модель, сетевая модель и другие. Каждая модель имеет свои особенности и подходит для определенных типов приложений.

Git является распределенной системой управления версиями, которая используется для отслеживания изменений в коде и совместной работы над проектами. Он позволяет разработчикам сохранять историю изменений, вносить исправления и сливать код с другими участниками команды. Git является очень популярным средством управления версиями и широко используется в различных областях разработки программного обеспечения.

UML, или Unified Modeling Language, является стандартизированным языком для визуализации, проектирования и документирования системного проектирования. UML диаграммы представляют собой графическое представление различных аспектов программных систем, таких как их структура, поведение, взаимодействие и архитектура. Они упрощают понимание сложных системных концепций и обеспечивают эффективное взаимодействие между разработчиками, аналитиками и заказчиками. UML диаграммы широко используются в разработке программного обеспечения для моделирования и анализа требований, проектирования архитектуры и создания документации.

В данной работе рассматривается ООП проект «шахматы». В проекте используются следующие технологии: распределенная система контроля версий Git; встроенная СУБД PostgreSQL; стандартизированный язык для визуализации UML; язык программирования Java. С помощью данных технологий реализуется основная логика, которая отражает правила игры. Правила Игры представлены в приложении А.

1. Теоретическая часть
   1. Дерево функций

Дерево функций для проектов — это структурированное иерархическое представление всех функций или возможностей, которые должны быть реализованы в проекте. Оно помогает разработчикам лучше понять общую картину проекта, определить зависимости между функциями, а также управлять задачами и ресурсами.

Дерево функций обычно строится по следующим принципам:

1. Разделение на уровни. Функции проекта группируются по уровням, где на верхнем уровне находятся общие функции, а более нижние уровни содержат более детализированные функции.

2. Древовидная структура. Функции организуются в виде древовидной структуры, где каждая функция имеет связи с более общими и более детализированными функциями.

3. Декомпозиция. Функции разбиваются на более мелкие подзадачи, чтобы каждая из них могла быть реализована отдельно, что упрощает управление и контроль над проектом.

Разработка дерева функций начинается с выявления общих целей проекта, затем определения основных функций, которые способствуют достижению этих целей, и дальнейшего их дробления на более мелкие части.

Дерево функций является инструментом управления проектами, поскольку оно помогает установить структуру проекта, определить требования к проекту и понять взаимосвязи между различными функциями.

* 1. UML диаграммы
     1. Разновидности UML диаграмм

UML (Unified Modeling Language) применяется в разработке программного обеспечения для визуализации, проектирования и документирования систем. UML диаграммы представляют собой графические модели, которые позволяют разработчикам и аналитикам увидеть структуру и поведение системы.[4]

Существует несколько разновидностей UML диаграмм, каждая из которых предназначена для отображения определенных аспектов системы:

1. UML-диаграмма классов отображает структуру системы, включая классы, их атрибуты и методы, а также связи между классами.
2. UML-диаграмма прецедентов описывает функциональность системы через взаимодействие актеров и системы.
3. UML-диаграмма последовательностей показывает взаимодействие различных объектов в системе в определенной последовательности.
4. UML-диаграмма состояний отображает различные состояния, в которых может находиться объект системы, и переходы между этими состояниями.
5. UML-диаграмма компонентов позволяет увидеть структуру программных компонентов и их взаимодействие.
6. UML-диаграмма развертывания описывает конфигурацию физических аппаратных средств, на которых работает система, и их взаимодействие.

Каждая из этих диаграмм обладает своими специфическими возможностями и используется для визуализации определенных аспектов системы. В совокупности они обеспечивают полное представление о структуре и поведении системы, что делает их незаменимым инструментом при разработке программного обеспечения.

* + 1. Виды связи в UML диаграмме классов

В UML (Unified Modeling Language) диаграмме классов можно представить различные виды связей между классами, которые отражают взаимодействие и зависимости между объектами системы. Вот несколько основных видов связей в UML диаграмме классов:

1. Наследование (Inheritance). Это отношение между двумя классами, когда один класс (подкласс или производный класс) наследует атрибуты и методы от другого класса (родительского класса). Наследование обозначается стрелкой с открытым треугольником, указывающим на родительский класс.

2. Ассоциация (Association). Это отношение, которое указывает на связь между объектами двух классов. Ассоциация обозначается линией, соединяющей два класса, с возможными метками и мультипликацией.

3. Агрегация (Aggregation). Это более слабая форма ассоциации, которая указывает, что один класс является составной частью другого класса, но может существовать отдельно. Агрегация обозначается ромбом на линии, соединяющей два класса.

4. Композиция (Composition). Это более сильная форма агрегации, которая указывает, что один класс является частью другого класса и не может существовать отдельно. Композиция обозначается заливкой ромба на линии, соединяющей два класса.

5. Реализация в UML обозначает отношение между двумя элементами модели, где один элемент (обычно класс или интерфейс) реализует функциональность, определенную другим элементом (обычно интерфейсом). Это отношение показывает, что класс или компонент предоставляет реализацию методов или операций, определенных в интерфейсе. В диаграмме классов связь реализация обозначается пунктирной линией с треугольником на конце.

6. Связь зависимость в UML обозначает отношение между двумя элементами модели, где один элемент зависит от другого элемента в том смысле. Это отношение показывает, что один элемент использует функциональность или информацию, предоставляемую другим элементом. В диаграмме классов связь зависимость обычно обозначается пунктирной линией с открытой стрелкой.

Графическое представление видов связи представлено на рисунке 1.

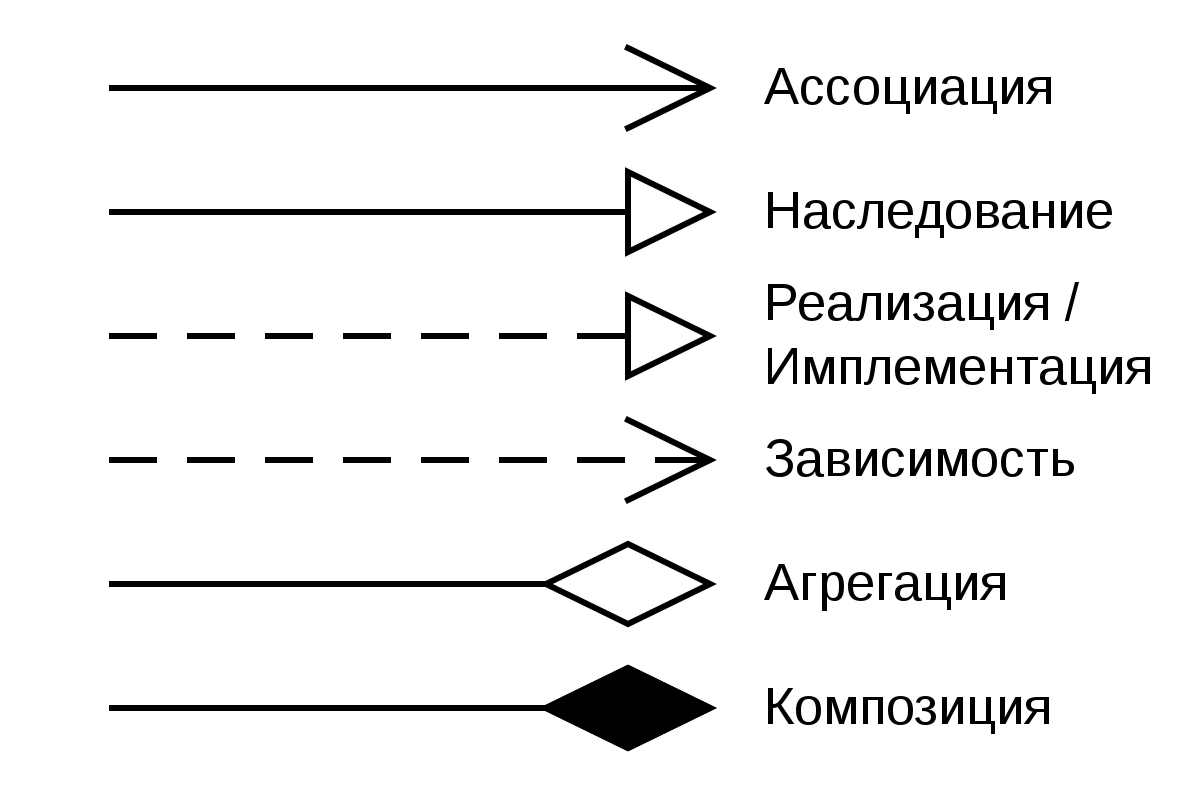


Рисунок 1 — Графическое представление связей в диаграмме классов

* 1. SQL. Виды SQL. Способы нормализации баз данных
     1. Описание базы данных

Базы данных — это организованный набор данных.

СУБД — система управления базами данных. Это специализированная ПО, призванная управлять базами данных. Достигается это взаимодействия пользователями с одной стороны и базой с другой.

СУБД должна позволять: определять, создавать, администрировать и выполнять запросы к базам данных.

Базы данных делятся на два вида: реляционные и нереляционные.

Реляционные модель оперирует понятиями «запись», атрибуты и «отношения».

Отношения — таблицы, атрибуты — названия столбцов. Запись — строка, значения в таблице.

РСУБД (реляционная СУБД) требует строгое определение структуры данных, храниых в базе. Т.е. отношения и атрибуты фиксированы.

Задачи, выполняемые СУБД:

1. Определение схемы данных (создание, изменение, удаление структуры);
2. Изменение данных — (добавление, изменение, удаление);
3. Получение данных — представление информации в форме пригодной к непосредственному использованию;
4. Администрирование БД — регистрация и управление пользователями, обеспечение безопасности данных, поддержание целостности, управление одновременным доступом, мониторинг за производительностью.

Определения, использующиеся в базах данных:

* Суперключ — такое множество атрибутов, которое удовлетворяет требованию уникальности, т.е. не существует записи с повторяющимися значениями. Важно отметить, что для соотношения с какими-то атрибутами, существует хотя бы один суперключ.
* Минимальный набор из всех атрибутов, от которых функционально зависит все атрибуты отношения, будут называться потенциальным ключом.
* Потенциальный ключ — это множество атрибут A, которое удовлетворяет требованиям уникальности и минимальности, т.е. не существует записей, с повторяющимися значениями A, и не существует такого строгого подмножества B, которое входит в A, которое могло бы удовлетворять требование уникальности.
* Ключевые атрибуты — это атрибуты, которые входят в потенциальный ключ. Неключевые атрибуты — не входят ни в один потенциальный ключ.
* Внешний ключ — набор атрибутов A отношения R называется внешним ключом, если тот же набор атрибутов A либо некое переименование A является супер ключом некого другого отношения S, причём множеством значений A по всем записям R в любой момент времени является подмножеством значений переименованного A по всем записям S.
  + 1. Нормализация баз данных

Способы нормализации базы данных представляют собой набор методов, направленных на улучшение структуры и производительности базы данных, устранение избыточности данных и повышение целостности информации. Нормализация базы данных обеспечивает удобство работы с данными, предотвращает ошибки и упрощает выполнение запросов.

Основными способами нормализации базы данных являются:

1. Первая нормальная форма (1NF): в этой форме каждый атрибут в таблице должен быть атомарным, то есть не должен иметь повторяющихся значений или быть составным.

2. Вторая нормальная форма (2NF): данные должны быть в 1NF, и каждый неключевой атрибут должен зависеть от ключа целиком, а не только от его части.

3. Третья нормальная форма (3NF): данные должны быть в 2NF, и неключевые атрибуты не должны зависеть друг от друга.

4. Нормальная форма Бойса-Кодда (BCNF): всегда следует приводить таблицы к BCNF, если они находятся в 3NF и не соответствуют этой форме.

5. Четвертая нормальная форма (4NF): атрибуты, которые зависят только от комбинаций значений других атрибутов, должны быть вынесены в отдельные таблицы.

6. Пятая нормальная форма (5NF): дополнительная нормализация для сложных многозначных зависимостей.

Использование этих методов позволяет сделать базу данных более эффективной, снизить избыточность данных и предотвратить появление аномалий при работе с информацией. Однако, важно учитывать, что излишняя нормализация может привести к более сложной структуре базы данных и затруднить выполнение некоторых запросов. Поэтому, при нормализации необходимо находить баланс между структурой данных и требованиями логики.

* + 1. Виды отношений между таблицами

В базах данных существует несколько видов отношений между таблицами, которые позволяют организовать данные и связи между ними. Вот некоторые из основных видов отношений:

1. Один к одному (1:1). В этом виде отношений каждая запись в одной таблице связана с одной записью в другой таблице. Это отношение используется для разделения сущности на две таблицы с разными аспектами.

2. Один ко многим (1:М). Это отношение, при котором одна запись в одной таблице связана с несколькими записями в другой таблице. Например, один автор может иметь много книг.

3. Многие ко многим (М:М). Этот вид отношений возникает, когда несколько записей в одной таблице связаны с несколькими записями в другой таблице. Для реализации такого отношения используется промежуточная таблица, которая связывает записи из обеих таблиц.

1.3.4. Описание языка SQL

SQL — это декларативный язык структурированных запросов (Structured Query Language), позволяющий хранить, манипулировать и извлекать данные из реляционных баз данных (далее — РБД, БД).

SQL позволяет:

* получать доступ к данным в системах управления РБД;
* описывать данные (их структуру);
* определять данные в БД и управлять ими;
* взаимодействовать с другими языками через модули SQL, библиотеки и предварительные компиляторы;
* создавать и удалять БД и таблицы;
* создавать представления, хранимые процедуры (stored procedures) и функции в БД;
* устанавливать разрешения на доступ к таблицам, процедурам и представлениям.

Основные SQL команды выглядят следующим образом:

* SELECT — выбор данных из базы данных.
* INSERT — добавление новых данных в базу данных.
* UPDATE — обновление существующих данных в базе данных.
* DELETE — удаление данных из базы данных.
* CREATE — создание новой таблицы, индекса или представления
* ALTER — изменение структуры существующей таблицы.
* DROP — удаление таблицы, индекса или представления.
* TRUNCATE — удаление всех записей из таблицы.
* JOIN — объединение данных из разных таблиц.
* GROUP BY — группировка данных по определенному критерию.
* ORDER BY — сортировка данных по определенному критерию.
* WHERE — фильтрация данных по определенному условию.
* HAVING — фильтрация группированных данных по определенному условию.
  1. Система управления версиями
     1. Виды систем управления версиями

Существует несколько типов систем управления версиями (СУВ), каждый из которых имеет свои особенности и предназначение. Вот некоторые из наиболее распространенных видов СУВ:

1. Локальные СУВ.
2. Распределенные СУВ.
3. Централизованные СУВ.

Каждая из этих систем управления версиями имеет свои преимущества и недостатки, и выбор СУВ зависит от потребностей проекта и предпочтений разработчиков.

* + 1. Стратегии слияния разных версий кода

Стратегии слияния различных версий кода включают в себя несколько подходов, которые позволяют эффективно управлять изменениями и объединять различные ветки разработки. Вот некоторые из них:

1. Слияние с предварительным уведомлением (Pre-Notification Merge): Этот подход включает предварительное уведомление всех разработчиков о намерении объединить ветки кода. Он даёт возможность разработчикам изложить свои планы по отношению к своим изменениям и предоставить время для обсуждения и получения обратной связи.
2. Регулярные слияния (Regular Merging): В этой стратегии разработчики регулярно заключают соглашения о слиянии своих изменений с главной веткой кода. Это обеспечивает регулярное добавление новых функций и исправлений ошибок в основную ветку, минимизируя разрывы между разными версиями кода.
3. Ручное слияние (Manual Merging): В случаях, когда автоматическое слияние может привести к конфликтам или ошибкам, требуется ручное вмешательство. Разработчики должны вручную анализировать конфликты и принимать решения о выборе наилучшего решения при объединении разных версий кода.
4. Трекинг изменений (Change Tracking): Этот подход позволяет разработчикам отслеживать все изменения, вносимые в код, и определять, какие изменения должны быть включены в основную ветку кода. Это помогает разработчикам более эффективно контролировать изменения и предотвращать возможные конфликты и проблемы слияния.

Выбор конкретной стратегии слияния зависит от потребностей и ограничений конкретного проекта. Однако все эти подходы направлены на обеспечение более эффективного процесса слияния различных версий кода.

* + 1. Основные команды Git
* «git init» — создание нового репозитория
* «git clone» — клонирование репозитория
* «git add» — добавление файлов в индекс
* «git commit» — сохранение изменений в репозитории
* «git push» — отправка изменений на удаленный сервер
* «git pull» — получение изменений с удаленного сервера
* «git branch» — создание, удаление и переключение между ветками
* «git merge» — объединение двух веток
* «git status» — проверка статуса файлов в репозитории
* «git log» — просмотр истории изменений в репозитории
  1. Объектно-ориентированное программирование
     1. Абстракция

Абстракция в объектно-ориентированном программировании (ООП) означает создание модели объекта, которая скрывает детали его реализации и предоставляет только необходимую информацию для работы с ним. [2]

В контексте ООП абстракция позволяет создавать классы, которые представляют абстрактные концепции или объекты, независимо от их конкретной реализации.

Абстракция также позволяет скрывать детали реализации от пользователя класса, предоставляя только определенные методы для взаимодействия с объектом.

Одним из способов реализации абстракции в ООП является использование абстрактных классов и интерфейсов, которые определяют только общие свойства и методы, но не содержат их реализации. Таким образом, абстракция является одним из ключевых принципов ООП, позволяющим упростить проектирование и разработку программного обеспечения.

Интерфейсы и абстрактные классы — это два способа реализации абстракции в Java.

Абстрактный класс — это класс, который не может быть инстанциирован, то есть создан как объект. Он может содержать как обычные методы, так и абстрактные методы, которые не имеют тела и должны быть реализованы в подклассах. Абстрактные классы могут содержать конструкторы, поля и методы, и их можно использовать для общей реализации поведения для нескольких подклассов.

Интерфейс — это абстрактный тип данных, который определяет методы, но не предоставляет их реализацию. В интерфейсе могут быть только абстрактные методы. Класс может реализовать несколько интерфейсов.

Основное различие между абстрактным классом и интерфейсом заключается в том, что класс может наследоваться только от одного абстрактного класса, но реализовать несколько интерфейсов. Кроме того, абстрактный класс может содержать реализацию методов, в то время как интерфейс — нет.

* + 1. Инкапсуляция

Инкапсуляция в объектно-ориентированном программировании (ООП) — это принцип, позволяющий скрыть детали реализации объекта от внешнего воздействия и предоставить доступ только к необходимым данным и методам.

В контексте классов и объектов, инкапсуляция означает объединение данных (полей) и методов внутри класса и предоставление контролируемого доступа к этим данным извне. Это достигается путем объявления полей класса как приватных или защищенных, а также предоставлением методов доступа — геттеров и сеттеров, которые позволяют управлять этими данными.

Инкапсуляция позволяет сокрыть внутренние данные и методы от изменений извне, предотвращает случайные изменения и обеспечивает безопасный доступ к данным. Также она способствует повторному использованию кода, поскольку изменение внутренней реализации класса не затрагивает другие части программы, использующие этот класс.

Инкапсуляция в Java достигается с помощью модификаторов доступа. В Java существует четыре модификатора доступа: public, private, protected и default (или package-private).[1]

* Public: классы, методы и переменные, объявленные с модификатором доступа public, могут быть доступны для любого другого класса.
* Private: классы, методы и переменные, объявленные с модификатором доступа private, могут быть доступны только внутри того же класса. Они недоступны длядругих классов.
* Protected: классы, методы и переменные, объявленные с модификатором доступа protected, могут быть доступны внутри того же пакета или из подклассов, даже если они находятся в другом пакете.
* Default (или package-private): если класс, метод или переменная не имеют явного модификатора доступа, то их доступ ограничен только теми классами, которые находятся в том же пакете.
  + 1. Полиморфизм

Полиморфизм в объектно-ориентированном программировании (ООП) означает способность объектов различных классов отвечать на одни и те же запросы с различными поведениями. Это позволяет использовать один и тот же метод или операцию для объектов разных классов без необходимости явного указания типа объекта.

Полиморфизм может быть достигнут через наследование, где дочерний класс переопределяет методы родительского класса, а также через интерфейсы или абстрактные классы, которые предоставляют общий интерфейс для различных классов.

Перегрузка один из методов реализации полиморфизма. Перегрузка в Java — это возможность определения нескольких методов с одинаковым именем в одном классе, но с разными параметрами. При этом каждый из перегруженных методов должен иметь уникальную сигнатуру, которая определяется типами и количеством параметров.

Когда вызывается перегруженный метод, компилятор Java определяет, какой именно метод должен быть вызван, на основе переданных аргументов. Если не удается найти точное соответствие, то компилятор пытается выполнить автоматическое преобразование типов данных для выбора подходящего метода.

Перегрузка методов позволяет использовать одно и то же имя метода для реализации различных действий в зависимости от переданных параметров.

* + 1. Наследование

Наследование в Java — это механизм, который позволяет одному классу наследовать свойства и методы другого класса. Класс, который наследует свойства и методы, называется подклассом, а класс, от которого наследуют, называется суперклассом.

При использовании наследования подкласс получает доступ ко всем открытым свойствам и методам суперкласса. Подкласс также может переопределить методы суперкласса, чтобы изменить их поведение в соответствии с требованиями подкласса.

Наследование позволяет создавать иерархию классов, где более общие свойства и методы выносятся в суперкласс, а более специфичные — в подклассы. Это упрощает повторное использование кода и делает программу более структурированной и легко поддерживаемой.

В Java наследование реализуется с помощью ключевого слова «extends». Класс-наследник (или подкласс) расширяет базовый класс (или суперкласс), заимствуя его поля, методы и другие свойства. в Java поддерживается только одиночное наследование, то есть класс может наследовать только от одного класса. Взаимодействия между родительскими классами и наследниками осуществляется с помощью ключевых слов «this» и «super».

1. Основная часть
   1. Основные части кода
      1. Класс Piece

Абстрактный класс фигуры необходим для выстраивания иерархии наследования. Абстрактный класс Piece определяет основной функционал у каждой фигуры, предоставляя возможность добавить собственную реализацию каждого метода. Каждая фигура обладает двумя полями: цветом и координатами, которые определяют положение фигуры. Далее рассматривается основной функционал, который присутствует у каждой фигуры.

Метод getAvailableMoveSquares(Board board) позволяет получить список клеток, на которые фигура может переместиться. Для этого на вход методу поступает доска, на которой представлена шахматная ситуация. Полная реализация метода показана в листинге 2.1.

Листинг 2.1 — Реализация метода getAvailableMoveSquares(Board board)

public Set<Coordinates> getAvailableMoveSquares(Board board) {  
 Set<Coordinates> result = new HashSet<>();  
 for (CoordinatesShift shift: getPieceMoves()) {  
 if (coordinates.canShift(shift)) {  
 Coordinates newCoordinates = coordinates.shift(shift);  
 if (isSquareAvailableForMove(newCoordinates, board)) {  
 result.add(newCoordinates);  
 }  
 }  
 }  
 return result;  
}

Метод isSquareAvailableForMove(Coordinates coordinates, Board board) позволяет проверить возможен ли ход фигуры на определенную клетку. Метод возвращает булевское значение, которое отражает возможность или невозможность хода. Полная реализация метода показана в листинге 2.2.

Листинг 2.2 — Реализация метода isSquareAvailableForMove

protected boolean isSquareAvailableForMove(Coordinates coordinates, Board board) {  
 return board.isSquareEmpty(coordinates) || board.getPiece(coordinates).color != color;  
}

Метод Set<CoordinatesShift> getPieceMoves() является абстрактным. Каждая фигура имеет свою реализацию данного метода относительно своей логики.

Метод Set<CoordinatesShift> getPieceAttacks() возвращает возможные атаки для фигуры на доске, для этого вызывается метод getPieceMoves().

Метод Set<Coordinates> getAttackedSquares(Board board) возвращает клетки, которые атакованы фигурой. Полная реализация метода показа в листинге 2.3.

Листинг 2.3 — Реализация метода getAttackedSquares

public Set<Coordinates> getAttackedSquares(Board board) {  
 Set<CoordinatesShift> pieceAttacks = getPieceAttacks();  
 Set<Coordinates> result = new HashSet<>();  
 for (CoordinatesShift pieceAttack : pieceAttacks) {  
 if (coordinates.canShift(pieceAttack)) {  
 Coordinates shiftedCoordinates = coordinates.shift(pieceAttack);  
 if (isSquareAvailableForAttack(shiftedCoordinates, board)) {  
 result.add(shiftedCoordinates);  
 }  
 }  
 }  
 return result;  
}

* + 1. Класс Board

Класс Board необходим для реализации шахматной доски, на которой расставляются фигуры. Для этого используется структура данных HashMap[3].

Далее рассматриваются основные методы, которые позволяют реализовать логику игры.

Метод isSquareDark(Coordinates coordinates) возвращает цвет клетки. Полная реализация метода показана в листинге 2.4.

Листинг 2.4 — Реализация метода isSquareDark

public static boolean isSquareDark(Coordinates coordinates) {  
 return (((coordinates.file.ordinal() + 1) + coordinates.rank) % 2) == 0;  
}

Метод isSquareEmpty(Coordinates coordinates) проверяет является ли клетка пустой. Полная реализация показана в листинге 2.5.

Листинг 2.5 — Реализация метода isSquareEmpty

public boolean isSquareEmpty(Coordinates coordinates) {  
 return !pieces.containsKey(coordinates);  
}

Метод isSquareAttackedByColor(Coordinates coordinates, Color color) проверяет, находится ли клетка под атакой фигур определенного цвета. Полная реализация показана в листинге 2.6.

Листинг 2.6 — Реализация метода isSquareAttackedByColor

public boolean isSquareAttackedByColor(Coordinates coordinates, Color color) {  
 List<Piece> pieces = getPiecesByColor(color);  
 for (Piece piece : pieces) {  
 Set<Coordinates> attackedSquares = piece.getAttackedSquares(this);  
  
 if (attackedSquares.contains(coordinates)) {  
 return true;  
 }  
 }  
 return false;  
}

Метод getPiecesByColor(Color color) возвращает все фигуры определенного цвета. Полная реализация показана в 2.7.

Листинг 2.7 — Реализация метода getPiecesByColor

public List<Piece> getPiecesByColor(Color color) {  
 List<Piece> result = new ArrayList<>();  
 for (Piece piece : pieces.values()) {  
 if (piece.color == color) {  
 result.add(piece);  
 }  
 }  
 return result;  
}

Метод makeMove(Move move) осуществляет движение фигур по доске. Полная реализация показана в листинге 2.8.

Листинг 2.8 — Реализация метода makeMove

public void makeMove(Move move) {  
 Piece piece = getPiece(move.from);  
  
 removePiece(move.from);  
 setPiece(move.to, piece);  
  
 moves.add(move);  
}

Метод getPiece(Coordinates coordinates) позволяет получить фигуру по координатам. Полная реализация показана в листинге 2.9.

Листинг 2.9 — Реализация метода getPiece

public Piece getPiece(Coordinates coordinates) {  
 return pieces.get(coordinates);  
}

Метод removePiece(Coordinates coordinates) позволяет удалить фигуру с доски. Полная реализация показана в листинге 2.10.

Листинг 2.11 — Реализация метода removePiece

public void removePiece(Coordinates coordinates) {  
 pieces.remove(coordinates);  
}

Метод setPiece(Coordinates coordinates, Piece piece) позволяет установить фигуру на определенные координаты. Полная реализация показана в листинге 2.12.

Листинг 2.12 — Реализация метода setPiece

public void setPiece(Coordinates coordinates, Piece piece) {  
 piece.coordinates = coordinates;  
 pieces.put(coordinates, piece);  
}

* + 1. Класс Game

Метод playOrCheckHistory() реализует разделение программы на два состояния: играть и просматривать историю игры, которая была сохранена в базу данных. Полная реализация метода показана в приложении Б.

Метод checkHistory() осуществляет просмотр ходов партии по введенному id партии. Также этот метод предоставляет возможность воспроизведения партии по ходам. Полная реализация метода показана в приложении В.

Метод gameLoop() осуществляет саму игру в шахматы. Для этого в реализации используется отдельный класс для ввода и методы для работы с доской, которые были описаны выше. Полна реализация метода показана в приложении Г.

Метод determineGameState(Board board, Color color) используется для определения состояния доски: присутствует матовая или патовая ситуация. Для этого используется класс GameStateChecker. Полная реализация метода показана в листинге 2.13.

Листинг 2.13 — Реализация метода determineGameState

private GameState determineGameState(Board board, Color color) {  
for (GameStateChecker checker : checkers) {  
 GameState state = checker.check(board, color);  
 if (state != GameState.ONGOING) {  
 return state;  
 }  
}  
return GameState.ONGOING;

* + 1. Класс CheckmateGameStateChecker

Класс CheckmateGameStateChecker наследуется от GameStateChecker. В классе CheckmateGameStateChecker реализуется один метод для проверки состояния доски. Полная реализация метода check(Board board, Color color) показана в приложении Д.

* + 1. Класс Database

Для подключения базы данных к проекту, необходимо её создать, используя выбранную СУБД. В данном проекте используется PostgreSQL.

Чтобы создать базу данных, необходимо открыть приложение pgAdmin, предварительно скачав PostgreSQL на персональный компьютер. Далее нужно выполнить следующий алгоритм действий:

1. Нажать на кнопку «Servers», которая показана на рисунке 1, и ввести пароль.

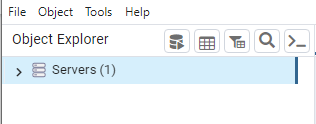


Рисунок 2 — «Servers»

1. Создать базу данных, нажав правой кнопкой мыши по всплывающему окну «Databases». Пример показан на рисунке 2.

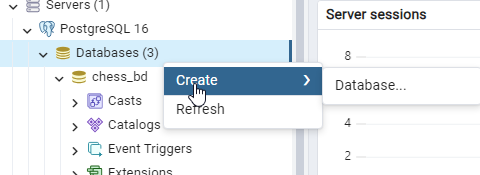


Рисунок 3 — Создание базы данных для проекта

После создания базы данных, необходимо осуществить подключение к проекту. Для этого необходимо открыть интегрированную среду разработки IneliJ IDEA и выполнить следующий алгоритм действий:

1. Открыть проект, к которому необходимо подключить базу данных. Выбрать панель «Database», которая показана на рисунке 3:

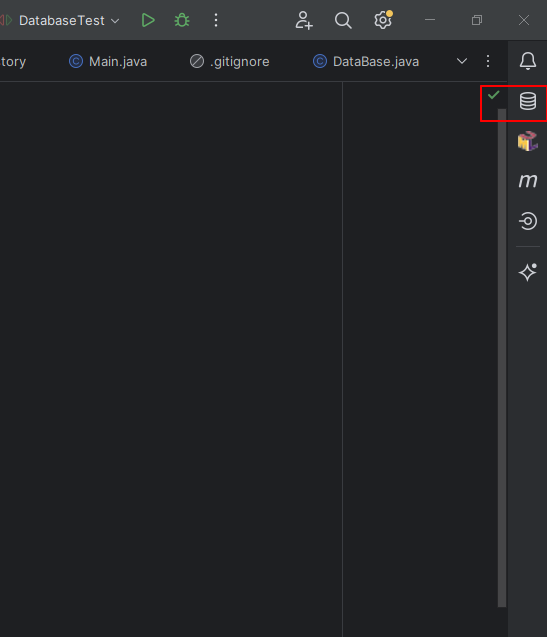


Рисунок 4 — Панель «Database»

1. Нажать «New», затем «Data Source» и выбрать нужную СУБД, как показано на рисунке 4:

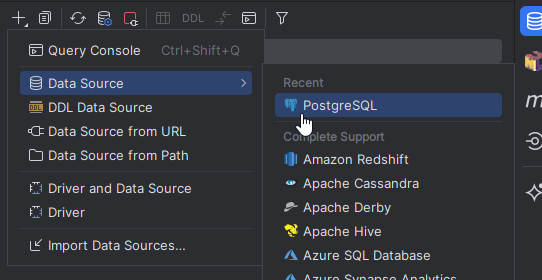


Рисунок 5 — Выбор СУБД для проекта

1. Необходимо ввести имя пользователя, пароль, название базы данных и нажать кнопку «Test connection», должно появиться сообщение, которое показано на рисунке 5:

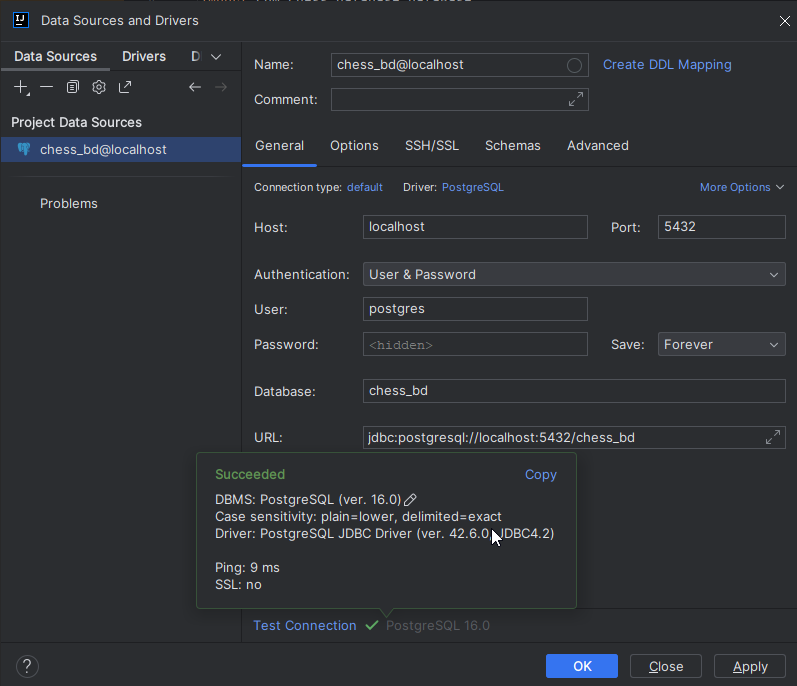


Рисунок 6 — Проверка соединения

После действий, описанных выше, к проекту будет подключена база данных. После этого необходимо настроить подключение с помощью JDBC, чтобы выполнять различные запросы. Для этого необходимо подключить зависимость, в которой есть методы для работы с PostgreSQL.

Для работы с базой данных реализуется класс «Database». В нем описывается методы для работы с подключенной базой данных.

После подключения зависимости и создания класса, необходимо реализовать подключение к базе данных через конструктор класса. Реализация показана на листинге 2.14.

Листинг 2.14 — подключение к базе данных.

try {  
 DatabaseProperties properties = new DatabaseProperties();  
 connection = DriverManager.getConnection(  
 properties.getUrl(),  
 properties.getLogin(),  
 properties.getPassword()  
 );  
} catch (SQLException e) {  
 throw new RuntimeException(e);

Для корректной работы программы необходима возможность сохранения даты начала и конца партии, цвет победителя и последовательность ходов. Для этого нужно создать создать таблицу, которая будет содержать указанные атрибуты. Создание таблицы показано на рисунке 6.

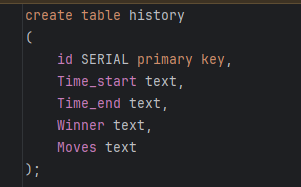


Рисунок 7 — Создание таблицы для проекта

Атрибут «id» отражает уникальный номер партии, поэтому он имеет тип данных SERIAL — каждый следующий добавляемый элемент увеличивается на 1.Оставшиеся 4 атрибута имеют тип text и отражают время начала и окончания, победителя и последовательность ходов партии соответственно.

Для дальнейшего извлечения, обновления и добавления информации необходимо реализовать SQL запросы в среде программирования Java. Для этого используется функционал JDBC.

Список запросов, которые необходимо реализовать:

1. SELECT — общий вид запроса «SELECT column1, column2, ... FROM table\_name WHERE condition», где «column1, column2, ...» — имена столбцов, которые необходимо извлечь; «table\_name» — имя таблицы или представления из которого нужно извлечь данные; «condition» — условия, которым должны удовлетворять строки удовлетворяющие которым нужно извлечь.
2. INSERT — общий вид запроса «INSERT INTO table\_name (column1, column2, ...) VALUES (value1, value2, ...);», где «table\_name» — имя таблицы, в которую нужно вставить новую строку; «(column1, column2, ...)» — список столбцов, в которые будут вставлены значения; «(value1, value2, ...)» — значения, которые будут вставлены в соответствующие столбцы. Значения должны соответствовать типу данных столбцов и быть указаны в правильном порядке.
3. DELETE — общий вид запроса «DELETE FROM table\_name WHERE condition», где «table\_name» — название таблицы, из которой необходимо удалить данные; «condition» — условия, по которым будут удаляться элементы.

Для реализации данных методов понадобятся следующие вспомогательные методы, реализующие следующие запросы:

1. «SELECT id FROM history WHERE id = (SELECT MAX(id) FROM history» — данный запрос возвращает посдений индекс в таблице. Этот метод понадобится для проверок и реализации запросов.
2. «ALTER SEQUENCE history\_id\_seq RESTART with» — данный запрос устанавливает новое значение для счётчика id. Обновлять счеткик нужно после удаления.

Далее реализуются вспомогательные методы. Для этого понадобятся такие особенности Java, как try с ресурсами.[1]

Внутри try необходимо создать объект, реализующий интерфейс Statement и с помощью подключения к базе данных присвоить объекту результат метода createStatement. Затем нужно создать строку и вписать в неё полный SQL запрос, который необходим. Далее нужно создать объект ResulSet и выполнить на нем SQL запрос. Создание методов представлено в листинге 2.15 и 2.16.

Листинг 2.15— метод «updateLastId».

private void updateLastId(int id) {  
 try (Statement statement = connection.createStatement()) {  
 String sql = "ALTER SEQUENCE history\_id\_seq RESTART with " + (id + 1);  
 statement.executeUpdate(sql);  
 } catch (SQLException e) {  
 throw new RuntimeException(e);  
 }  
}

Листинг 2.16 — метод «getLastId».

public int getLastId() {  
 try (Statement statement = connection.createStatement()) {  
 String sql = """  
 SELECT id  
 FROM history  
 WHERE ID = (SELECT MAX(id) FROM history)  
 """;  
 ResultSet resultSet = statement.executeQuery(sql);  
 while (resultSet.next()) {  
 return resultSet.getInt("id");  
 }  
 } catch (SQLException e) {  
 throw new RuntimeException(e);  
 }  
 return 0;  
}

Далее реализуется запрос INSERT, который будет добавлять данные в созданную таблицу. Для этого используется такая же идея, как и в запросах, описанных выше. Отличие будет в SQL запросе, который будет составляться из данных, передаваемых в этот метод. На вход методу поступает 4 значения: ZonedDateTime start — время начала; ZonedDateTime end — время окончания; String winner — победитель в партии; List<Move> moves — последовательность ходов.[3] Из этих данных формируется строка, которая будет частью SQL запроса. Полный код представлен в листинге 2.17.

Листинг 2.17 — INSERT метод.

public void makeInsertSql(ZonedDateTime start, ZonedDateTime end, String winner, List<Move> moves) {  
 try (Statement statement = connection.createStatement()) {  
 String info = "'" + start + "', '" + end + "', '" + winner + "', '" + moves + "'";  
 String sql = "insert into history(Time\_start, Time\_end, Winner, Moves) values (" + info + ")";  
 statement.executeUpdate(sql);  
 } catch (SQLException e) {  
 throw new RuntimeException(e);  
 }  
}

Далее реализуется DELETE метод. Для этого потребуется тот же функционал, какой был в предыдущих методах. Для удаления из таблицы необходимо составить SQL запрос, при этом указать нем id удаляемой строки. Поэтому метод получает на вход индекс, по которому необходимо удалить строку. Также после удаления обновляется id, чтобы элементы шли по порядку. Полная реализация представлена в листинге 2.18.

Листинг 2.18 — DELETE метод.

public void deleteData(int id) {  
 try (Statement statement = connection.createStatement()) {  
 String sql = "DELETE FROM history where id =" + id;  
 statement.executeUpdate(sql);  
  
 this.updateLastId(this.getLastId());  
 } catch (SQLException e) {  
 throw new RuntimeException(e);  
 }  
}

Далее реализуется SELECT метод. Для этого потребуется функционал, который описан выше. Для составления SQL запроса необходим id элемента, который нужно вывести из таблицы. Вся информация, полученная из SELECT запроса, добавляется в List и возвращается из метода. Полная реализация представлена в листинге 2.19.

Листинг 2.19 — SELECT метод.

public List<String> getSelectSql(int id) {  
 try (Statement statement = connection.createStatement()) {  
 String sql = "SELECT \* FROM history where id = " + id;  
 ResultSet resultSet = statement.executeQuery(sql);  
 List<String> data = new ArrayList<>();  
 while (resultSet.next()) {  
 data.add(resultSet.getString("time\_start"));  
 data.add(resultSet.getString("time\_end"));  
 data.add(resultSet.getString("winner"));  
 data.add(resultSet.getString("moves"));  
 }  
 return data;  
 } catch (SQLException e) {  
 throw new RuntimeException(e);  
 }  
}

Такой набор методов позволит корректно работать с базой данных. В проекте имеется возможность сохранять, извлекать и удалять данные из таблицы.

* 1. Дерево функций

Дерево функций используется в данном проекте для декомпозирования задач. Это позволяет эффективно распределить время разработки и выделить основные части, которые необходимо реализовать.

Основные состояния игры: играть и управлять игрой. Каждое из этих состояний декомпозируется на мелкие подзадачи. Функция, от которых не происходит дальнейшая детализация иерархии, являются базовыми.

Дерево функций проекта показано в приложении Е.

* 1. UML диаграмма последовательностей

Полная диаграмма последовательностей показана в приложении Ж.

Диаграмма последовательности отражает жизненный цикл программы. Из диаграммы следует, что жизненный цикл программы продолжается до момента окончания партии. Затем происходит сохранение партии в базу данных и вывод номера пар на экран.

На диаграмме последовательности также можно проследить взаимодействие классов между собой с помощью концептуальных действий.

* 1. Диаграмма классов

Полная диаграмма классов показана в приложении З.

Диаграмма классов отражает иерархию классов между собой. Основные связи описаны в пункте 1.2.2. На основе них можно понять, как классы функционируют относительно друг друга.

В частности, на диаграмме видны иерархии наследования между классами, предназначенными для фигур. Также можно наблюдать классы, которые связаны агрегацией и композицией. Из этого можно сделать выводы, какие классы зависят от других.

1. Тестирование
   1. Описание Junit

Тестирование программы будет осуществляться с помощью Java библиотеки JUnit. Unit — это фреймворк для тестирования и создания модульных тестов в языке программирования Java. Он предоставляет набор аннотаций, классов и методов, которые помогают разработчикам автоматизировать процесс тестирования и обеспечить надежность и качество кода.

JUnit позволяет создавать юнит-тесты, которые проверяют отдельные компоненты или модули программного обеспечения. Юнит-тесты изолируют части кода и проверяют их поведение, убеждаясь, что они работают правильно и не производят нежелательные побочные эффекты.

Преимущества использования JUnit:

1. Простота. JUnit предоставляет простой и понятный способ написания тестовых методов.

2. Автоматизация. JUnit автоматизирует процесс выполнения тестов и предоставляет отчеты о результатах выполнения тестов.

3. Интеграция с средой разработки. JUnit легко интегрируется с популярными средами разработки Java, такими как Eclipse и IntelliJ IDEA.

4. Расширяемость. JUnit предоставляет возможность создания собственных аннотаций, расширений и правил тестирования.

В листингах ниже тестируются методы с помощью Junit. Для этого создается тест класс в папке test.

* 1. Тестовые классы
     1. Класс BoardTest

В тест-классе BoardTest осуществляется проверка трех методов для работы с доской: getPiece, makeMove, isSquareAttackedByColor.

Состояние шахматной доски, которое используется для тестирования метода «getPiece», показано на рисунке 8.

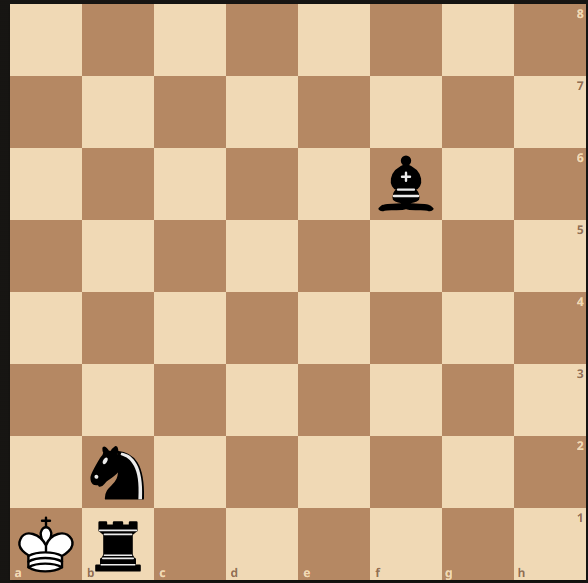


Рисунок 8 — Состояние шахматной доски

Для проверки текущего расположения каждой фигуры используется тест-метод, реализация которого показана в листинге 3.1.

Листинг 3.1 — тестирование метода «getPiece».

@Test  
public void getPieceTest() {  
 board = (new BoardFactory()).fromFEN("8/8/5b2/8/8/8/1n6/Kr6 w - - 0 1");  
 Assert.assertTrue(board.getPiece(new Coordinates(File.A, 1)) instanceof King);  
 Assert.assertTrue(board.getPiece(new Coordinates(File.B, 1)) instanceof Rook);  
 Assert.assertTrue(board.getPiece(new Coordinates(File.B, 2)) instanceof Knight);  
 Assert.assertTrue(board.getPiece(new Coordinates(File.F, 6)) instanceof Bishop);  
}

Для тестирования метода «makeMove» понадобится состояние шахматной доски, показанное на рисунке \_.

Полная реализация тестирования метода показана в листинге 3.2.

Листинг 3.2 — тестирование метода «makeMove».

@Test  
public void makeMoveTest() {  
 board = (new BoardFactory()).fromFEN("8/8/5b2/8/8/8/1n6/Kr6 w - - 0 1");  
 board.makeMove(new Move(new Coordinates(File.A, 1), new Coordinates(File.B, 1)));  
 Assert.assertTrue(board.getPiece(new Coordinates(File.B, 1)) instanceof King);  
}

Для тестирования метода «isSquareAttackedByColor» используется состояние шахматной доски, показанное на рисунке 9.

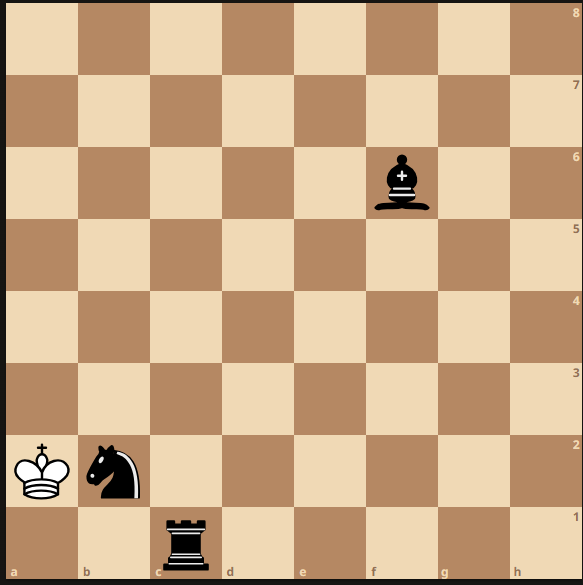


Рисунок 9 — Состояние шахматной доски

Полная реализация тестирования метода показана в листинге 3.3.

Листинг 3.3 — тестирование метода «isSquareAttackedByColor».

@Test  
public void isSquareAttackedByColorTest() {  
 board = (new BoardFactory()).fromFEN("8/8/5b2/8/8/8/Kn6/2r5 w - - 0 1");  
 Assert.assertTrue(board.isSquareAttackedByColor(new Coordinates(File.B, 1), Color.BLACK));  
 Assert.assertTrue(board.isSquareAttackedByColor(new Coordinates(File.A, 1), Color.BLACK));  
 Assert.assertFalse(board.isSquareAttackedByColor(new Coordinates(File.A, 3), Color.BLACK));  
}

* + 1. Класс CheckmateGameCheckerTest

В тест-классе CheckmateGameCheckerTest осуществляется проверка четырех методов: gameStateCheckMateToWhiteKing, gameStateCheckMateToBlackKing, gameStateOngoing, gameStateCheckStalemate. Эти методы необходимы для проверки доски на наличие мата.

Для тестирования метода «gameStateCkeckMateToWhiteKing» потребуется состояние шахматной доски, показанное на рисунке 10.

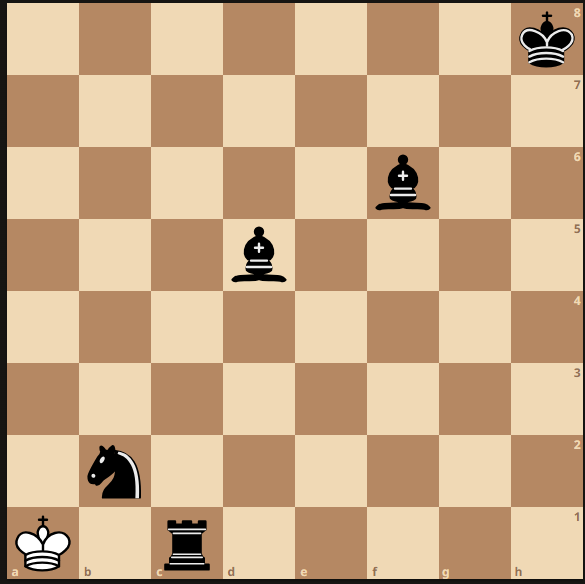


Рисунок 10 — Состояние доски

Полная реализация тест-метода представлена в листинге 3.4.

Листинг 3.4 — тестирование метода «gameStateCkeckMateToWhiteKing».

@Test  
public void gameStateCheckMateToWhiteKing() {  
 board = (new BoardFactory()).fromFEN("7k/8/5b2/3b4/8/8/1n6/K1r5 w - - 0 1");  
 Assert.assertEquals(GameState.CHECKMATE\_TO\_WHITE\_KING, checkerColor.check(board, Color.WHITE));  
}

Для тестирования метода «gameStateOngoing» потребуется состояние шахматной доски, показанное на рисунке 11.

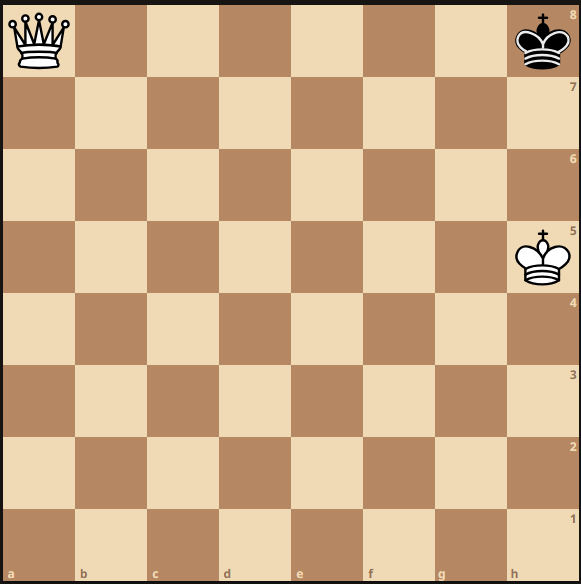


Рисунок 11 — Состояние доски

Полная реализация тест-метода представлена в листинге 3.5.

Листинг 3.5 — тестирование метода «gameStateOngoing».

@Test  
public void gameStateOngoing() {  
 board = (new BoardFactory()).fromFEN("Q6k/8/8/7K/8/8/8/8 w - - 0 1");  
 Assert.assertEquals(GameState.ONGOING, checkerColor.check(board, Color.BLACK));  
}

Для тестирования метода «gameStateCheckStalemate» потребуется состояние шахматной доски, показанное на рисунке 12.

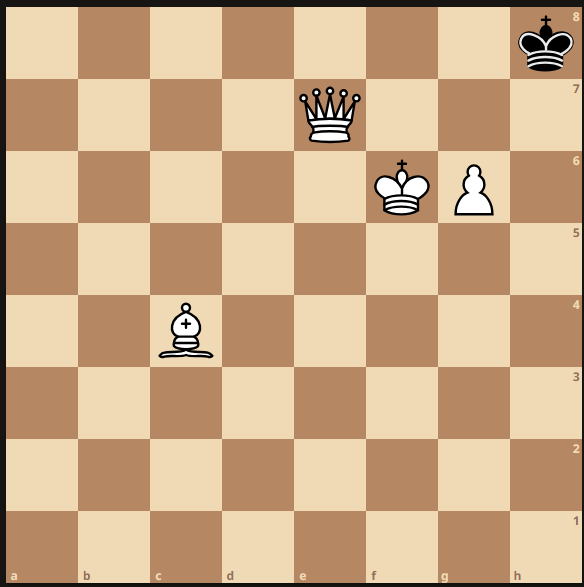


Рисунок 12 — Состояние доски

Полная реализация тест-метода представлена в листинге 3.6.

Листинг 3.6 — тестирование метода «gameStateCheckStalemate».

@Test  
public void gameStateCheckStalemate() {  
 board = (new BoardFactory()).fromFEN("7k/4Q3/5KP1/8/2B5/8/8/8 w - - 0 1");  
  
 Assert.assertEquals(GameState.STALEMATE, checker.check(board, Color.BLACK));  
}

Для тестирования метода «gameStateCkeckMateToBlackKing» потребуется состояние шахматной доски, показанное на рисунке 13.

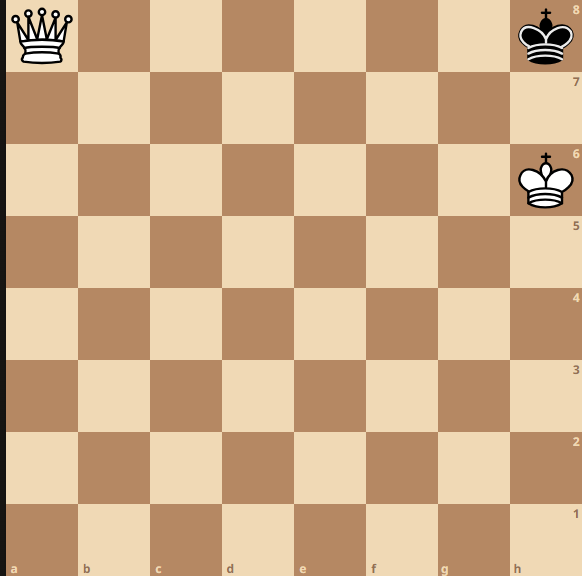


Рисунок 13 — Состояние доски

Полная реализация тест-метода представлена в листинге 3.7.

Листинг 3.7 — тестирование метода «gameStateCkeckMateToBlackKing».

@Test  
public void gameStateCheckMateToBlackKing() {  
 board = (new BoardFactory()).fromFEN("Q6k/8/7K/8/8/8/8/8 w - - 0 1");  
  
 Assert.assertEquals(GameState.CHECKMATE\_TO\_BLACK\_KING, checkerColor.check(board, Color.BLACK));  
}

* + 1. DatabaseTest

Для корректного написания тест-методов понадобится вспомогательный метод, который добавляет элемент в таблицу. Это необходимо для тестирования DELETE и INSERT запроса. Полная реализация метода представлена в листинге 3.8.

Листинг 3.8 — добавление элемента в таблицу.

private void addElementForTest() {  
 List<Move> moves = new ArrayList<>();  
 moves.add(new Move(new Coordinates(File.B, 2), new Coordinates(File.B, 4)));  
 dataBase.makeInsertSql(ZonedDateTime.now(), ZonedDateTime.now(), "Black", moves);  
}

Далее реализуем тест метод для нахождения последнего id в таблице. Полная его реализация представлена в листинге 3.9.

Листинг 3.9 — тест нахождения последнего id.

@Test  
public void lastIdInTableTest() {  
 Assert.assertEquals(8, dataBase.getLastId());  
 Assert.assertNotEquals(7, dataBase.getLastId());  
}

Информация, хранящаяся в таблице, представлена на рисунке 14.

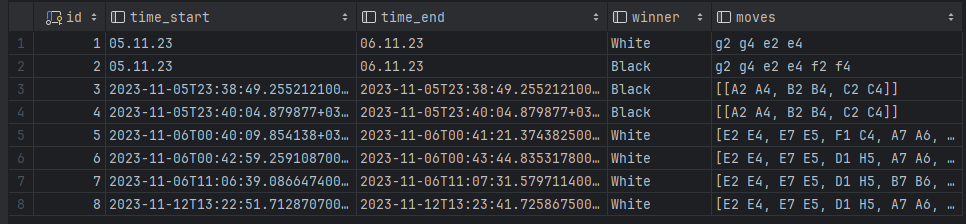


Рисунок 14 — Информация в таблице

Как мы видим, последний id = 8, поэтому в тест методе мы сравниваем последний индекс с 8 и ожидаем верного результата. Результат теста представлен на рисунке 15.

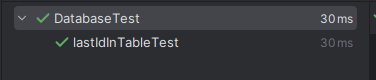


Рисунок 15 — Результат теста

Далее протестируем INSERT метод. Его тестирование будет основываться на методе, который возвращает последний id. Полная реализация тест метода показана в листинге 3.10.

Листинг 3.10 — тест сохранения в таблицу.

@Test  
public void insertSqlTest() {  
 int id = dataBase.getLastId();  
 addElementForTest();  
 Assert.assertEquals(id + 1, dataBase.getLastId());  
 dataBase.deleteData(dataBase.getLastId());  
}

Проверка основывается на сравнении id элементов. В начале вычисляется id последнего элемента, а затем вычисляем id после добавления. Для прохождения теста необходимо, чтобы совпадали id предыдущего с прибавлением единицы и нынешнего. Затем этот элемент удаляется из таблицы.

Далее реализуется тест метод для SELECT запроса. Полная реализация представлена в листинге 3.11.

Листинг 3.11 — тест сохранения в таблицу.

@Test  
public void selectSqlTest() {  
 List<String> testCase = new ArrayList<>();  
 testCase.add("2023-11-06T00:40:09.854138+03:00[Europe/Moscow]");  
 testCase.add("2023-11-06T00:41:21.374382500+03:00[Europe/Moscow]");  
 testCase.add("White");  
 testCase.add("[E2 E4, E7 E5, F1 C4, A7 A6, D1 H5, A6 A5, H5 F7]");  
 List<String> result = dataBase.getSelectSql(5);  
 Assert.assertEquals(testCase, result);  
}

В нем используется List, в который складываются строки таблицы, которые представлены на рисунке 7. В качестве примера используются данные, находящиеся под id = 5. Затем выполняется запрос и сравниваются результаты.

Заключительный тест метод будет проверять DELETE запрос. Полная реализация представлена в листинге 3.12.

Листинг 3.12 — тест сохранения в таблицу.

@Test  
public void deleteDataTest() {  
 addElementForTest();  
 int id = dataBase.getLastId();  
 dataBase.deleteData(id);  
 Assert.assertNotEquals(id, dataBase.getLastId());  
}

В нем добавляется элемент, чтобы затем его удалить, предварительно сохранив id этого элемента. Так как элемент добавляется в конец, а затем удаляется, последний id должен остаться таким же, как и до удаления. Поэтому сравниваются элементы до добавления и после.

* + 1. Класс PiecesAvailableMovesTest

В тест-классе «PiecesAvailableMovesTest» будет проверяться метод «getAvailableMoveSquares», который возвращает возможные ходы фигуры. Для этого будут использоваться отдельные тест-методы для каждой фигуры.

Для тестирования метода у ферзя используется состояние доски, показанное на рисунке 16.

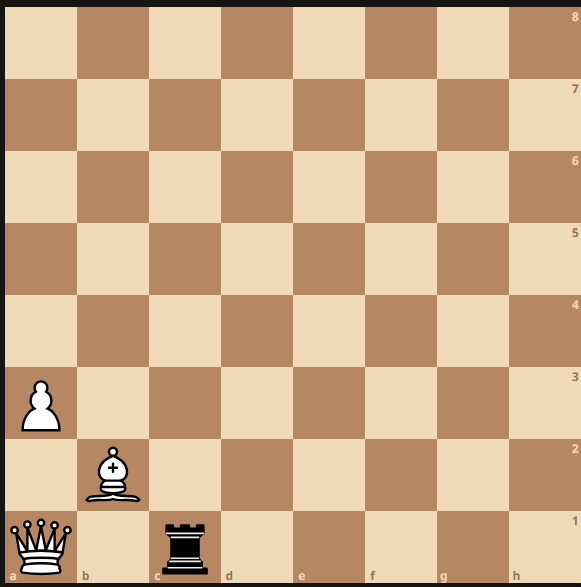


Рисунок 16 — Состояние доски

Полная реализация тест-метода представлена в листинге 3.13.

Листинг 3.13 — тестирование метода у ферзя.

@Test  
public void testQueenMoves() {  
 board = (new BoardFactory()).fromFEN("8/8/8/8/8/P7/1B6/Q1r5 w - - 0 1");  
 // 8/8/8/8/8/P7/1B6/Q1r5 w - - 0 1  
   
 Set<Coordinates> testCaseMoves = new HashSet<>();  
 testCaseMoves.add(new Coordinates(File.A, 2));  
 testCaseMoves.add(new Coordinates(File.B, 1));  
 testCaseMoves.add(new Coordinates(File.C, 1));  
 piece = new Queen(Color.WHITE, new Coordinates(File.A, 1));  
  
 Assert.assertEquals(testCaseMoves, piece.getAvailableMoveSquares(board));  
}

Для тестирования метода у слона используется состояние доски, показанное на рисунке 17.

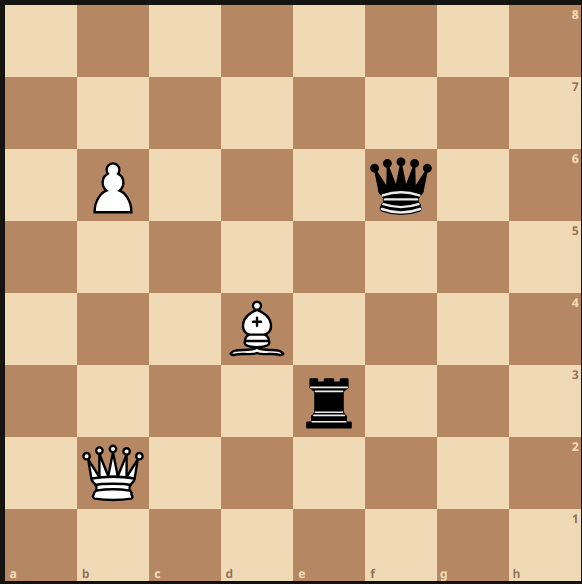


Рисунок 17 — Состояние доски

Полная реализация тест-метода представлена в листинге 3.14.

Листинг 3.14 — тестирование метода у слона.

@Test  
public void testBishopMove() {  
 board = (new BoardFactory()).fromFEN("8/8/1P3q2/8/3B4/4r3/1Q6/8 w - - 0 1");  
   
 Set<Coordinates> testCaseMoves = new HashSet<>();  
 testCaseMoves.add(new Coordinates(File.E, 3));  
 testCaseMoves.add(new Coordinates(File.F, 6));  
 testCaseMoves.add(new Coordinates(File.E, 5));  
 testCaseMoves.add(new Coordinates(File.C, 3));  
 testCaseMoves.add(new Coordinates(File.C, 5));  
 piece = new Bishop(Color.WHITE, new Coordinates(File.D, 4));  
  
 Assert.assertEquals(testCaseMoves, piece.getAvailableMoveSquares(board));  
}

Для тестирования метода у ладьи используется состояние доски, показанное на рисунке 18.

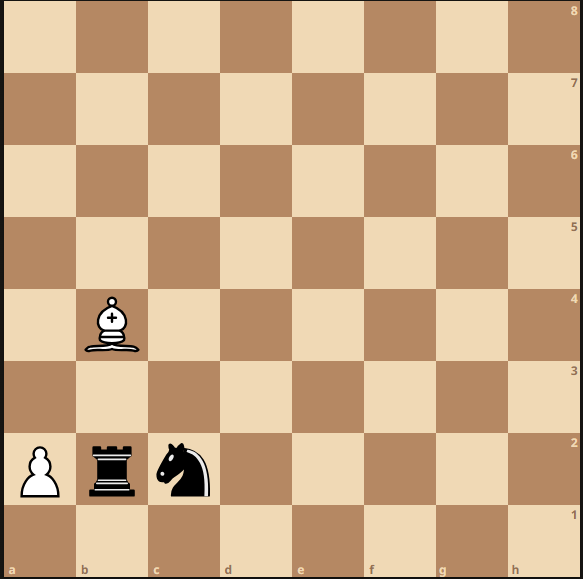


Рисунок 18 — Состояние доски

Полная реализация тест-метода представлена в листинге 3.15.

Листинг 3.15 — тестирование метода у ладьи.

@Test  
public void testRookMoves() {  
 board = (new BoardFactory()).fromFEN("8/8/8/8/1B6/8/Prn5/8 w - - 0 1");   
 Set<Coordinates> testCaseMoves = new HashSet<>();  
 testCaseMoves.add(new Coordinates(File.B, 1));  
 testCaseMoves.add(new Coordinates(File.A, 2));  
 testCaseMoves.add(new Coordinates(File.B, 3));  
 testCaseMoves.add(new Coordinates(File.B, 4));  
 piece = new Rook(Color.BLACK, new Coordinates(File.B, 2));  
 Assert.assertEquals(testCaseMoves, piece.getAvailableMoveSquares(board));  
}

Для тестирования метода у коня используется состояние доски, показанное на рисунке 19.

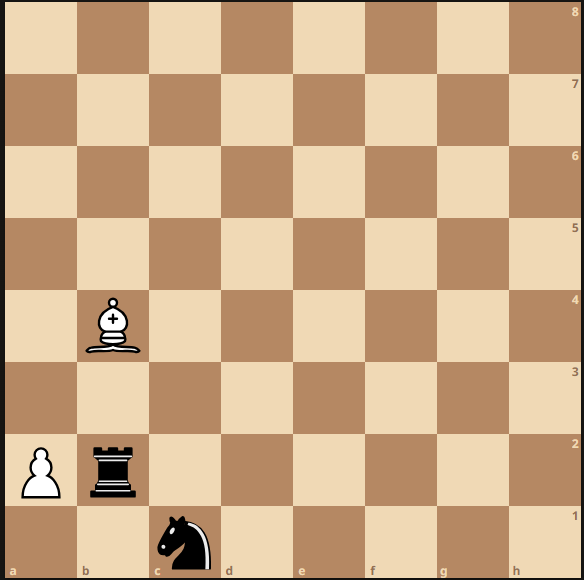


Рисунок 19 — Состояние доски

Полная реализация тест-метода представлена в листинге 3.16.

Листинг 3.16 — тестирование метода у коня.

@Test  
public void testKnightMoves() {  
 board = (new BoardFactory()).fromFEN("8/8/8/8/1B6/8/Pr6/2n5 w - - 0 1");  
   
 Set<Coordinates> testCaseMoves = new HashSet<>();  
 testCaseMoves.add(new Coordinates(File.E, 2));  
 testCaseMoves.add(new Coordinates(File.A, 2));  
 testCaseMoves.add(new Coordinates(File.B, 3));  
 testCaseMoves.add(new Coordinates(File.D, 3));  
  
 piece = new Knight(Color.BLACK, new Coordinates(File.C, 1));  
  
 Assert.assertEquals(testCaseMoves, piece.getAvailableMoveSquares(board));  
}

Для тестирования метода у пешки используется состояние доски, показанное на рисунке 20.

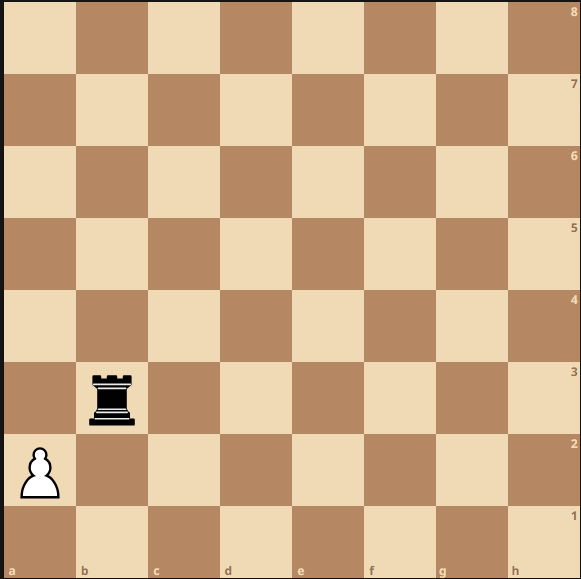


Рисунок 20 — Состояние доски

Полная реализация тест-метода представлена в листинге 3.17.

Листинг 3.17 — тестирование метода у пешки.

@Test  
public void testPawnMoves() {  
 board = (new BoardFactory()).fromFEN("8/8/8/8/8/1r6/P7/8 w - - 0 1");  
   
 Set<Coordinates> testCaseMoves = new HashSet<>();  
 testCaseMoves.add(new Coordinates(File.A, 4));  
 testCaseMoves.add(new Coordinates(File.A, 3));  
 testCaseMoves.add(new Coordinates(File.B, 3));  
  
 piece = new Pawn(Color.WHITE, new Coordinates(File.A, 2));  
  
 Assert.assertEquals(testCaseMoves, piece.getAvailableMoveSquares(board));  
}

Для тестирования метода у короля используется состояние доски, показанное на рисунке 21.

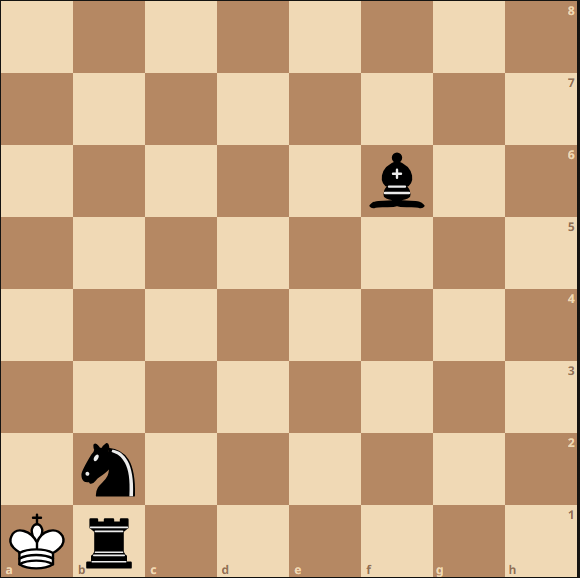


Рисунок 21 — Состояние доски

Полная реализация тест-метода представлена в листинге 3.18.

Листинг 3.18 — тестирование метода у короля.

@Test  
public void testKingMoves() {  
 board = (new BoardFactory()).fromFEN("8/8/5b2/8/8/8/1n6/Kr6 w - - 0 1");  
   
 Set<Coordinates> testCaseMoves = new HashSet<>();  
 testCaseMoves.add(new Coordinates(File.A, 2));  
 testCaseMoves.add(new Coordinates(File.B, 1));  
  
 piece = new King(Color.WHITE, new Coordinates(File.A, 1));  
  
 Assert.assertEquals(testCaseMoves, piece.getAvailableMoveSquares(board));  
}

* 1. Результат тестирования

Результат запуска тестовых классов с помощью интегрированной среды разработки InteliJ IDEA показан на рисунке 22.

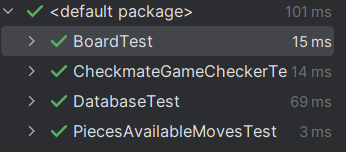


Рисунок 22 — Результат тестирования

Заключение

Объектно-ориентированное программирование (ООП) представляет собой эффективную методологию разработки программных систем, основанную на концепции объектов и классов. Введение ООП в программирование привнесло новые возможности для создания модульных, гибких и масштабируемых приложений.

Язык программирования Java является примером объектно-ориентированного языка, который поддерживает основные принципы ООП, такие как наследование, инкапсуляция и полиморфизм.

Базы данных обеспечивают централизованное хранилище для хранения информации и предлагают гибкость и масштабируемость при обработке данных.

Распределенная система управления версиями Git осуществляет контроль кода и позволяет эффективно вносить обновления и изменения.

Использование UML диаграмм позволяет графически отразить архитектуру проекта и его работу.

В совокупности выше описанные инструменты являются эффективным средством для решения практических задач.

Список литературы

1. Шилдт, Г. Java. Полное руководство, 10-е изд.: Пер. с англ. – СПб. ООО «Альфакнига», 2018. – 1488 с. ISBN 978-5-6040043-6-4.

2. Мейер, Б. Объектно-ориентированное конструирование программных систем: Русская Редакция; 2015. – 768 с. ISBN 5-7502-0255-0.

3. Лафоре, Р. Структуры данных и алгоритмы в Java. / Классика Computers Science. 2-е изд. – СПб.:Питер, 2013. – 704 с. ISBN 978-5-496-007405.

4. Буч, Г., Рамбо, Д., Якобсон, И. Язык UML. / Руководство пользователя, М.:ДМК Пресс, 2006. – 496с. ISBN 5-94074-334-X.

Приложение а

Шахматы – это игра для двух игроков, которые играют на доске, состоящей из 64 квадратов, расположенных в восьми рядах по восемь квадратов в каждом ряду, цвета квадратов чередуются. Существуют вертикали, которые нумеруются буквами (A-H). Существуют горизонтали, которые нумеруются числами (1-8).

Каждый игрок начинает игру с 16 фигур: один король, одна королева, две ладьи, два коня, два слона и восемь пешек. Фигуры на доске расставляются зеркально. Очерёдность выставления фигур следующая:

Для белых фигуры выставляются на первой горизонтали, начиная с вертикали A, в следующей последовательности: ладья, конь, слон, ферзь, король, слон, конь, ладья. Вторая горизонталь занимается восьмью пешками полностью.

Для чёрных фигуры выставляются зеркально белым.

Цель игры – поставить короля противника под шах (угрозу захвата) таким образом, чтобы он не мог уйти от угрозы или защититься от неё. Если король находится под шахом и не может быть защищён или уйти от угрозы, это называется матом, и игра заканчивается.

Правила хода:

1. Каждый игрок делает ход по очереди. Белые ходят первыми.

2. Каждая фигура имеет свои правила хода.

3. Пешка может двигаться только вперёд на одну или две клетки в своём первом ходе, а затем на одну клетку вперёд.

4. Ладья может двигаться только по горизонтали или вертикали.

5. Слон может двигаться по диагонали своего цвета.

6. Конь может двигаться Г-образно (две клетки по горизонтали или вертикали, а затем на одну клетку по вертикали или горизонтали).

7. Король может двигаться на одну клетку в любом направлении.

8. Ферзь сочетает в себе возможность ходов двух фигур: слона и ладьи.

9. Фигуры могут захватывать фигуры противника, заменяя их на клетке, на которую они заходят.

10. Когда король находится под шахом, игрок должен защитить его или уйти от угрозы.

11. Если пешка достигает последней горизонтали доски, она может быть заменена на любую другую фигуру, кроме короля.

12. Если король не может двигаться, но он не находится под шахом, а также не может сходить никакая другая фигура, это называется патом, и игра заканчивается ничьей.

Приложение б

public void playOrCheckHistory() {  
 System.out.println("Write: PLAY or HISTORY");  
 String check = scanner.nextLine();  
  
 check = check.toUpperCase();  
 while (true) {  
 switch (check) {  
 case "PLAY": {  
 gameLoop();  
 return;  
 }  
 case "HISTORY": {  
 checkHistory();  
 return;  
 }  
  
 default: {  
 System.out.println("Wrong data, please retype PLAY or HISTORY");  
 check = scanner.nextLine().toUpperCase();  
 }  
 }  
 }  
}

Приложение в

public void checkHistory() {  
 System.out.println("Write the ID of the party");  
 int id = scanner.nextInt();  
  
 while (true) {  
 if (id > dataBase.getLastId() || id <= 0) {  
 System.out.println("Wrong data : there is no such id");  
 System.out.println("Retype id");  
 id = scanner.nextInt();  
 continue;  
 }  
  
 break;  
 }  
  
 List<Move> moves = dataBase.getMoves(id);  
 System.out.println("Moves in party : " + moves);  
 System.out.println("Do you want to reproduce the party? (yes, no)");  
 Scanner scanner1 = new Scanner(System.in);  
 String check = scanner1.nextLine();  
  
 check = check.toLowerCase();  
  
 switch (check) {  
 case "yes": {  
 render.render(board);  
  
 try {  
 Thread.sleep(1500);  
 } catch (InterruptedException e) {  
 throw new RuntimeException(e);  
 }  
  
 for (Move move : moves) {  
 board.makeMove(move);  
 render.render(board);  
  
 try {  
 Thread.sleep(2000);  
 } catch (InterruptedException e) {  
 throw new RuntimeException(e);  
 }  
 }  
 }  
 case "no": {  
 System.out.println("Program end");  
 }  
 }  
}

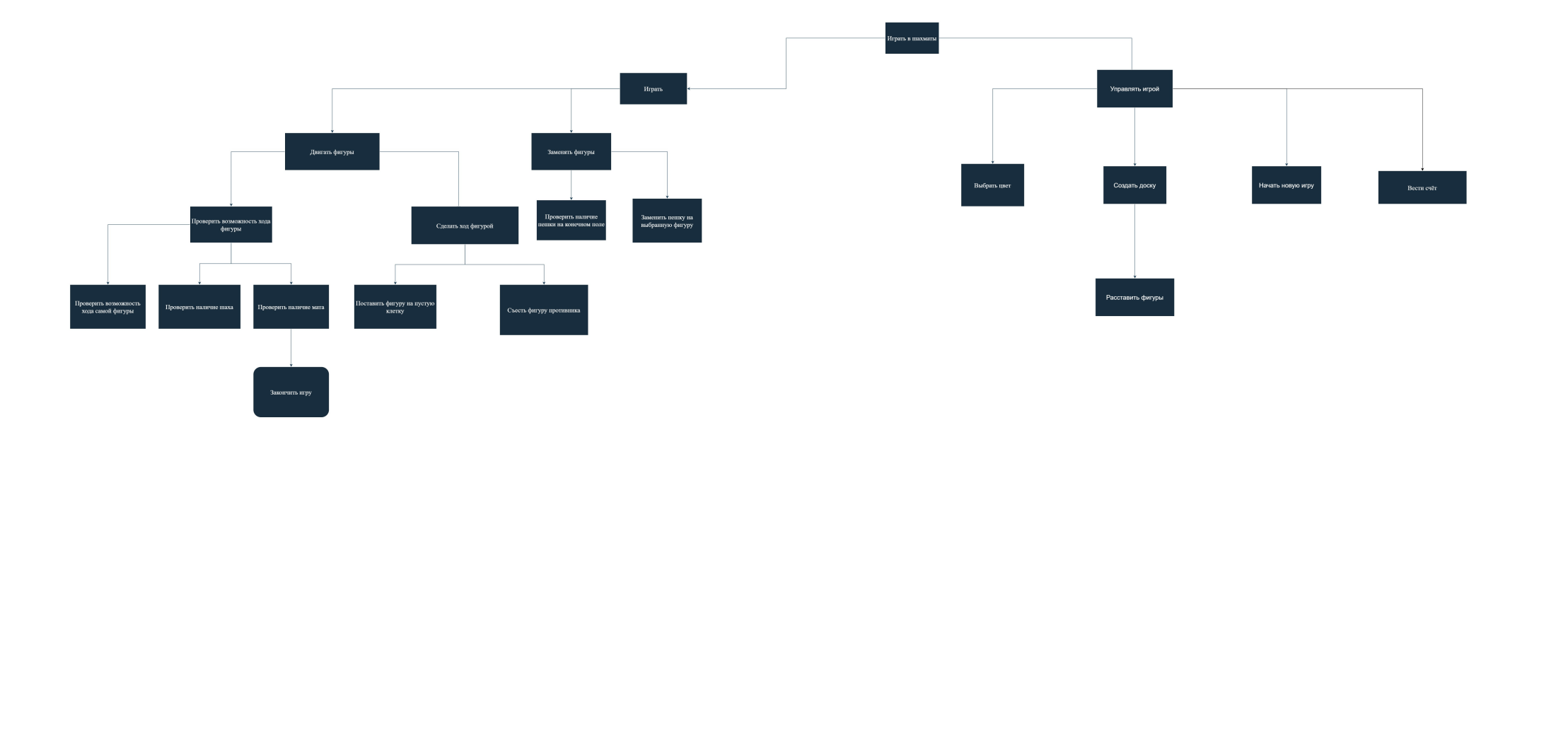
Приложение г

public void gameLoop() {  
 Color colorToMove = Color.WHITE;  
  
 GameState state = determineGameState(board, colorToMove);  
  
 ZonedDateTime start = ZonedDateTime.now();  
  
 while (state == GameState.ONGOING) {  
 render.render(board);  
  
 if (colorToMove == Color.WHITE) {  
 System.out.println("White to move");  
 } else {  
 System.out.println("Black to move");  
 }  
  
 Move move = InputCoordinates.inputMove(board, colorToMove, render);  
  
  
 // make move  
 board.makeMove(move);  
 board.checkPawnPosition(move);  
  
 // pass move  
 colorToMove = colorToMove.opposite();  
  
 state = determineGameState(board, colorToMove);  
 }  
  
 render.render(board);  
 System.out.println("Game ending with state = " + state);  
  
 ZonedDateTime end = ZonedDateTime.now();  
 String winner = state.getWinner(state);  
  
 dataBase.makeInsertSql(start, end, winner, board.moves);  
 System.out.println("Game id : " + dataBase.getLastId());  
}

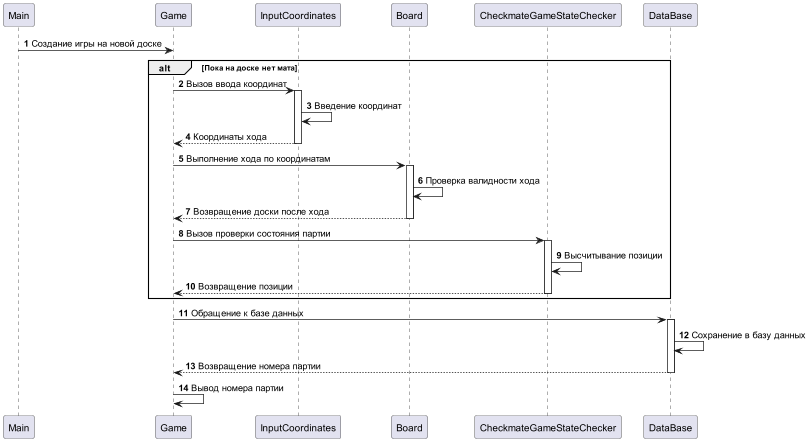
Приложение д

@Override  
public GameState check(Board board, Color color) {   
 Piece king = board.getPiecesByColor(color).stream().filter(piece -> piece instanceof King).findFirst().get();  
   
 if (!board.isSquareAttackedByColor(king.coordinates, color.opposite())) {  
 return GameState.ONGOING;  
 }  
  
 List<Piece> pieces = board.getPiecesByColor(color);  
  
 for (Piece piece : pieces) {  
 Set<Coordinates> availableMoveSquares = piece.getAvailableMoveSquares(board);  
  
 for (Coordinates coordinates : availableMoveSquares) {  
 Board clone = new BoardFactory().copy(board);  
 clone.makeMove(new Move(piece.coordinates, coordinates));  
  
 Piece cloneKing = clone.getPiecesByColor(color).stream().filter(p -> p instanceof King).findFirst().get();  
  
  
 if (!clone.isSquareAttackedByColor(cloneKing.coordinates, color.opposite())) {  
 return GameState.ONGOING;  
 }  
 }  
 }  
  
 if (color == Color.WHITE) {  
 return GameState.CHECKMATE\_TO\_WHITE\_KING;  
 } else {  
 return GameState.CHECKMATE\_TO\_BLACK\_KING;  
 }  
}

Приложение е



Приложение ж



Приложение з

