МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«РОССИЙСКИЙ ГОСУДАРСТВЕННЫЙ ПЕДАГОГИЧЕСКИЙ
УНИВЕРСИТЕТ им. А. И. ГЕРЦЕНА»

Основная профессиональная образовательная программа Направление подготовки 09.03.01 Информатика и вычислительная техника
Направленность (профиль) «Технологии разработки программного обеспечения» форма обучения – очная

Выпускная квалификационная работа

Разработка React-приложения "Классические шахматы"

Научный руководитель:

Кандидат физ. - мат. наук, ассистент, доцент

_____Н. Н. Жуков

«__» _____ 2023 г.

Автор работы:

Студент группы 4об-ИВТ-2/19

_____М. Д. Глебов

«__» _____ 2023 г.

Санкт-Петербург
2023

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

На сегодняшний день направление frontend-разработки является одним из самых быстроразвивающихся в IT отрасли. Транснациональные корпорации вкладывают большое количество средств в развитие индустрии, продукты с открытым исходным кодом, новые языки и библиотеки. Данная работа посвящена созданию современного веб-приложения для игры в шахматы, используя современные подходы к разработке и такие технологии, как typescript и React.

В 2012 году, компания Microsoft представила язык программирования Typescript, статически типизированный язык программирования, который расширяет возможности JavaScript. Он позволяет выявлять ошибки на этапе разработки, упрощает поддержку кода и улучшает его читаемость.

React — это одна из самых популярных библиотек для разработки веб-приложений. Её главным преимуществом является использование виртуального DOM. Виртуальный DOM — это легковесная копия реального DOM, которая хранится в памяти браузера. Когда в React происходят изменения в интерфейсе, React сравнивает виртуальный DOM с реальным DOM и обновляет только те элементы, которые были изменены. Это позволяет существенно ускорить обновление интерфейса и улучшить производительность приложения.

Ещё одно преимущество React — это возможность создавать компоненты. Компоненты — это независимые блоки кода, которые могут быть многократно использованы в приложении. Они позволяют разбить

приложение на более мелкие и управляемые части, что упрощает поддержку и разработку.

Разработчики React также предоставляют Create React App (далее CRA), который представляет собой готовый пакет инструментов для комфортной разработки. CRA не требует сложной настройки и идеально подходит для ознакомления с библиотекой и создания приложения в кратчайшие сроки на основе готовых решений.

В качестве сборщика модулей в CRA используется webpack. Это мощный инструмент с возможностью подключения собственных плагинов для обработки различных типов файлов и анализа вашего приложения. Важным для разработчика инструментом, предоставляемым webpack, является DevServer. Он предоставляет возможность отслеживать внесенные изменения в режиме реального времени. В совокупности с плагином Hot Module Replacement, процесс пересборки приложения занимает кратчайшие сроки. Webpack имеет два варианта для сборки приложения: production и development. Production сборка подразумевает минификацию, то есть удаление из конечной сборки отступов и всего неиспользуемого кода в результате tree shaking, в результате получается JavaScript файл с минимальным весом для более быстрой скорости передачи клиенту.

Шахматы являются одной из наиболее популярных и уважаемых игр в мире. Они имеют давнюю историю, уходящую своими корнями в древнейшие цивилизации, и представляют собой не только развлечение, но и объект изучения искусства стратегического мышления. Неудивительно, что шахматы стали одним из первых объектов, над которыми ученые начали изучать человеческий интеллект, и до сих пор они остаются популярным объектом исследования в области искусственного интеллекта и машинного обучения.

Предполагается реализация следующих задач:

- Создание и настройка среды для написания кода приложения

- Реализация методологии DevOps для непрерывной интеграции и деплоя нового функционала
- Разработка поля для игры, которое в дальнейшем можно будет использовать и для других настольных игр, вроде шашек.
- Разработать универсальный класс, на котором будут основываться все фигуры и который содержит все основные возможности для перемещения по игровой доске
- Разработать шахматные фигуры и реализовать игровые механики, такие как рокировка и превращение пешек в ферзя на последней горизонтали.
- Добавить таймер для контроля времени и фиксирование потерянных фигур

В настоящее время шахматы продолжают оставаться актуальными, привлекая множество людей во всем мире, в том числе и тех, кто хочет улучшить свои навыки мышления и развить свою креативность. Они также остаются популярным объектом исследования в области компьютерных наук, и создание программных решений, способных играть в шахматы на профессиональном уровне, остается одним из вызовов для современной информатики. В связи с этим, разработка программного решения, позволяющего играть в классические шахматы с использованием языка программирования TypeScript и библиотеки React, является актуальной и интересной задачей, позволяющей сочетать в себе традиции и современные технологии.

ГЛАВА 1. ТЕОРИТИЧЕСКАЯ ЧАСТЬ

1.1 TYPESCRIPT

TypeScript — это язык программирования, который является надмножеством JavaScript. Он добавляет статическую типизацию, что позволяет разработчикам легче отслеживать ошибки в коде на ранних этапах разработки и создавать более надежные приложения.

1.1.1 ПРЕИМУЩЕСТВА TYPESCRIPT

Основные преимущества TypeScript:

- Статическая типизация: TypeScript позволяет определять типы данных, которые будут использоваться в приложении, что помогает предотвратить ошибки во время выполнения программы.
- Улучшенная поддержка IDE: TypeScript позволяет использовать мощные функции IDE, такие как автодополнение кода, рефакторинг, навигация по коду и т.д.
- Обратная совместимость с JavaScript: TypeScript является надмножеством JavaScript, поэтому любой код на JavaScript может быть легко перенесен в TypeScript, а код на TypeScript может быть скомпилирован в код на JavaScript.

- Легко поддерживать большие проекты: TypeScript упрощает поддержку больших проектов, поскольку его статическая типизация позволяет быстро находить и исправлять ошибки, а также позволяет разрабатывать сложные приложения с лучшей структурой и организацией кода.
- Большое сообщество разработчиков: TypeScript имеет широкое сообщество разработчиков, что обеспечивает хорошую поддержку и обмен знаниями между разработчиками.

1.1.2 СТРОГАЯ ТИПИЗАЦИЯ

В JavaScript используется динамическая типизация. Из этого следует два заключения:

1. При операции с переменными разных типов они будут автоматически приведены к одному типу.
2. Любая переменная может произвольно менять свой тип во время выполнения программы.

Решение этой проблемы в виде TypeScript добавляет строгую статическую типизацию. Строгость заключается в запрете на автоматическое приведение типов. Например:

```
const age = 43
const name = 'Mary'

// Выдаст ошибку, складывать числа и строки нельзя
const result = age + name
// В JavaScript присвоилось бы значение "43Mary"
```

Статическая типизация значит, что проверка типов происходит на этапе сборки приложения или написания кода. Ошибки несовпадения типов будут заметны на стадии написания кода. То есть не нужно запускать программу, чтобы узнать об ошибках в типизации. Главное отличие TypeScript от JavaScript — возможность добавлять аннотации типов к переменным, аргументам функций и их возвращаемым значениям. Например, типизация переменной:

```
// Теперь переменной age можно присвоить только число
let age: number

// Будет работать
age = 43

// Выдаст ошибку
age = '34'
```

Типизация параметров функции:

```
function sayMyName(name: string) {
  console.log(`Привет, ${name}`)
}

// Будет работать
sayMyName('Игорь')
// Привет, Игорь

// Выдаст ошибку
sayMyName(42)
```

Типизация возвращаемого значения:

```
 function getCurrentDate(): Date {
  // Будет работать
  return new Date()
}

function getCurrentDate(): Date {
  // Выдаст ошибку
  return 'now'
}
```

В TypeScript можно создавать сложные типы путем объединения простых. Есть два популярных способа сделать это: с помощью объединений(union-типов) и с помощью дженериков(generics).

С помощью объединения вы можете объявить, что тип может быть одним из многих типов. Например, вы можете описать тип **boolean** как **true** или **false**:

```
type MyBool = true | false;
```

Популярным случаем использования типов объединения является описание набора строковых или числовых литералов, которыми может быть значение:

```
type WindowStates = "open" | "closed" | "minimized";
type LockStates = "locked" | "unlocked";
type PositiveOddNumbersUnderTen = 1 | 3 | 5 | 7 | 9;
```

Объединения также позволяют работать с различными типами. Например, у вас может быть функция, которая принимает массив или строку:

```
function getLength(obj: string | string[]) {
  return obj.length;
}
```

Дженерики предоставляют типам переменные. Обычный пример - массив. Массив без дженериков может содержать что угодно. Массив с дженериками может описывать значения, которые он содержит.

```
type StringArray = Array<string>;
type NumberArray = Array<number>;
type ObjectWithNameArray = Array<{ name: string }>;
```

Еще один удобным инструментом для типизации являются интерфейсы. Интерфейсы применимы к объектам и описывают их поля. Рассмотрим некоторые возможности интерфейсов:

- **readonly**: Хотя это не изменит никакого поведения во время выполнения, свойство, помеченное как **readonly**, не может быть записано во время проверки типов. Использование модификатора **readonly** не обязательно означает, что значение полностью неизменяемо - или, другими словами, что его внутреннее содержимое

не может быть изменено. Это просто означает, что само свойство не может быть перезаписано.

- Опциональные операторы: Чаще всего мы имеем дело с объектами, у которых могут быть установлены опциональные свойства. В таких случаях мы можем пометить эти свойства как необязательные, добавив знак вопроса (**?**) в конце их имен.
- Наследование: Ключевое слово **extends** в интерфейсе позволяет нам эффективно копировать члены из других именованных типов и добавлять любые новые члены. Это может быть полезно для сокращения объема объявлений типов, которые нам приходится писать, и для сигнализации о том, что несколько различных объявлений одного и того же свойства могут быть связаны.

```
interface ComponentProps {
  readonly id: number;  // Поле только для считывания.
  name: string; // Определяет поле как строковый тип.
  email?: string; // Определяет поле как опциональный строковый тип.
  foo: () => void; // Определяет поле как функцию
}
```

В TypeScript существует «строгий режим» — он вынуждает указывать типы в тех случаях, когда язык не может определить их сам.

Если проект только стартует, и TypeScript был изначально выбран как основной язык, лучше включить строгий режим сразу. Если проект был написан на JavaScript, и TypeScript внедряется туда постепенно, строгий режим может сильно замедлить этот процесс.

1.1.3 ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

ООП (объектно-ориентированное программирование) является одним из основных подходов к программированию, который используется для создания сложных и масштабируемых программных систем. TypeScript предоставляет возможность реализации ООП в проектах благодаря ряду инструментов и функций, которые делают написание объектно-ориентированного кода проще и более понятным.

В TypeScript классы играют центральную роль в реализации ООП. Они позволяют создавать объекты, которые могут содержать свойства и методы, а также наследовать их от других классов. Кроме того, TypeScript также поддерживает интерфейсы и абстрактные классы, которые позволяют описывать контракты и реализовывать сложные иерархии типов.

TypeScript является объектно-ориентированным языком программирования, который поддерживает все основные принципы ООП. Принципы ООП включают в себя инкапсуляцию, наследование и полиморфизм.

Инкапсуляция — это принцип ООП, который заключается в том, чтобы скрыть реализацию объекта от других частей программы и обеспечить контролируемый доступ к его методам и свойствам.

В TypeScript инкапсуляция может быть реализована с помощью модификаторов доступа, таких как **public**, **private** и **protected**. Эти модификаторы определяют уровень доступности свойств и методов внутри класса и его подклассов.

Модификатор **public** означает, что свойство или метод доступно из любого места в программе. Это является значением по умолчанию и может быть опущено при объявлении.

Модификатор **private** означает, что свойство или метод доступен только внутри того класса, в котором он был объявлен. Другие части программы не могут получить доступ к этим свойствам и методам напрямую.

Модификатор **protected** означает, что свойство или метод доступен только внутри того класса, в котором он был объявлен, а также внутри его дочерних классов.

С помощью модификаторов доступа разработчики могут контролировать доступ к своим свойствам и методам и скрыть реализацию объекта от других частей программы. Это позволяет создавать более безопасный и контролируемый код, который легче поддерживать и расширять в будущем.

Пример использования модификаторов доступа в TypeScript:

```
class Animal {
  private name: string;

  constructor(name: string) {
    this.name = name;
  }

  public getName() {
    return this.name;
  }

  protected makeSound() {
    console.log("The animal makes a sound");
  }
}

class Dog extends Animal {
  public bark() {
    this.makeSound(); // доступ к защищенному методу родительского класса
    console.log("The dog barks");
  }
}

const dog = new Dog("Rex");
```

```
console.log(dog.getName()); // доступ к публичному методу родительского
класса
dog.bark(); // доступ к публичному методу дочернего класса
```

В данном примере класс **Animal** имеет приватное свойство **name**, которое может быть получено только через публичный метод **getName()**. Также он имеет защищенный метод **makeSound()**, который доступен только внутри класса и его дочерних классов. Класс **Dog** является дочерним классом **Animal** и имеет свой публичный метод **bark()**, который вызывает защищенный метод **makeSound()** родительского класса.

Наследование — это один из основных принципов ООП, который позволяет создавать новые классы на основе уже существующих. В TypeScript, наследование реализуется с помощью ключевого слова **extends**.

При создании нового класса на основе уже существующего, новый класс получает все свойства и методы родительского класса, которые можно переопределить или дополнить в дочернем классе.

Например, если у нас есть класс **Animal**, который имеет свойства и методы для работы с животными, мы можем создать новый класс **Dog**, который наследует все свойства и методы класса **Animal**, а также может иметь дополнительные свойства и методы, специфичные для собак.

Для этого достаточно объявить класс **Dog** следующим образом:

```
class Animal {
  name: string;
  age: number;

  constructor(name: string, age: number) {
    this.name = name;
    this.age = age;
  }

  eat(food: string) {
    console.log(`${this.name} is eating ${food}`);
  }
}

class Dog extends Animal {
  breed: string;
```

```
  constructor(name: string, age: number, breed: string) {
    super(name, age);
    this.breed = breed;
  }

  bark() {
    console.log(`Woof! My name is ${this.name}, I'm ${this.age} years old,
and I'm a ${this.breed}.`);
  }
}
```

В этом примере класс **Dog** наследует все свойства и методы класса **Animal**, а также имеет дополнительное свойство **breed** и метод **bark()**, который выводит информацию о собаке.

Полиморфизм — это принцип ООП, который позволяет объектам одного класса иметь различное поведение в зависимости от контекста использования.

В TypeScript полиморфизм достигается с помощью механизма переопределения методов и свойств в классах-наследниках. Это позволяет использовать методы и свойства базового класса в классах-наследниках, а также переопределять их при необходимости.

Например, если у нас есть класс **Animal** с методом **makeSound()**, который выводит звук животного, то мы можем создать новый класс **Dog**, который переопределяет этот метод и выводит звук лая собаки:

```
class Animal {
  makeSound() {
    console.log('The animal makes a sound');
  }
}

class Dog extends Animal {
  makeSound() {
    console.log('The dog barks');
  }
}
```

Также полиморфизм может проявляться в использовании классов-наследников вместо базового класса в контексте, где ожидается объект базового класса. Например, если у нас есть функция, которая принимает

объект класса **Animal** и вызывает метод **makeSound()**, то мы можем передать в эту функцию объект класса **Dog**, и метод **makeSound()** будет вызван из класса **Dog**:

```
function makeAnimalSound(animal: Animal) {
  animal.makeSound();
}

const animal = new Animal();
const dog = new Dog();

makeAnimalSound(animal); // выводит "The animal makes a sound"
makeAnimalSound(dog); // выводит "The dog barks"
```

Таким образом, TypeScript предоставляет разработчикам все необходимые средства для реализации ООП-принципов в своих проектах. Это позволяет создавать более эффективный, модульный и легко поддерживаемый код, который может быть легко масштабирован в будущем.

1.2 REACT

React — это одна из самых популярных библиотек для разработки пользовательских интерфейсов веб-приложений. Она была создана Facebook и используется многими компаниями и разработчиками во всем мире для создания современных и масштабируемых приложений.

React базируется на концепции компонентного подхода, который позволяет создавать независимые компоненты интерфейса и объединять их в более крупные структуры, такие как страницы и приложения. Это делает код более понятным, модульным и легко поддерживаемым.

1.2.1 JSX И TSX РАСШИРЕНИЯ

JSX — это расширение языка JavaScript, которое позволяет создавать структуру пользовательского интерфейса в React в виде XML-подобного кода. Он делает код более читабельным и легко понятным, а также позволяет использовать JavaScript-выражения внутри HTML-подобных тегов.

TSX — это расширение JSX, которое добавляет поддержку типизации в React-приложениях с использованием TypeScript. Таким образом, TSX позволяет создавать компоненты с типизированными свойствами и состояниями, что делает код более надежным и устойчивым к ошибкам.

Вместо того, чтобы писать привычный HTML-код, в React мы используем JSX (или TSX), где мы описываем структуру интерфейса с помощью JSX или TSX элементов. Эти элементы являются простыми JavaScript-объектами, которые описывают, как выглядит наш интерфейс.

Например, мы можем создать простой элемент кнопки в JSX следующим образом:

```
const button = <button onClick={() => console.log('Button clicked!')}>Click
me!</button>;
```

В данном примере мы создаем кнопку, которая будет выводить сообщение в консоль, когда на нее нажмут. С помощью JSX мы можем легко описать не только внешний вид элементов, но и их поведение.

TSX позволяет нам добавлять типы к JSX-элементам и свойствам, что позволяет нам писать более безопасный и читабельный код. Например, мы можем добавить тип для свойства onClick у нашей кнопки, чтобы TypeScript мог проверить, что мы передаем функцию, а не строку или число:

```
type ButtonProps = {
  onClick: () => void;
}

const button = <button onClick={() => console.log('Button clicked!')}>Click
me!</button>;
```

Таким образом, JSX и TSX являются мощными инструментами для создания интерфейсов в React, которые делают код более понятным и читабельным.

1.2.2 КОМПОНЕНТЫ И УПРАВЛЕНИЕ СОСТОЯНИЕМ

Компоненты — это основные строительные блоки в React, которые позволяют создавать многоразовые и масштабируемые пользовательские интерфейсы. Они представляют из себя независимые блоки кода, которые могут быть использованы повторно в разных частях приложения.

Компонентный подход в React основывается на том, что пользовательский интерфейс разбивается на множество маленьких, независимых компонентов. Каждый компонент имеет свой собственный набор свойств (props) и состояний (state), которые могут изменяться с течением времени. Компоненты могут быть классовыми или функциональными.

Функциональные компоненты в React до версии 16.8 были ограничены в своих возможностях и не имели состояний и методов жизненного цикла. Однако, начиная с версии 16.8, был добавлен хук - **useState**, который позволяет функциональным компонентам иметь состояния.

Также, начиная с версии 16.8, в React появились хуки (hooks), которые позволяют функциональным компонентам иметь методы жизненного цикла и другие возможности классовых компонентов.

Таким образом, функциональные компоненты в React уже не являются полностью лишенными состояний и методов жизненного цикла, но все еще могут использоваться без них в более простых случаях, что делает их более легковесными и производительными по сравнению с классовыми компонентами.

Одним из ключевых отличий между классовым и функциональным компонентом в React является синтаксис. Классовые компоненты создаются с помощью ключевого слова **class** от встроенного класса **React.Component**, а функциональные - с помощью ключевого слова **function** или стрелочной функцией.

Классовые компоненты имеют доступ к методам жизненного цикла, таким как **componentDidMount**, **componentDidUpdate** и **componentWillUnmount**. Эти методы позволяют управлять состоянием компонента и его поведением в разных стадиях жизненного цикла. Однако, в React большинство методов жизненного цикла стали устаревшими, и в будущем они могут быть удалены из React.

Функциональные компоненты в React могут использовать хуки, такие как **useState**, **useEffect** и **useContext**, которые позволяют управлять состоянием компонента и его поведением в разных стадиях жизненного цикла. Так же у разработчика есть возможность создавать собственные хуки. Они представляют из себя функции, названия которых принято начинать со слова **use**[HookName].

Еще одним отличием между классовым и функциональным компонентом в React является производительность. Функциональные компоненты обычно выполняются быстрее и используют меньше ресурсов,

чем классовые компоненты. Это связано с тем, что функциональные компоненты не имеют сложного синтаксиса и не создают дополнительных экземпляров классов при их вызове.

В React для передачи данных между компонентами используется концепция "подъема состояния" (lifting state up) и передача данных через props. Это означает, что если состояние нужно использовать в нескольких компонентах, оно должно быть определено в их общем родительском компоненте. Таким образом, родительский компонент является единственным источником правды для своих дочерних компонентов.

Когда происходит изменение состояния в дочернем компоненте, оно передается обратно в родительский компонент через колбэки, которые также передаются через props. Родительский компонент обновляет свое состояние и затем передает новые данные обратно в дочерние компоненты через props. Таким образом, все компоненты остаются в синхронизации.

Для того, чтобы компонент понимал, когда ему надо обновляться, он должен обзавестись состоянием (state). React отслеживает состояния компонентов и обновляет ваше приложение, когда состояние изменяется. Существует ошибочное мнение, что React изначально оптимизирован и будет обновлять только конкретные ноды, в которых произошли изменения. В действительности React только предоставляет инструменты для оптимизации рендера. Вопрос оптимизации будет рассмотрен более подробно в дальнейшем. Итак, мы понимаем, что нам необходимо создать в компоненте состояние, и для этого используется хук **useState**.

useState — это хук, который позволяет функциональным компонентам React иметь локальное состояние. Хук **useState** принимает начальное значение состояния и возвращает массив из двух элементов. Первый элемент - текущее значение состояния, второй элемент - функция, которая позволяет обновлять это состояние.

Например, можно использовать **useState** для хранения состояния введенного пользователем текста в текстовом поле. При изменении текста, функция обновления состояния будет вызываться, и новое состояние будет сохраняться.

Использование **useState** упрощает управление состоянием в функциональных компонентах и помогает избежать многих проблем, связанных с мутированием состояния напрямую.

Например, можно использовать **useState** для управления состоянием модального окна, которое должно отображаться или скрываться при нажатии на кнопку. Также можно использовать **useState** для управления текущим выбранным элементом списка, который может изменяться при нажатии на различные элементы списка.

Хук **useState** имеет много преимуществ перед использованием классовых компонентов и **setState**. Он более простой и лаконичный, позволяет избежать проблем, связанных с мутацией состояния напрямую, и позволяет функциональным компонентам иметь локальное состояние.

1.2.3 ОПТИМИЗАЦИЯ РЕНДЕРИНГА

При разработке веб-интерфейсов очень важно, чтобы у пользователя не возникало проблем с работой приложения, не зависимо от типа его устройства и вычислительной мощности. Современные приложения стараются уйти от излишней нагрузки устройств пользователей путем оптимизации рендеринга интерфейса. Первые веб-сайты при навигации по страницам внутри одного ресурса каждый раз отправляли пользователю новый документ, с часто

повторяющимися элементами. Условно, вы пользуетесь сайтом для поиска кулинарных рецептов и при просмотре новой страницы каталога у вас заново нагружает систему перерисовка статичных элементов: header, footer и прочее.

Данную проблему решает подход Single Page Application (SPA). SPA — это тип веб-приложения, в котором все необходимые ресурсы загружаются один раз при первом посещении сайта, после чего взаимодействие пользователя с приложением происходит без перезагрузки страницы. Такие приложения используют механизмы динамического обновления содержимого на странице, обычно используя технологии AJAX или WebSockets.

Одним из главных преимуществ SPA является более быстрый и плавный пользовательский опыт, поскольку приложение не перезагружается каждый раз при переходе между страницами. Кроме того, SPA позволяет более гибко управлять состоянием приложения, так как данные могут быть загружены только в тот момент, когда они действительно нужны, что снижает нагрузку на сервер и улучшает производительность.

В React SPA обычно строится с использованием маршрутизации (routing), где каждая "страница" приложения отображается через компонент, соответствующий определенному URL-адресу. React Router является одной из самых популярных библиотек для маршрутизации в React. Кроме того, для SPA необходимо обеспечить управление состоянием приложения, которое может быть реализовано с помощью управляемых компонентов (controlled components) и хуков состояния (state hooks).

Оптимизация производительности является важной задачей при разработке React-приложений, особенно для тех, у кого есть большие компоненты с множеством подкомпонентов, которые могут изменяться в зависимости от состояния. Для улучшения производительности и избегания ненужных повторных рендерингов React предоставляет **React.memo** и несколько хуков: **useCallback** и **useMemo**. Основа оптимизации рендеринга

это мемоизация, техника оптимизации вычислений, которая заключается в сохранении результатов выполнения функции для заданных входных параметров и использовании этих результатов при повторном вызове функции с теми же самыми входными параметрами, вместо повторного вычисления функции.

useCallback — это хук, который возвращает мемоизированную версию колбэка. Это полезно, когда вы передаете колбэк в дочерний компонент, который будет рендериться многократно. Мемоизированная версия колбэка сохраняется между рендерингами и не изменяется, если его зависимости не изменились.

useMemo — это хук, который позволяет мемоизировать результат вычисления сложных выражений. Это полезно, когда результат вычисления используется многократно в компоненте, но изменяется редко. Мемоизированная версия результата сохраняется между рендерингами и не изменяется, если его зависимости не изменились.

Важно понимать разницу между **useCallback** и **useMemo**. **useCallback** мемоизирует колбэк, тогда как **useMemo** мемоизирует результат вычисления. Если вы хотите мемоизировать результат сложных вычислений, используйте **useMemo**, а если вам нужно мемоизировать колбэк, используйте **useCallback**.

Например, если у вас есть компонент **MyComponent**, который получает свойства **prop1** и **prop2**, и использует их для вычисления значения **result**, вы можете использовать **useMemo**, чтобы мемоизировать результат:

```
function MyComponent({ prop1, prop2 }) {
  const result = useMemo(() => {
    // Сложные вычисления
    return /* результат вычислений */;
  }, [prop1, prop2]);

  return /* компонент с использованием результата */;
}
```

React.memo — это высокопроизводительный метод оптимизации для функциональных компонентов в React, который позволяет избежать повторного рендеринга компонента, если его входные параметры не изменились.

Когда компонент обернут в **React.memo**, React будет запоминать результаты рендеринга этого компонента и не будет повторно рендерить его, если его входные параметры не изменились. Это позволяет избежать ненужных операций рендеринга и улучшает производительность вашего приложения.

React.memo принимает два параметра: сам компонент и функцию, которая сравнивает предыдущие и текущие значения входных параметров компонента. Если эти значения равны, компонент не будет рендериться повторно. По умолчанию React сравнивает значения входных параметров по ссылке, поэтому если значения - объекты или массивы, вам может потребоваться написать свою собственную функцию сравнения, чтобы избежать нежелательного повторного рендеринга.

Важно отметить, что **React.memo** работает только с входными параметрами и не имеет никакого отношения к состоянию компонента или контексту. Если вы изменяете состояние или контекст внутри компонента, он все равно будет перерендериваться, даже если обернут в **React.memo**.

В целом, **React.memo** следует использовать только в тех случаях, когда вы уверены, что компонент не должен рендериться повторно, если его входные параметры не изменились. Это может произойти, например, когда компонент получает данные из статических источников или когда данные не меняются во время жизненного цикла компонента. На сегодняшний день довольно сложно перегрузить пользовательское устройство излишней мемоизацией, так как она требует лишь небольшое по современном меркам количество памяти и как правило ограничивается примитивным сравнением.

ГЛАВА 2. ПРАКТИЧЕСКАЯ ЧАСТЬ

После прохождения таких этапов, как идея, разработка требований и проектирования наступает этап разработки. Разработка предполагает написание компонентов интерфейса, внутренней логики, программирование, настройку рабочей среды и технологий. В качестве IDE была использована WebStorm, это мощный инструмент для веб-разработчика, он представляет из себя уже готовый комплект решений и, в отличии от аналогов, таких как Visual Studio Code, не требует длительной персональной настройки.

Убедитесь, что у вас установлен Node.js и npm. Вы можете проверить это, запустив в терминале команды:

```
node -v
npm -v
```

Если обе этих команды вернули версии, то Node.js и npm установлены корректно.

Установите create-react-app, выполнив следующую команду:

```
npm install -g create-react-app
```

Эта команда установит create-react-app глобально, что позволит вам создавать новые приложения с его помощью.

В связке с npm версии 5.2.0 и выше идет и npx (node package executor). Он служит для локального запуска программных пакетов. Создайте новое приложение с шаблоном TypeScript, выполнив команду:

```
npx create-react-app my-app --template typescript
```

Здесь my-app — это название вашего нового приложения. Вы можете использовать любое другое название.

После завершения установки необходимых зависимостей, перейдите в папку my-app и запустите приложение командой:

```
cd my-app
npm start
```

В результате выполнения команды будет получено приложение с уже готовым списком зависимостей и подключенной средой для тестирования, разработки и запуска локального сервера. Его необходимо отчистить от сторонних файлов, чтобы остался корневой компонент **App** и **index.ts** файл.

На данном этапе уже можно приступить к написанию функционала, но в рамках данной работы мы рассмотрим еще некоторые технологии для разработки приложений на профессиональном уровне.

2.1 ОРГАНИЗАЦИЯ СРЕДЫ РАЗРАБОТКИ

При разработке приложения важно понимать, что в больших командах с проектом будут взаимодействовать люди различных профессий с различными компетенциями, такие как дизайнеры, аналитики, заказчики / инвесторы. Для комфортного взаимодействия с приложением на всех этапах разработки существует решение в виде Storybook.

Storybook — это инструмент для разработки и тестирования компонентов пользовательского интерфейса в изоляции от остального приложения. С помощью Storybook можно создавать и документировать компоненты, просматривать их в различных состояниях и настройках, а также проверять их взаимодействие с различными данными.

Storybook поддерживает множество различных фреймворков и библиотек для создания пользовательского интерфейса, включая React, Vue, Angular, Ember и многие другие. Он предоставляет разработчикам и дизайнерам гибкий и удобный интерфейс для работы с компонентами, что позволяет быстро прототипировать и отлаживать новые компоненты, а также улучшать существующие.

Основные возможности Storybook:

1. Изоляция компонентов. Компоненты могут быть протестированы в изоляции от других компонентов и приложения в целом, что позволяет обнаружить и устранить проблемы до того, как они повлияют на работу всего приложения.
2. Различные состояния компонентов. Storybook позволяет отображать компоненты в различных состояниях, таких как различные размеры, цвета, типы ввода и т.д. Это помогает дизайнерам и разработчикам увидеть, как компоненты выглядят в различных условиях.
3. Документация компонентов. Storybook позволяет создавать документацию для компонентов, что делает их более доступными и понятными для других разработчиков. Документация может содержать описание компонента, его свойства и методы, примеры использования и т.д.
4. Тестирование компонентов. Storybook позволяет быстро проверять компоненты в различных условиях и настройках, что помогает обнаружить и устранить проблемы до того, как они повлияют на работу всего приложения.
5. Интеграция с другими инструментами. Storybook может быть легко интегрирован с другими инструментами для разработки и тестирования, такими как Jest, Cypress, Chromatic и т.д.

Использование Storybook может значительно ускорить разработку и улучшить качество пользовательского интерфейса, поскольку позволяет быстро и удобно работать с компонентами в изоляции от других частей приложения.

В рамках текущего проекта Storybook используется для просмотра компонентов и регрессионного тестирования. Благодаря интеграции с chromatic, Storybook может использоваться в учебных проектах на уровне качества коммерческих проектов. Интеграция позволяет расположить Storybook на отдельном сервере с различными правами доступа, командными взаимодействиями по типу проверок внесенных изменений и ревю. Самым мощным инструментом chromatic является скриншотное тестирование без каких-либо предварительных настроек. Chromatic просматривает все stories проекта и, при изменении, запускает сравнение имеющихся скриншотов с новыми. Таким образом будет заметно, если в каком-то компоненте неожиданно изменится содержание и ошибка не пройдет в финальную сборку.

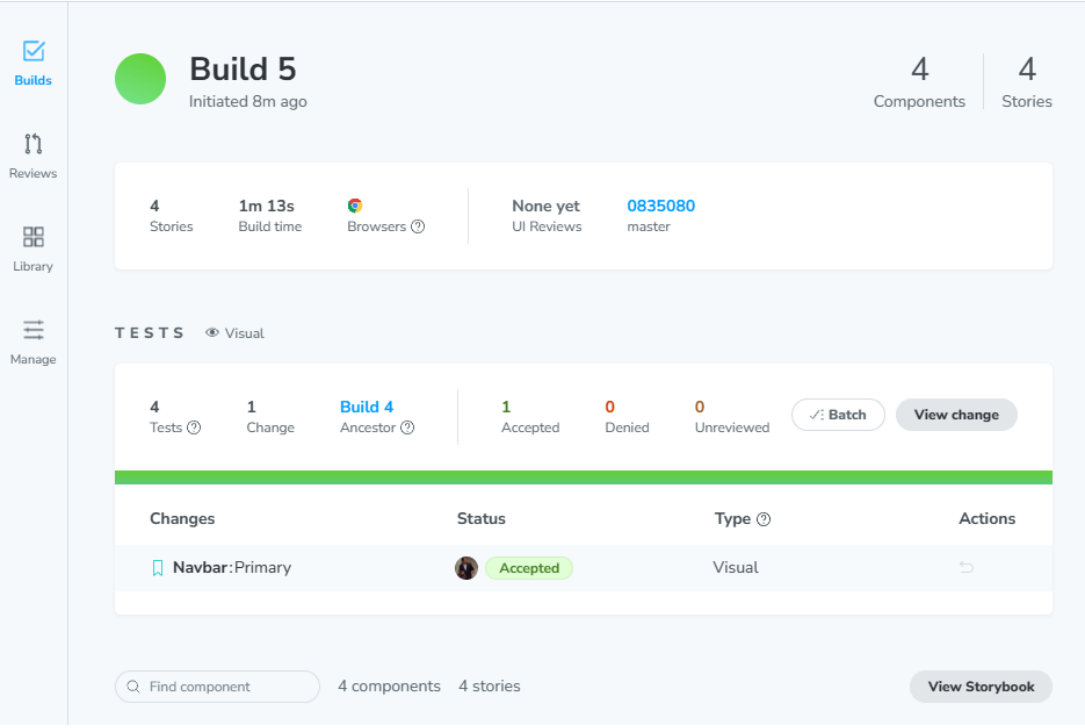

Рис.1 Интерфейс chromatic

При каждом изменении в проекте инициализируется новая сборка Storybook и его дальнейшая публикация. Как видно на рисунке 1, после публикации предоставляется возможность просмотреть изменения и верифицировать их. Так же можно просмотреть каждый отдельный компонент.

Как было указано ранее, для возможности просмотра компонента, необходимо написать stories. Stories или story представляет собой описание отдельного компонента для работы в Storybook. Рассмотрим пример stories для компонента Timer:

```
import React from 'react'
import { Meta, ComponentStory } from '@storybook/react'
import Timer from '../components/Timer'
import '../App.css'

export default {
  title: 'interface/Timer',
  component: Timer,
  argTypes: {
    backgroundColor: { control: 'color' }
  }
} as Meta<typeof Timer>

const Template: ComponentStory<typeof Timer> = (args) => <Timer {...args} />

export const Primary = Template.bind({})
Primary.args = {}
```

В качестве метаданных передается заголовок, который подразумевает группировки из нескольких stories и сам компонент. Затем создается моковый компонент и на его основе прописываются различные состояния. В данном случае описано единственное состояние **Primary**, но их может быть множество, например с использованием различных тем и стилей.

В данном проекте все stories будут находиться в общей директории, как подразумевается разработчиками при первом подключении Storybook к проекту. При работе с более крупными проектами все же рекомендуется хранить stories в одной директории с описываемым компонентом.

Таким образом Storybook в коммерческой разработке приложений может значительно упростить и ускорить процесс работы над пользовательским интерфейсом. Этот инструмент позволяет разработчикам и дизайнерам создавать и тестировать компоненты в изоляции от остальной части приложения, а также документировать их и проверять взаимодействие с различными данными.

В качестве средства для поддержания чистоты и общей стилистики написания приложения был использован ESLint. ESLint — это инструмент статического анализа кода JavaScript, который позволяет разработчикам обнаруживать и исправлять ошибки, несоответствия стандартам кодирования и потенциальные проблемы в их коде. Он используется для поддержания качества кода, улучшения читаемости и понимания кода, упрощения сопровождения кода, а также для повышения безопасности и производительности приложений.

ESLint позволяет настраивать набор правил для проверки кода, используя конфигурационные файлы, которые можно настроить под конкретные нужды проекта. Это означает, что разработчики могут определить свои собственные стандарты написания кода и требования к качеству, а также использовать стандарты написания кода, таких компаний как Airbnb, Google, Facebook и других.

Основные возможности и преимущества ESLint:

- Автоматическое обнаружение ошибок, несоответствий стандартам написания кода и потенциальных проблем в коде.
- Настраиваемые правила для проверки кода в соответствии с требованиями проекта и лучшими практиками.
- Интеграция с различными инструментами разработки, такими как редакторы кода, системы сборки, системы контроля версий и тестовые фреймворки.

- Возможность использования дополнительных плагинов и расширений для расширения функциональности.
- Удобство использования и быстрое обнаружение ошибок при сохранении изменений в коде.
- Помогает улучшить качество кода, обеспечивая единый стандарт написания кода и снижая вероятность ошибок в коде.

ESLint является популярным инструментом среди разработчиков и часто используется в коммерческих проектах для обеспечения высокого качества кода и улучшения производительности и безопасности приложений.

На данном этапе встает вопрос автоматизации работы всей рабочей инфраструктуры проекта. В дальнейшем также появится необходимость выполнения тестов, в связке с линтером и деплоем storybook становится актуальным вопрос реализации continuous integration и continuous delivery.

Continuous Integration (CI) — это процесс автоматической сборки и тестирования приложения после каждого коммита в репозиторий. Он позволяет обнаруживать ошибки и конфликты кода на ранних этапах разработки, что снижает затраты на их устранение и повышает качество кода.

Continuous Delivery (CD) — это процесс автоматической доставки приложения в производственную среду после успешной сборки и тестирования. Он позволяет быстро доставлять новые функции и исправления в продакшн, ускоряет процесс релиза и снижает затраты на доставку приложения.

Так как проект изначально был расположен на площадке GitHub, для реализации CI будет использоваться решение GitHub Actions. GitHub Actions — это набор инструментов для автоматизации сборки, тестирования, публикации и развертывания приложений, которые работают внутри репозитория на GitHub. С помощью GitHub Actions вы можете создавать

настраиваемые рабочие процессы, которые могут выполняться при различных событиях, таких как коммиты, создание веток, создание Pull Request и т.д.

Каждый рабочий процесс состоит из одного или нескольких шагов, которые можно выполнять на различных операционных системах и с использованием различных языков программирования. GitHub Actions предоставляет встроенные шаблоны для распространенных сценариев, а также позволяет настраивать свои собственные шаги и условия выполнения.

Основные преимущества GitHub Actions:

1. Простота настройки - GitHub Actions интегрируется непосредственно с репозиторием, поэтому не требуется настройка отдельного сервера для запуска сборок и тестов.
2. Гибкость - GitHub Actions позволяет настраивать рабочие процессы для выполнения различных задач на различных платформах и языках программирования.
3. Интеграция с другими инструментами - GitHub Actions может интегрироваться с другими инструментами для сборки, тестирования, деплоя и т.д., такими как Docker, AWS, Google Cloud, Azure и т.д.
4. Расширяемость - GitHub Actions позволяет создавать и использовать собственные шаги и настраивать процессы выполнения для удовлетворения уникальных потребностей.
5. Удобство - GitHub Actions предоставляет графический интерфейс для создания и управления рабочими процессами, а также позволяет запускать их из командной строки.

Для хостинга приложения используется сервис Vercel. Vercel — это облачный хостинг, специализирующийся на быстрой и простой разработке веб-сайтов и приложений. Он предлагает бесплатный и платный тарифы для хранения, развертывания и непрерывной доставки веб-сайтов и приложений.

С бесплатным тарифом на Vercel вы можете развернуть до трех проектов, используя 1000 часов бесплатного времени выполнения в месяц, а также бесплатный SSL-сертификат для каждого проекта. Также есть возможность автоматического масштабирования проектов при необходимости.

С помощью Vercel вы можете связать свой репозиторий GitHub, GitLab или Bitbucket и использовать Vercel для непрерывной доставки вашего кода в продакшн сборку. Это означает, что вы можете настроить процесс автоматического развертывания при каждом коммите в репозиторий. Также возможно использование специальных команд для запуска автоматического тестирования перед каждым развертыванием.

Vercel также предоставляет инструменты для управления каналами развертывания, настройки среды и просмотра логов. В целом, Vercel — это мощный инструмент для разработки веб-приложений и веб-сайтов, который обеспечивает быстрое развертывание и обновление, а также гибкие настройки непрерывной интеграции и доставки.

2.2 ИГРОВАЯ ДОСКА

В самом начале разработки необходимо реализовать игровое поле. Оно представляет из себя клеточное поле, формата 8 на 8. Определим поля объекта **Cell** (игровой клетки):

```
export class Cell {
  readonly x: number
  readonly y: number
  readonly color: Colors
  figure: Figure | null
```

```
board: Board
available: boolean
id: number
```

Поля **x** и **y** класса **Cell** содержат информацию о своих координатах по оси абсцисс ординат соответственно. Поле **color** указывает на цвет ячейки, так как в шахматах цвета клеток играют важную роль для координации фигур, например чернопольные и белопольные слоны. Поле **figure** указывает на находящуюся на клетке фигуру или ее отсутствии. **Board** содержит информацию о доске, к которой принадлежит клетка. Свойство **available** необходимо для подсвечивания клетки при выборе фигуры на доске, если у игровой фигуры есть возможность походить на клетку, свойство примет значение **true** и клетка подсветится доступной. **Id** содержит уникальный номер клетки.

Рассмотрим методы класса **Cell**:

- isEmpty – возвращает false, если на клетке находится фигура.

```
isEmpty () {
    return this.figure === null
  }
```

- isEmptyVertical – проверяет на доступность клетки по вертикали.

```
isEmptyVertical (target: Cell): boolean {
    if (this.x !== target.x) { return false }

    const min = Math.min(this.y, target.y)
    const max = Math.max(this.y, target.y)

    for (let y = min + 1; y < max; y++) {
      if (!this.board.getCell(this.x, y).isEmpty()) {
        return false
      }
    }
    return true
}
```

- isEmptyHorizontal - проверяет на доступность клетки по горизонтали.

```
isEmptyHorizontal (target: Cell): boolean {
    if (this.y !== target.y) { return false }
```

```
    const min = Math.min(this.x, target.x)
    const max = Math.max(this.x, target.x)

    for (let x = min + 1; x < max; x++) {
      if (!this.board.getCell(x, this.y).isEmpty()) {
        return false
      }
    }
    return true
  }
```

- isEmptyDiagonal - проверяет на доступность клетки по диагонали.

```
isEmptyDiagonal (target: Cell): boolean {
    const absX = Math.abs(target.x - this.x)
    const absY = Math.abs(target.y - this.y)
    if (absY !== absX) { return false }

    const dy = this.y < target.y ? 1 : -1
    const dx = this.x < target.x ? 1 : -1

    for (let i = 1; i < absY; i++) {
      if (!this.board.getCell(this.x + dx * i, this.y + dy *
i).isEmpty()) { return false }
    }
    return true
  }
```

- isEnemy – если на клетке находится фигура, возвращает true в случае, когда цвет не совпадает с выбранной фигурой.

```
isEnemy (target : Cell) : boolean {
    if (target.figure) {
      return this.figure?.color !== target.figure.color
    }
    return false
  }
```

- setFigure – устанавливает фигуру на клетку.

```
setFigure (figure : Figure) {
    this.figure = figure
    this.figure.cell = this
  }
```

- addLostFigure – добавляет фигуру в массив утерянных фигур.

```
addLostFigure (figure : Figure) {
    figure.color === Colors.BLACK
      ? this.board.lostBlackFigures.push(figure)
      : this.board.lostWhiteFigures.push(figure)
  }
```

- moveFigure – перемещает фигуру на клетку.

```
moveFigure (target: Cell) {
    if (this.figure && this.figure?.canMove(target)) {
      this.figure?.moveFigure(target)
      if (target.figure && (target.figure.color !==
this.figure.color)) {
        this.addLostFigure(target.figure)
      }
      target.setFigure(this.figure)
      this.figure = null
    }
  }
```

Рассмотрим класс **Board**:

```
cells: Cell[][] = []
lostBlackFigures: Figure[] = []
lostWhiteFigures: Figure[] = []
```

- cells – двумерный массив столбцов и строк, состоящий из клеток.
- lostBlackFigures – массив утерянных черных фигур.
- lostWhiteFigures – массив утерянных белых фигур.

Рассмотрим методы класса **Board**:

- highlightCells (selectedCell: Cell | null) – подсвечивает доступные для хода клетки.

```
  public highlightCells (selectedCell: Cell | null) {
    for (let i = 0; i < this.cells.length; i++) {
      const row = this.cells[i]
      for (let j = 0; j < row.length; j++) {
        const target = row[j]
        target.available = !!selectedCell?.figure?.canMove(target)
      }
    }
  }
```

- getCopyBoard – создает копию текущей доски.

```
public getCopyBoard (): Board {
    const newBoard = new Board()
    newBoard.cells = this.cells
    newBoard.lostBlackFigures = this.lostBlackFigures
    newBoard.lostWhiteFigures = this.lostWhiteFigures
    return newBoard
  }
```

- initCells – инициализирует клетки доски.

```
public initCells () {
    for (let i = 0; i < 8; i++) {
      const row: Cell[] = []
      for (let j = 0; j < 8; j++) {
        if ((i + j) % 2 !== 0) {
          row.push(new Cell(this, j, i, Colors.BLACK, null))
        } else {
          row.push(new Cell(this, j, i, Colors.WHITE, null))
        }
      }
      this.cells.push(row)
    }
  }
```

- getCell (x:number, y:number) – возвращает клетку по заданным координатам.

```
public getCell (x:number, y:number) {
    return this.cells[y][x]
  }
```

Board так же содержит набор приватных методов для расстановки фигур.

```
private addPawns () {
  for (let i = 0; i < 8; i++) {
    new Pawn(Colors.BLACK, this.getCell(i, 1))
    new Pawn(Colors.WHITE, this.getCell(i, 6))
  }
}

private addKings () {
  new King(Colors.BLACK, this.getCell(4, 0))
  new King(Colors.WHITE, this.getCell(4, 7))
}

private addQueens () {
  new Queen(Colors.BLACK, this.getCell(3, 0))
  new Queen(Colors.WHITE, this.getCell(3, 7))
}

private addRooks () {
  new Rook(Colors.BLACK, this.getCell(0, 0))
  new Rook(Colors.WHITE, this.getCell(0, 7))
  new Rook(Colors.BLACK, this.getCell(7, 0))
  new Rook(Colors.WHITE, this.getCell(7, 7))
}

private addBishops () {
  new Bishop(Colors.BLACK, this.getCell(2, 0))
  new Bishop(Colors.WHITE, this.getCell(2, 7))
  new Bishop(Colors.BLACK, this.getCell(5, 0))
  new Bishop(Colors.WHITE, this.getCell(5, 7))
}
```

```
private addKnights () {
  new Knight(Colors.BLACK, this.getCell(1, 0))
  new Knight(Colors.WHITE, this.getCell(1, 7))
  new Knight(Colors.BLACK, this.getCell(6, 0))
  new Knight(Colors.WHITE, this.getCell(6, 7))
}

public addFigures () {
  this.addPawns()
  this.addBishops()
  this.addKings()
  this.addKnights()
  this.addRooks()
  this.addQueens()
}
```

Каждый конкретный метод определяет местоположение и цвет для типов фигур, после чего, уже в публичном методе, все фигуры расставляются по доске.

Так же определим цвета в качестве **enum** перечисления и создадим класс игрока с полем цвета.

2.3 ШАХМАТНЫЕ ФИГУРЫ

Самая большая часть логики приложение содержится в игровых фигурах. Рассмотрим шахматные фигуры и правила их перемещения:

- Пешка - В свой первый ход может ходить как на одну, так и на две клетки вперед. В любой ход, кроме своего первого, может ходить лишь на одну клетку вперед. Может производить взятие только на одну клетку по диагонали по траектории движения. Не может ходить вперед, если следующая перед ней клетка занята другой

фигурой. По достижении последней горизонтали, становится Ферзем со всеми возможностями этой фигуры.

- Король - Может ходить в любом направлении в пределах одной клетки. Может осуществить рокировку
- Ферзь - Может двигаться на любое количество клеток по вертикали, горизонтали и диагонали.
- Ладья - Может двигаться на любое количество клеток в пределах поля по вертикали или горизонтали.
- Конь - Может двигаться через препятствия. Конь двигается на две клетки по вертикали и затем на одну клетку по горизонтали, или наоборот, на две клетки по горизонтали и на одну клетку по вертикали.
- Слон - Может двигаться на любое количество клеток по диагонали.

Все фигуры кроме коней не могут перешагивать через другие фигуры, а также все фигуры, кроме пешек, могут забрать вражескую фигуру на клетке, которая находится на траектории движения.

Сначала реализуем класс, от которого в дальнейшем будут наследоваться все игровые фигуры.

```
export class Figure {
  color: Colors
  logo: typeof logo | null
  cell: Cell
  name: FigureNames
  id: number
```

У каждой фигуры есть собственный цвет, иконка, информация о клетке, на которой она находится, имя и уникальный id. Так же определим основные правила для перемещения каждой фигуры, это невозможность атаковать короля и фигуры своего цвета:

```
canMove (target: Cell) : boolean {
    if (target.figure?.color === this.color) { return false }
    if (target.figure?.name === FigureNames.KING) { return false }
```

```
    return true
  }
```

Для наименований фигур так же создается перечисление:

```
export enum FigureNames {
  FIGURE = 'Фигура',
  KING = 'Король',
  KNIGHT = 'Конь',
  PAWN = 'Пешка',
  QUEEN = 'Ферзь',
  ROOK = 'Ладья',
  BISHOP = 'Слон',
}
```

На данном этапе можно приступить к реализации правил для каждой фигуры. Каждой фигуре присваивается имя из перечисления и иконка, при помощи **super** вызывается конструктор родительского класса для передачи цвета и клетки фигуры. Опишем правила для перемещения для каждой фигуры:

- Пешка:

```
canMove (target: Cell): boolean {
    if (!super.canMove(target)) { return false }
    const direction = this.cell.figure?.color === Colors.BLACK ? 1
: -1
    const firstStepDirection = this.cell.figure?.color ===
Colors.BLACK ? 2 : -2

    if (
      target.y === this.cell.y + direction
      && target.x === this.cell.x
      && target.isEmpty()
    ) {
      return true
    }

    if (this.isFirstStep
      && target.y === this.cell.y + firstStepDirection
      && this.cell.board.getCell(this.cell.x,this.cell.y +
direction).isEmpty()
      && target.x === this.cell.x
      && target.isEmpty()
    ) {
      return true
    }

    if (target.y === this.cell.y + direction &&
      (target.x === this.cell.x + 1 || target.x === this.cell.x -
1) &&
      this.cell.isEnemy(target)) {
```

```
      return true
    }

    return false
  }
```

- Король:

```
canMove (target: Cell): boolean {
    console.log(this.cell.board.getCell(0, this.color ===
Colors.WHITE ? 7 : 0).figure?.isFirstStep)
    if (!super.canMove(target)) { return false }
    const dx = Math.abs(this.cell.x - target.x)
    const dy = Math.abs(this.cell.y - target.y)

    if (this.isFirstStep && (   // Castling rules
      (target.x === 2
        && this.cell.board.getCell(1, this.cell.y).isEmpty()
        && this.cell.board.getCell(2, this.cell.y).isEmpty()
        && this.cell.board.getCell(3, this.cell.y).isEmpty()
        && this.cell.board.getCell(0, this.color === Colors.WHITE
? 7 : 0).figure?.isFirstStep
      )
      || (target.x === 6
        && this.cell.board.getCell(5, this.cell.y).isEmpty()
        && this.cell.board.getCell(6, this.cell.y).isEmpty()
        && this.cell.board.getCell(7, this.color === Colors.WHITE
? 7 : 0).figure?.isFirstStep
      )
      )
      && (this.color === Colors.WHITE
        ? target.y === 7
        : target.y === 0
      )
    ) {
      return true
    }

    if (dx <= 1 && dy <= 1) {
      return true
    }
    return false
  }
```

- Ферзь:

```
canMove (target: Cell): boolean {
    if (!super.canMove(target)) { return false }
    if (this.cell.isEmptyVertical(target)) { return true }
    if (this.cell.isEmptyHorizontal(target)) { return true }
    if (this.cell.isEmptyDiagonal(target)) { return true }
    return false
  }
```

- Ладья:

```
canMove (target: Cell): boolean {
  if (!super.canMove(target)) { return false }
  if (this.cell.isEmptyVertical(target)) { return true }
  if (this.cell.isEmptyHorizontal(target)) { return true }
  return false
}
```

- Конь:

```
canMove (target: Cell): boolean {
  if (!super.canMove(target)) { return false }
  const dx = Math.abs(this.cell.x - target.x)
  const dy = Math.abs(this.cell.y - target.y)

  return (dx === 1 && dy === 2) || (dx === 2 && dy === 1)
}
```

- Слон:

```
canMove (target: Cell): boolean {
  if (!super.canMove(target)) { return false }
  if (this.cell?.isEmptyDiagonal(target)) { return true }
  return false
}
```

Таким образом, все основные правила для игры в шахматы были реализованы. Логика приложения на этом ограничивается, но ее можно развивать и дополнять в будущем, например добавить возможность игры в шашки. В данном варианте используются три основных принципа ООП, а таже возможности языка typescript, такие как enum перечисления и типизация параметров и объектов.

2.4 ИНТЕРФЕЙС

Интерфейс приложения был разработан с использованием компонентного подхода. Это означает, что html документ не является монолитным, а собирается из множества модулей. Это дает возможность создавать обособленные модули и избегать избыточной связанности (coupling), сохраняя сцепленность между компонентами (cohesion), объединяя их в компонентах более высокого порядка.

Данное приложение состоит из следующих компонентов:

- App – корневой компонент. Минималистичен, представляет из себя сборку готовых компонентов.

```
const App = () => {
  return (
    <div className="app">
      <Navbar className="header" />
      <ChessBoard />
    </div>
  )
}
```

- Navbar – Представляет из себя заголовок приложения.

```
const Navbar = ({ className }: NavbarProps) => {
  return (
    <div className={className}>
      <div>
        <p className='header-text'>КЛАССИЧЕСКИЕ ШАХМАТЫ</p>
      </div>
    </div>
  )
}
```

- ChessBoard – большой компонент, содержит состояния и обработчики событий. Состоит из доски, таймера и двух компонентов с утерянными фигурами.

```
const ChessBoard = () => {
  const [board, setBoard] = useState(new Board())
  const [whitePlayer] = useState(new Player(Colors.WHITE))
  const [blackPlayer] = useState(new Player(Colors.BLACK))
```

```
  const [currentPlayer, setCurrentPlayer] = useState<Player |
null>(null)

  useEffect(() => {
    restart()
    setCurrentPlayer(whitePlayer)
  }, [])

  function restart () {
    const newBoard = new Board()
    newBoard.initCells()
    newBoard.addFigures()
    setBoard(newBoard)
    setCurrentPlayer(whitePlayer)
  }

  function changePlayer () {
    setCurrentPlayer(currentPlayer?.color === Colors.WHITE ?
blackPlayer : whitePlayer)
  }

  return (
    <div className="content">
      <Timer currentPlayer={currentPlayer} restart={restart}/>
      <BoardComponent
        board={board}
        setBoard={setBoard}
        currentPlayer={currentPlayer}
        changePlayer={changePlayer}
      />
      <div>
        <LostFigures title={'Черные фигуры'}
figures={board.lostBlackFigures}/>
      </div>
      <div>
        <LostFigures title={'Белые фигуры'}
figures={board.lostWhiteFigures}/>
      </div>
    </div>
  )
}
```

- Timer – Реализация игровых часов с кнопкой для перезапуска игры.

```
const Timer: FC<TimerProps> = ({ currentPlayer, restart,
playersTime = 300 }) => {
  const [blackTime, setBlackTime] = useState(playersTime)
  const [whiteTime, setWhiteTime] = useState(playersTime)
  const timer = useRef<null | ReturnType<typeof
setInterval>>(null)

  useEffect(() => {
    startTimer()
  }, [currentPlayer])

  function startTimer () {
    if (timer.current) {
      clearInterval(timer.current)
    }
```

```
    const callback = currentPlayer?.color === Colors.WHITE
      ? decrementWhiteTimer
      : decrementBlackTimer
    timer.current = setInterval(callback, 1000)
  }

  function decrementBlackTimer () {
    setBlackTime(prev => prev > 0 ? prev - 1 : 0)
  }

  function decrementWhiteTimer () {
    setWhiteTime(prev => prev > 0 ? prev - 1 : 0)
  }

  const handleRestart = () => {
    setWhiteTime(playersTime)
    setBlackTime(playersTime)
    restart()
  }

  return (
    <div>
      <div>
        <button className="btn" data-testid="Restart_Button"
onClick={handleRestart}>Restart game</button>
      </div>
      <h2 data-testid="Black_Timer">Черные - {Math.floor(blackTime
/ 60)}:{(blackTime % 60) < 10 ? '0' + (blackTime % 60) :
(blackTime % 60)}</h2>
      <h2 data-testid="White_Timer">Белые - {Math.floor(whiteTime
/ 60)}:{(whiteTime % 60) < 10 ? '0' + (whiteTime % 60) :
(whiteTime % 60)}</h2>
        {whiteTime === 0
          ? (
                <div data-testid="Black_Timeout_Win">
                    Черные победили по времени
                </div>
            )
          : blackTime === 0
            ? (
                <div data-testid="White_Timeout_Win">
                    Белые победили по времени
                </div>)
            : null
        }
    </div>
  )
}
```

- LostFigures – Компонент принимает массив из фигур и отображает их списком.

```
const LostFigures: FC<LostFiguresProps> = ({ title, figures }) =>
{
  return (
    <div className="lost">
      <h3 data-testid = "Lost_Figures-title">{title}</h3>
      {figures.map(figure =>
      <div className="lostFigure" key={figure.id} data-
testid="Lost_Figures">
          {figure.logo && <img width={20} height={20}
src={figure.logo} alt={figure.name} data-testid="Lost_Figure"/>}
```

```
{figure.name}
      </div>
      )}
    </div>
  )
}
```

- BoardComponent – Игровая доска, состоящая из клеток.

```
const BoardComponent: FC<BoardProps> = ({ board, setBoard,
currentPlayer, changePlayer }) => {
  const [selectedCell, setSelectedCell] = useState<Cell |
null>(null)

  const click = (cell: Cell) => {
    if (selectedCell && selectedCell !== cell &&
selectedCell.figure?.canMove(cell)) {
      selectedCell.moveFigure(cell)
      changePlayer()
      setSelectedCell(null)
    } else {
      if (cell.figure?.color === currentPlayer?.color) {
        setSelectedCell(cell)
      }
    }
  }

  useEffect(() => {
    highlightCells()
  }, [selectedCell])

  function highlightCells () {
    board.highlightCells(selectedCell)
    updateBoard()
  }

  function updateBoard () {
    const newBoard = board.getCopyBoard()
    setBoard(newBoard)
  }

  return (
    <div>
      <h3>Текущий игрок {currentPlayer?.color}</h3>
        <div className="board">
          {board.cells.map((row, index) =>
            <React.Fragment key={index}>
              {row.map(cell =>
              <CellComponent
                click={click}
                cell={cell}
                key={cell.id}
                selected={cell.x === selectedCell?.x && cell.y ===
selectedCell?.y}
              />
              )}
            </React.Fragment>
          )}
        </div>
  </div>
  )
}
```

- CellComponent – Игровая клетка, составная деталь игровой доски.

```
const CellComponent: FC<CellProps> = ({ cell, selected, click })
=> {
  return (
    <div
      className={['cell', cell.color, selected ? 'selected' :
''].join(' ')}
      onClick={() => click(cell)}
      style={{ background: cell.available && cell.figure ? 'green'
: '' }}
    >
      {cell.available && !cell.figure && <div
className={'available'}/>}
      {cell.figure?.logo && <img src={cell.figure.logo} alt=""/>}
    </div>
  )
}
```

Данные компоненты реализуют концепцию слабой связанности и высокой сцепленности. Часть компонентов полностью изолирована друг от друга, они отвечают одному концепту шахматного приложения и являются универсальными и переиспользуемыми. В случае добавления новых игр, в компоненты с утерянными фигурами можно передавать массивы с новыми фигурами и они будут выполнять свой функционал, таймер же вовсе не потребует никаких изменений, так как он полностью изолирован.

Благодаря работе React с состояниями, компоненты будут перерисовываться динамически, а проблема, связанная с лишними перерисовками некоторых компонентов, была решена при помощи мемоизации. Так же все компоненты содержат интерфейсы, которые описывают их параметры и типы данных.

ЗАКЛЮЧЕНИЕ

В рамках данной работы было разработано веб-приложение для игры в классические шахматы. Приложение построено на современном стандарте SPA, оптимизированно с точки зрения потребления вычислительных ресурсов и отвечает по следующим критериям:

- Обустроена профессиональная среда для разработки и дальнейшей поддержки приложения.
- Реализована DevOps методология с использованием CI/CD pipeline.
- Разработано поля для игры, которое в дальнейшем можно будет использовать и для других настольных игр.
- Разработан универсальный класс, на котором основываются все фигуры.
- Разработаны шахматные фигуры и реализованы игровые механики, такие как рокировка и превращение пешек в ферзя на последней горизонтали.
- Добавлен таймер для контроля времени и фиксирование потерянных фигур

Также были затронуты преимущества технологий typescript и React, приложение было разработано согласно стандартам, предоставленными разработчиками Microsoft и Meta.

СПИСОК ЛИТЕРАТУРЫ

1. Alex Banks, Eve Porcello. Learning React / Modern Patterns for Developing React Apps. – Себастопол, Калифорния, США : O'Reilly Media, Inc., 2020. – 295 с. – ISBN 9781492051725.
2. Adam D. Scott. JavaScript Everywhere. – Себастопол, Калифорния, США : O’Reilly Media, Inc., 2020. – 318 с. – ISBN 9781492046981.
3. Черный Борис. Профессиональный TypeScript / Разработка масштабируемых JavaScript-приложений. — СПб.: Питер, 2020
4. Facebook Open Source. Документация библиотеки React. 2022. Meta Platforms, Inc. URL: https://ru.reactjs.org/docs/getting-started.html (дата обращения: 20.05.2022)
5. TINKOFF. 12 советов по внедрению TypeScript в React-приложениях, - 2020 / habr.com. URL: https://habr.com/ru/company/tinkoff/blog/505488/ (дата обращения 09.06.2022)
6. UlbiTV. CI CD наглядные примеры, 2022 / YouTube. URL: https://youtu.be/ANj7qUgzNq4 (дата обращения 20.08.2022)
7. Github. Документация Github Actions. – 2022 / GitHub, Inc. URL: https://docs.github.com/ru/actions (дата обращения: 20.08.2022)
8. HABR – cleverowl. Бесплатные хостинги для веб-разработчиков. - 2020 / habr.com. URL: https://habr.com/ru/post/535168/ (дата обращения 25.06.2022)
9. HABR - WinPooh73. Определяем веса шахматных фигур регрессионным анализом, - 2015 // habr.com. URL: https://habr.com/ru/post/254753/ (дата обращения 20.04.2022)
10. ФИДЕ. Правила шахмат. - 2022 URL: http://chess.sainfo.ru/lawsr.html (дата обращения 23.05.2022)

11. Checkly Документация Chekly. – 2022 / Checkly Inc. URL: https://www.checklyhq.com/docs/ (дата обращения: 20.08.2022)
12. HABR - OTUS Что такое CI/CD? / Разбираемся с непрерывной интеграцией и непрерывной поставкой. - 2020 // habr.com. URL: https://habr.com/ru/company/otus/blog/515078/ (дата обращения 21.08.2022)
13. Ляхов Александр Федорович Информационный анализ игры в шахматы // КИО. - 2005. №5. URL: https://cyberleninka.ru/article/n/informatsionnyy-analiz-igry-v-shahmaty (дата обращения: 10.02.2023).
14. Мамбетов Р.А. ПОСТРОЕНИЕ ПОЛЬЗОВАТЕЛЬСКИХ ИНТЕРФЕЙСОВ ВЕБ-ПРИЛОЖЕНИЙ С ПОМОЩЬЮ БИБЛИОТЕКИ REACT // Экономика и социум. - 2019. №5 (60). URL: https://cyberleninka.ru/article/n/postroenie-polzovatelskih-interfeysov-veb-prilozheniy-s-pomoschyu-biblioteki-react (дата обращения: 21.02.2023).
15. Яровая Екатерина Владимировна АРХИТЕКТУРНЫЕ РЕШЕНИЯ В БИБЛИОТЕКИ РЕАКТ // Столыпинский вестник. - 2022. №5. URL: https://cyberleninka.ru/article/n/arhitekturnye-resheniya-v-biblioteki-reakt (дата обращения: 21.02.2023).
16. Князев Илья Вадимович, Коптева Анна Витальевна РАЗРАБОТКА И АНАЛИЗ ПОСТЕПЕННОГО ВНЕДРЕНИЯ ПРОВЕРКИ И ВЫВОДА ТИПОВ ДАННЫХ С ПОМОЩЬЮ ПАРАМЕТРИЧЕСКОГО ПОЛИМОРФИЗМА И ИСПОЛЬЗОВАНИЯ ЯЗЫКА ПРОГРАММИРОВАНИЯ TYPESCRIPT // European research. - 2021. №1 (38). URL: https://cyberleninka.ru/article/n/razrabotka-i-analiz-postepennogo-vnedreniya-proverki-i-vyvoda-tipov-dannyh-s-pomoschyu-parametricheskogo-polimorfizma-i (дата обращения: 15.04.2023).
17. Берьянов Максим Сергеевич, Салахов Илья Равильевич, Иванов Михаил Дмитриевич ИССЛЕДОВАНИЕ СОВРЕМЕННОГО СТЕКА REACT РАЗРАБОТКИ В 2022 ГОДУ // Столыпинский вестник. - 2022. №8. URL:

https://cyberleninka.ru/article/n/issledovanie-sovremennogo-steka-react-razrabotki-v-2022-godu (дата обращения: 15.04.2023).

18. Ивашкин Д. И., Масюков В. А. Сравнительный анализ инструментов разработки web-приложений // Программные продукты и системы. - 2002. №2. URL: https://cyberleninka.ru/article/n/sravnitelnyy-analiz-instrumentov-razrabotki-web-prilozheniy (дата обращения: 15.04.2023).
19. Бетеев К.Ю., Муратова Г.В. КОНЦЕПЦИЯ VIRTUAL DOM В БИБЛИОТЕКЕ REACT.JS // ИВД. - 2022. №3 (87). URL: https://cyberleninka.ru/article/n/kontseptsiya-virtual-dom-v-biblioteke-react-js (дата обращения: 19.04.2023).
20. Скопин Игорь Николаевич О ФУНКЦИОНАЛЬНОМ ПРОГРАММИРОВАНИИ И МОДУЛЬНОСТИ // Проблемы информатики. - 2019. №3 (44). URL: https://cyberleninka.ru/article/n/o-funktsionalnom-programmirovanii-i-modulnosti (дата обращения: 19.04.2023).