МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
федеральное государственное бюджетное образовательное учреждение высшего образования
«Тольяттинский государственный университет»

Кафедра Прикладная математика и информатика
(наименование)

01.03.02 Прикладная математика и информатика
(код и наименование направления подготовки / специальности)

Компьютерные технологии и математическое моделирование
(направленность (профиль) / специализация)

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА (БАКАЛАВРСКАЯ РАБОТА)

на тему: Разработка и интеграция UI библиотеки React компонентов для повышения качества и скорости разработки веб-приложений

Обучающийся И.А. Яницкий _____
(Инициалы Фамилия) (Личная подпись)

Руководитель канд. пед. наук, доцент, Т. А. Агошкова
(ученая степень (при наличии), ученое звание (при наличии), Инициалы Фамилия)

Консультант канд. пед. наук, доцент, А. В. Егорова
(ученая степень (при наличии), ученое звание (при наличии), Инициалы Фамилия)

Тольятти 2025

Аннотация

Выпускная квалификационная работа посвящена разработке и интеграции пользовательской библиотеки UI-компонентов для веб-приложений на основе технологий React и TypeScript. Актуальность темы обусловлена растущими требованиями к скорости и качеству разработки пользовательских интерфейсов, а также необходимостью стандартизации визуальных и функциональных элементов в рамках современных фронтенд-проектов.

В рамках работы была поставлена цель – создать модульную и переиспользуемую библиотеку компонентов, способствующую ускорению разработки, повышению читаемости и поддерживаемости кода, а также обеспечению единого пользовательского опыта. В ходе выполнения проекта был проведён обзор существующих решений и подходов к созданию UI-библиотек, выявлены их преимущества и недостатки, на основе чего сформулированы требования к собственной реализации.

Разработанная библиотека строится по принципам масштабируемости, атомарности и независимости компонентов, что позволяет эффективно использовать её в различных проектах. Также реализована система документации и демонстрации компонентов для упрощения внедрения в команды разработки. В качестве подтверждения эффективности решения проведена интеграция библиотеки в реальное веб-приложение, по результатам которой зафиксированы улучшения в процессе разработки: снижение количества повторяющегося кода, повышение согласованности интерфейсов и сокращение времени на создание новых элементов UI.

Практическая значимость работы заключается в возможности дальнейшего расширения и адаптации библиотеки под нужды конкретных команд или компаний, а также в использовании полученных наработок в построении внутренних дизайн-систем.

Abstract

The title of the graduation work is *Development and Integration of UI Library React Components to Improve the Quality and Speed of Web Application Development*.

The graduation work consists of an introduction, three parts, 12 figures, 3 tables, a conclusion, and a list of 42 references including foreign sources.

The aim of this graduation work is to develop a reusable UI component library that improves frontend development speed and quality and ensures consistency.

The object of the graduation work is the process of creating standardized user interface components for web applications.

The subject of the graduation work is the design and implementation of a scalable UI library using React and TypeScript.

The key issue of the graduation work is to enhance development efficiency, maintainability, and interface uniformity in frontend projects.

The graduation work may be divided into three logically connected parts: theoretical overview, implementation, and integration.

The first part outlines the relevance of UI libraries, reviews existing solutions, and defines requirements for a custom library.

The second part describes the development of atomic, reusable components, documentation tools, and a demo environment.

The third part covers integration into a real project and shows improvements such as reduced code duplication, better interface consistency, and faster UI development.

In conclusion, we would like to stress that the developed library is practical and adaptable to various project needs. Nevertheless, further improvements are possible. Nevertheless, more experimental data are required.

The work is of interest for a wide circle of readers involved in frontend development and component-based UI design.

Содержание

Введение

В современном цифровом мире компании всё чаще сталкиваются с необходимостью создания собственных веб-приложений – как для обеспечения эффективного взаимодействия с клиентами, так и для решения внутренних задач. Примерами таких решений являются интернет-магазины, платформы онлайн-услуг, внутренние корпоративные системы управления задачами, документооборотом, персоналом и финансами. Усложнение бизнес-процессов приводит к росту требований к качеству и функциональности пользовательского интерфейса. Чтобы соответствовать этим требованиям, в разработке всё шире применяются современные инструменты и фреймворки, такие как React, Angular и Vue.js. Эти технологии обеспечивают модульность, повторное использование компонентов и масштабируемость кода [14].

Тем не менее, с увеличением количества компонентов и усложнением архитектуры проекта даже при использовании мощных инструментов возникают новые проблемы: фрагментарность решений, дублирование логики, несогласованность дизайна, снижение читаемости кода. Эти факторы увеличивают технический долг, замедляют внедрение новых функций и снижают стабильность продукта. Возникает необходимость в создании унифицированного решения, которое обеспечит стандартизированный подход к разработке компонентов пользовательского интерфейса, повысит производительность команды, снизит количество ошибок и упростит сопровождение кода [1][5].

В качестве ответа на эту потребность в рамках данной работы предлагается разработка UI-библиотеки компонентов на базе React.

Объектом исследования является процесс создания пользовательских интерфейсов веб-приложений [24], а предметом – методы и практики стандартизации разработки компонентов с использованием React [8].

Основная цель исследования – повысить качество и скорость разработки веб-интерфейсов путём создания набора переиспользуемых компонентов,

реализованных в едином стиле и по единым архитектурным принципам. Такая библиотека позволит разработчикам сосредоточиться на логике приложения, снижая затраты на повторную реализацию типовых элементов интерфейса и обеспечивая их визуальное и поведенческое единообразие [15].

Для достижения поставленной цели решаются следующие задачи:

- анализ существующих подходов к созданию UI-библиотек,
- проектирование архитектуры компонентов и структуры библиотеки,
- реализация набора компонентов с учётом масштабируемости и адаптируемости,
- интеграция библиотеки в реальный проект и оценка её влияния на процесс разработки.

Методологическая база исследования включает инженерный подход: анализ потребностей, проектирование решений, их реализация и апробация. В процессе разработки проводилось тестирование библиотеки в условиях реальной разработки, что позволило получить объективную оценку её эффективности и выявить области для дальнейшего развития.

Практическая значимость работы заключается в возможности использования библиотеки в широком спектре проектов для повышения качества интерфейсов, упрощения командной разработки и снижения временных и ресурсных затрат на поддержку [21].

Выпускная квалификационная состоит из введения, трёх разделов, заключения, списка литературы и приложений. В первом разделе рассматриваются теоретические основы и анализируются существующие решения в области UI-библиотек. Второй раздел описывает требования к разрабатываемому продукту, архитектуру компонентов и процесс реализации. В третьем разделе представлены результаты внедрения библиотеки в реальный проект, анализ эффективности и предложения по дальнейшему улучшению. В заключении подводятся итоги работы и формулируются выводы по результатам исследования.

1 Формирование концепции UI библиотеки

1.1 Анализ существующих UI библиотек на основе React

Создание и дальнейшая разработка интерфейсов веб-приложений представляет собой сложный и многогранный процесс, который включает в себя проектирование, реализацию и оптимизацию пользовательского взаимодействия. Одним из ключевых аспектов этого процесса является использование компонентов – самостоятельных элементов макета, которые выполняют определенную логику, получают, хранят и обрабатывают необходимые данные, а также обладают индивидуальным внешним видом. Компонентный подход позволяет создавать гибкие, переиспользуемые элементы, которые могут легко комбинироваться и модифицироваться в рамках различных проектов.

Однако при длительной работе над сложными веб-приложениями, особенно в командах, состоящих из нескольких разработчиков, часто возникает проблема дублирования кода и функционала. Разные участники проекта могут создавать компоненты, которые практически идентичны по логике и поведению, но отличаются незначительными деталями, такими как стилизация, расположение или несущественные параметры. Со временем это приводит к накоплению избыточного кода, снижению читаемости проекта и усложнению его поддержки. Исправление ошибок и внесение изменений в такой код требует дополнительных усилий, а его масштабирование становится все более затруднительным [10].

Чтобы избежать подобных проблем, в разработке веб-интерфейсов применяются различные вспомогательные инструменты и подходы. Одним из наиболее эффективных решений является использование сторонних UI-библиотек, которые предоставляют готовые, хорошо структурированные и протестированные компоненты [3].

Такие библиотеки, как Material-UI, Ant Design и Bootstrap, позволяют

разработчикам стандартизировать интерфейсные элементы, обеспечивая их единообразный стиль и функциональность [2]. Кроме того, использование готовых решений значительно ускоряет процесс разработки, сокращая необходимость в создании однотипных компонентов с нуля.

Для определения наиболее удачного и эффективного решения для начала необходимо провести сравнительный анализ уже готовых решений, которые были описаны выше. Выбрав наилучший вариант можно будет определить качества, которые необходимо перенести в собственную библиотеку, а также учесть слабые стороны всех решений, чтобы создать наиболее конкурентны продукт. Сравнение с указанием критериев и положения у каждой из библиотеки будет указано в таблице 1.

Таблица 1 - Сравнение готовых решений

Критерий	Material-UI	Ant Design	Bootstrap
Функциональность	Предоставляет обширный набор компонентов, включая кнопки, карточки, модальные окна и многое другое.	Включает стандартные элементы, а также таблицы, формы и диаграммы.	Содержит компоненты для создания адаптивных веб-приложений, включая сетку, формы и навигационные панели.
Вес библиотеки	Средний	Тяжелее	Легкий
Поддержка TypeScript	Да	Да	Частично
Фреймворк	React	React	CSS + JS
Простота использования	Имеет хорошо структурированные примеры и простую интеграцию, но может потребоваться время для освоения из-за обширного функционала.	Предоставляет детальные руководства и множество примеров, но может потребоваться время для полного освоения.	Обладает знакомым многим разработчикам синтаксисом и легкостью интеграции, что делает его подходящим для быстрого прототипирования.

Продолжение таблицы 1

Критерий	Material-UI	Ant Design	Bootstrap
Документация	Предоставляет подробные примеры и руководства, что облегчает процесс разработки.	Имеет обширные руководства и примеры, что способствует быстрому освоению.	Предлагает обширные руководства и большое количество примеров для быстрого старта.
Поддержка и обновления	Активное сообщество разработчиков и регулярные обновления обеспечивают надежную поддержку.	Активное развитие и поддержка со стороны сообщества и разработчиков.	Обладает активным сообществом и регулярными обновлениями, что делает его надежным выбором.
Производительность	Оптимизирован для быстрой работы приложений, но может быть тяжелее по сравнению с другими библиотеками.	Обеспечивает высокую производительность благодаря оптимизированной структуре кода, но может быть тяжелее по сравнению с другими библиотеками.	Оптимизирован для быстрой работы, но может потребоваться дополнительная настройка для достижения максимальной производительности.

Для сравнительного анализа были выбраны следующие критерии: функциональность, вес библиотеки, поддержка TypeScript, фреймворк, простота использования, документация, поддержка и обновления, производительность. Проведенный анализ показал, что все рассматриваемые библиотеки обладают высокой популярностью среди разработчиков, предоставляют широкий функционал и хорошо оптимизированы, что предотвращает лишнюю нагрузку на систему. Они имеют активную поддержку как со стороны создателей, так и сообщества, а также сопровождаются качественной документацией [20].

Однако в процессе исследования были выявлены и недостатки:

- Библиотеки обладают значительным весом, что может негативно сказаться на производительности веб-приложений.
- Только две из трех библиотек поддерживают работу с React, что

ограничивает их универсальность.

– Освоение и применение методов, предлагаемых библиотеками, требует времени, что может усложнить их внедрение для новых пользователей.

Таким образом, при выборе UI-библиотеки важно учитывать ее размер, совместимость с фреймворком проекта и сложность изучения, чтобы найти наиболее подходящее решение для конкретных задач.

1.2 Определение требований к новой библиотеке компонентов

Для создания библиотеки UI-компонентов, которая минимизирует существующие проблемы готовых решений и обладает улучшенными характеристиками, необходимо определить основные требования к ее разработке. Эти требования помогут избежать недостатков популярных UI-библиотек, рассмотренных ранее, и создать удобное, эффективное и производительное решение [13].

Ключевым требованием является полная совместимость библиотеки с фреймворком React, поскольку он будет использоваться в качестве базовой платформы разработки. Благодаря этому требованию разработчики смогут легко интегрировать компоненты в свои проекты, не тратя дополнительное время на адаптацию или изменение архитектуры приложений. Поскольку сама библиотека создается с учетом React, ее совместимость обеспечивается автоматически [5].

Одним из главных факторов, влияющих на производительность веб-приложений, является размер подключаемых зависимостей. Поэтому библиотека должна быть компактной и включать только самые необходимые, базовые компоненты. С увеличением ее объема возрастает конечный размер проекта, что негативно сказывается на скорости сборки и производительности.

Ключевая идея заключается в использовании атомарного подхода: сложные элементы интерфейса можно создавать на основе базовых

компонентов, без необходимости включения громоздких решений [12]. Это позволит разработчикам адаптировать библиотеку под нужды конкретного проекта без лишнего кода и избыточных функций.

Для обеспечения стабильности и надежности кода библиотека должна разрабатываться с применением TypeScript.

Этот язык добавляет строгую типизацию в JavaScript, что особенно важно при создании масштабных проектов с участием больших команд.

Основные преимущества TypeScript:

- предотвращает ошибки приведения типов;
- позволяет устанавливать жесткие ограничения на входные параметры функций;
- исключает вероятность получения null-значений в неожиданных местах;
- улучшает читаемость и поддерживаемость кода.

Использование TypeScript существенно снижает вероятность возникновения критических ошибок на этапе разработки, что положительно сказывается на качестве продукта [11].

Одной из распространенных проблем готовых UI-библиотек является сложность их настройки и интеграции. Многие популярные решения обладают широким функционалом, но из-за этого их внедрение требует значительных усилий. В предлагаемой библиотеке упор делается на простоту использования и удобство конфигурирования [22]. Это обеспечит легкость встраивания компонентов в проекты, а также возможность кастомизации без необходимости глубокого изучения документации или сложных манипуляций с кодом.

Для предотвращения ошибок и обеспечения стабильности разработки необходимо внедрение интеграционных тестов. Эти тесты позволят проверять работу компонентов в изолированных условиях, гарантируя их корректное поведение при различных сценариях использования. Пример теста: если пользователь нажимает на кнопку со счетчиком, значение счетчика должно

увеличиваться на единицу [3].

Такая проверка позволяет убедиться, что компоненты работают ожидаемым образом, и любые изменения в коде не ломают существующий функционал.

Регулярное тестирование значительно сокращает время на отладку, предотвращает появление неожиданных багов и повышает качество разрабатываемого решения.

Эффективная разработка требует организации процесса работы, особенно в команде. Внедрение системы контроля версий (Git) обеспечит:

- отслеживание изменений на каждом этапе разработки,
- возможность совместной работы над проектом без конфликтов,
- легкость возврата к предыдущим версиям при необходимости,
- удобство тестирования новых гипотез без риска нарушить рабочий код.

Использование Git позволяет разработчикам разрабатывать функционал в отдельных ветках, тестировать изменения и интегрировать их в основной код только после успешной проверки. Это создает устойчивую и предсказуемую среду разработки, снижая вероятность ошибок и ускоряя процесс выпуска обновлений [23].

Выделенные требования формируют основу для разработки UI-библиотеки, которая будет легкой, гибкой, надежной и удобной в использовании. Реализация этих принципов обеспечит высокое качество компонентов, их простую интеграцию в проекты и улучшит процессы разработки веб-приложений. Для наглядного представления все требования и их краткое описание будут представлены в таблице 2.

Таблица 2 - Итоговые требования

Требование	Выбранное решение	Краткое описание
Фреймворк	React	Библиотека должна быть способна к интеграции в проекты на React
Размерность	Наименьшая	Конечная разработанная библиотека должна занимать как можно меньше памяти
Строгость типизации	Строгая(TypeScript)	Реализация должна быть создана с использованием строгой типизации для предотвращения потенциальных ошибок
Сложность реализации	Простейшая	Реализация должна быть выполнена с наименьшими усложнениями во избежания сложности понимания и интеграции библиотеки
Стабильность компонентов	Интеграционные тесты	Внедрение Unit тестов положительно сказывается на потенциале дальнейшего развития проекта, путем обеспечения проверок работоспособности старых методов и решений
Контроль версий	Git	Применение в разработке систем контроля версий, например, Git, позволит проще проверять гипотезы, работать в команде с разными направлениями и целями, а также возврат к старым версиям по необходимости

Определенные требования к разработке UI-библиотеки обеспечивают ее совместимость с React, компактность, гибкость и удобство интеграции. Использование атомарного подхода позволяет создавать сложные интерфейсы без лишнего кода, а применение TypeScript повышает надежность и читаемость кода. Интеграционные тесты гарантируют стабильность компонентов, а система контроля версий Git упрощает командную разработку. Соблюдение этих принципов создаст легкую, производительную и удобную в использовании библиотеку UI-компонентов, способствующую ускорению и повышению качества разработки веб-приложений.

2 Проектирование и оценка UI библиотеки

2.1 Разработка архитектуры UI библиотеки

Перед началом разработки UI-библиотеки необходимо продумать ее архитектуру, чтобы обеспечить структурированность кода, удобство поддержки и соответствие современным методологиям разработки. Определение архитектурных принципов и структуры проекта позволит создать удобное, гибкое и масштабируемое решение.

В основе архитектуры UI-библиотеки лежат три ключевых методологии, широко применяемые в профессиональной среде разработки: DRY (Don't Repeat Yourself), Atomic Design и BEM (Block-Element-Modifier) [8].

DRY – «Не повторяй себя». Этот принцип направлен на исключение дублирования кода [27]. В контексте UI-библиотеки он подразумевает:

- Вынос повторяющейся логики в отдельные переиспользуемые функции и компоненты.
- Разделение бизнес-логики и UI-компонентов.

Ниже представлены примеры, которые демонстрируют этот принцип. Сначала на рисунке 1 создаются два компонента, которые повторяют функционал друг друга, при этом различаются исключительно в названии, что создает один лишний элемент, которой не несет большой пользы для всего проекта, это уменьшает переиспользоваемость.

На рисунке 2 создается отдельный элемент, который выполняет заданную для него логику, и который уже в компоненте вызывается для реализации логики счета, которая может быть также вызвана в любом другом компоненте по необходимости.

```jsx
const Counter1 = () => {
  const [count, setCount] = React.useState(0);
  return <button
    onClick={() => setCount(count + 1)}>Count: {count}
  </button>;
};

const Counter2 = () => {
  const [count, setCount] = React.useState(0);
  return <button
    onClick={() => setCount(count + 1)}>Count: {count}
  </button>;
};
```

Рисунок 1 - Компоненты дубликаты

```jsx
const useCounter = (initialValue = 0) => {
  const [count, setCount] = useState(initialValue);
  const increment = () => setCount(count + 1);
  return { count, increment };
};

const CounterButton = () => {
  const { count, increment } = useCounter();
  return <button
    onClick={increment}>Count: {count}
  </button>;
};
```

Рисунок 2 - Выделенный элемент логики

Atomic Design – Разбиение интерфейса на атомарные элементы. Методология Atomic Design предлагает организовать компоненты в виде иерархической структуры:

- Атомы – базовые элементы (кнопки, иконки, текстовые поля).
- Молекулы – объединение нескольких атомов (например, поле ввода с кнопкой отправки).

– Организмы – сложные блоки, состоящие из молекул и атомов.
– Шаблоны – каркас страницы, состоящий из различных организмов.
– Страницы – конкретная реализация шаблонов с переданными данными.

Преимущества метода:

– Улучшает читаемость кода.
– Облегчает тестирование и отладку.
– Повышает гибкость компонентов.

Пример представлен на рисунке 3, где сначала создается базовый компонент Button, который реализует общую функциональность и принимает параметры. Затем этот компонент используется внутри других, более сложных компонентов, без необходимости дублирования кода.

```
const Button = ({ onClick, children }) => (
  <button
    onClick={onClick}
    className="btn">
    {children}
  </button>
);

const SearchBar = () => (
  <div>
    <input type="text" />
    <Button>Поиск</Button>
  </div>
);
```

Рисунок 3 - Атомный дизайн

BEM – Организация CSS-классов. BEM (Block-Element-Modifier) используется для структурированной стилизации компонентов. Все элементы абстрактно представляются как блоки и элементы, которым могут присваиваться также модификаторы, таким образом создается иерархия в

HTML древе, что повышает читаемость и интуитивность кода.

Основные принципы:

– Блок (Block) – независимый компонент (button).
– Элемент (Element) – составная часть блока (button__icon).
– Модификатор (Modifier) – изменяет внешний вид блока или элемента (button_primary).

Преимущества:

– Улучшает читаемость кода.
– Делает CSS более предсказуемым.
– Облегчает масштабирование и поддержку проекта.

В примере ниже представлена реализация этого метода. На рисунке 4 показан компонент, который включает в себя названия классов, созданных с помощью BEM.

```
const Button = ({ type = "primary", icon, children }) => {
  return (
    <button className={`button button_${type}`}>
      {icon && <span className="button__icon">{icon}</span>}
      {children}
    </button>
  );
};
```

Рисунок 4 - Компонент с BEM

На рисунке 5 показан SASS файл, который включает в себя иерархию классов по BEM для стилизации вышеописанного класса.

Рисунок 5 - SASS файл по BEM

После определения методологий, необходимо определить саму структуру проекта, того, как необходимо составить в нем файловую систему для удобного понимания и хорошей организации [17]. Для этого была выбрана одна из стандартных структур, которая применяется при frontend-разработке - в корневом каталоге будет располагаться две папки: src и dist, где первая хранит папку с компонентами, с которыми производится работа до сборки, а также другие папки, в которых могут храниться вспомогательные функции и утилиты, во второй папке хранятся уже готовые, собранные элементы, которые возможно посмотреть уже в браузере, либо применить в другом месте. Папка компонента заключает в себе 4 файла, один отвечает за логику и внешний вид компонента с расширением. tsx, второй хранит в себе стили sass, в третьем прописываются интеграционные тесты, а последний используется для экспорта компонента в другие компоненты. Также в корне располагаются конфигурационные файлы для всего проекта, для сборки библиотеки и настройки TypeScript.

На рисунке 6 показан пример того, как будет организована файловая структура всего проекта.

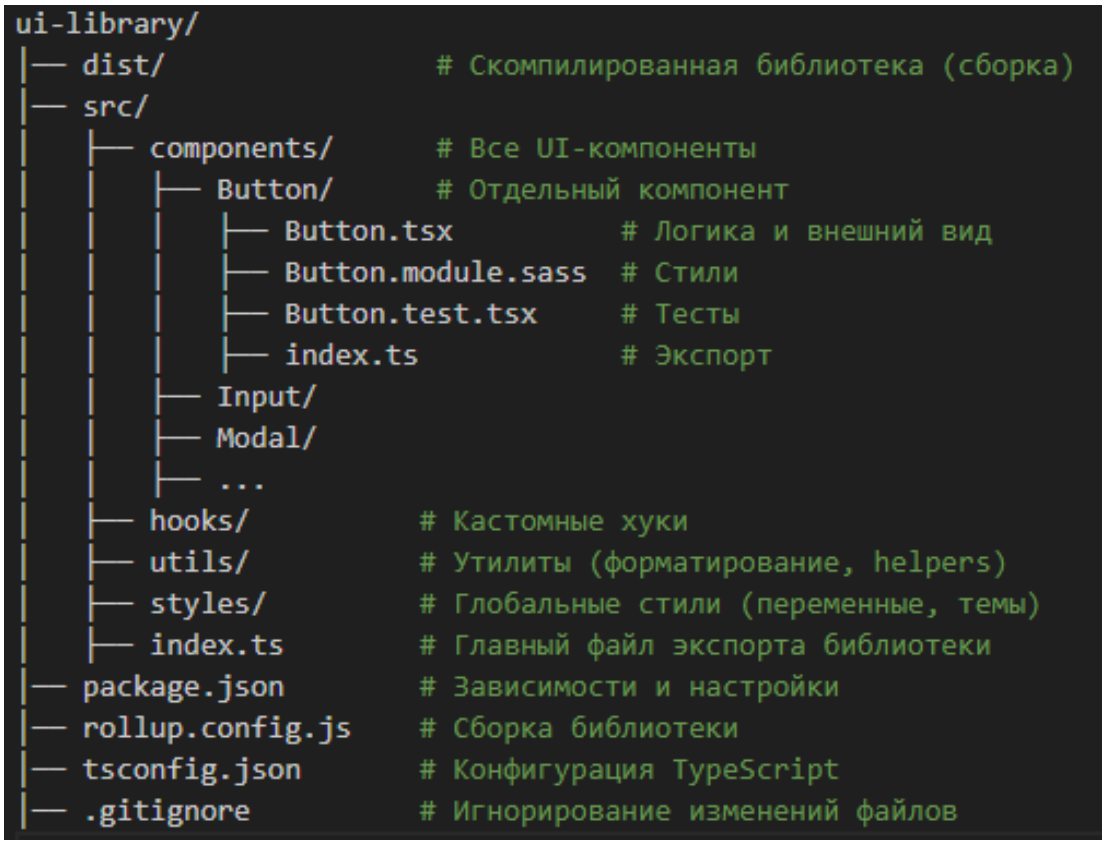


Рисунок 6 - Файловая структура

В React, как и во многих других фреймворках существует несколько способов реализации компонентов, каждый из которых подходит для разных ситуаций, а некоторые из них уже считаются устаревшими, но все также находят применение.

Одним из самых основных и понятных является классовый, где компоненты представляются в виде отдельных классов, которые имеют свои собственные переменные и функции. Основным преимуществом, как уже было сказано выше, является его хорошая интерпретируемость, так как любой человек, который знаком и работал с принципами ООП, поймет, что и как работает. Однако главный минус, что он является устаревшим и многий функционал, который уже считается базовым в React, невозможен к

реализации в подобных элементах. Но он все равно все еще встречается в силу своей понятности. Ниже на рисунке 7 представлена реализация классового компонента Button [18].

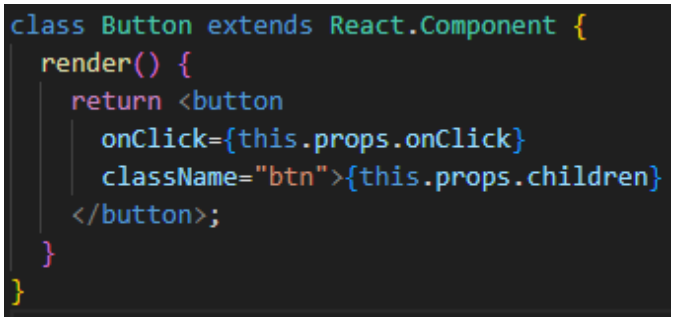


Рисунок 7 - Классовая реализация компонента

На данный момент самым распространенными и поддерживаемыми являются функциональные компоненты. Они представляют из себя стрелочную функцию, которые принимают в себя переменные в виде объекта - пропсы, которые уже дальше, внутри тела самой функции, способны приниматься, обрабатываться и передаваться дальше при необходимости. Главное преимущество заключается в том, что это является флагманским подходом, вследствие чего, весь новый функционал разрабатывается именно на них, а также весь уже имеющийся основной также завязывается на работе с ними, который, например, в классовых компонентах, невозможен, либо реализуется более сложными и обходными путями. Также, они по сравнению вышеупомянутыми занимают значительно меньше строк и места, что повышает его читаемость и экономит рабочее пространство [8]. Ниже представлен функциональный компонент Button на рисунке 8.

Также выделяются компоненты высшего порядка, которые встречаются реже всего, но моментами необходимы, для реализации определенного функционала. Основная идея этого подхода заключается в том, что такой компонент представляет из себя обертку поверх другого уже созданного компонента, который расширяет его возможности [8].

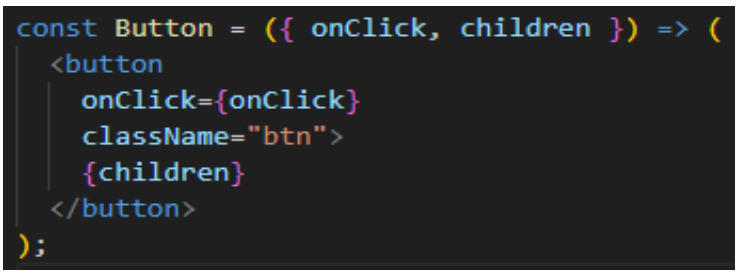


Рисунок 8 - Функциональная реализация компонента

При определенных условиях он оправдан, так без могут плодиться бессмысленные компоненты, которые будут очень сильно повторять друг друга, что негативно влияет на общий опыт разработки. Однако основным минусом является его довольно сложное чтение, понимание и его дальнейшая модификация, что усложняет его отладку и дальнейшее применение, посему встречается довольно редко, по сравнению с функциональными компонентами. Ниже представлен компонент на рисунке 9, который расширяет функционал уже ранее созданного.

Рисунок 9 - Компонент высшего порядка

Для подведения итогов была описана таблица 3 с критериями для каждого подхода, из которой следует, что функциональный подход создания

компонентов является преимущественным.

Таблица 3 - Способы реализации компонентов

Критерий	Функциональные компоненты	Классовые компоненты	HOC (Higher Order Components)
Синтаксис	Простая функция	Класс, расширяющий React.Component	Функция, принимающая другую в себя
Жизненный цикл	useEffect	Методы состояний	Использует жизненный цикл оборачиваемого компонента
Переиспользуемость	Комбинирование хуков	Ограниченная	Высокая (инкапсулирует логику)
Читаемость	Высокая	Низкая (больше кода)	Сложная отладка

Продуманная архитектура UI-библиотеки обеспечивает ее структурированность, удобство поддержки и соответствие современным методологиям. Использование принципов DRY, Atomic Design и BEM позволяет минимизировать дублирование кода, улучшить модульность компонентов и упростить стилизацию. Выбранная файловая структура проекта способствует удобной организации кода и масштабируемости. Анализ различных подходов к созданию компонентов показал, что функциональные компоненты являются наиболее предпочтительными благодаря лаконичности, поддержке современного функционала и лучшей читаемости. Эти решения формируют надежную основу для эффективной разработки UI-библиотеки.

2.2 Разработка математической модели для оценки компонентов

После разработки прототипа решения необходимо проверить, провести исследование того, насколько проект получается оптимизированным, качественным и эффективным, для этого необходимо разработать математическую модель, которая была бы способна продемонстрировать вышеописанные качества с помощью метрики. Также важным моментом

является необходимость проверки того, насколько хорошо решение справляется со своей первоначальной задачей, улучшением и ускорением разработки после своего внедрения, из-за чего необходимо продумать дополнительную модель, которая бы отвечала за это.

Перед составлением первой модели необходимо выделить ключевые свойства работы самой библиотеки и ее компонентов по отдельности, которые бы можно было применять в исследовании. Если обратиться к требованиям, которые были поставлены в разделах выше, то можно выделить такие свойства как размер библиотеки, выраженный в МБ, процент покрытия кода интеграционными тестами и количество выполненных методологий. Помимо этого также можно добавить сторонние параметры, например, покрытие линтером, который демонстрирует то, насколько код является “чистым” и стабильным с точки зрения строгости типизации, количество критических ошибок, которые в будущем могут привести к тому, что проект срочно прекратит свою работу, и обычных замечаний, которые не несут большой потенциальной угрозы устойчивости работы всего проекта, но могут привести к путанице в будущем.

Библиотека подразумевает создание такого решения, которое бы сократило количество используемых строк для достижения необходимого результата, из чего можно выделить свойства, которые бы определяли эффективность внедрения этого решения: количество строк JSX разметки, количество строк CSS(SASS) стилей, время сборки всего проекта. Для более равномерного представления все эти свойства следует рассматривать в распределении на каждый компонент, чтобы радикальные улучшения в одном единственном месте не отклоняли оценку слишком сильно.

Для наглядности все вышеописанные показатели будут вынесены в отдельные для каждой из математических моделей таблицы 4 и 5 с приведенными для них единицами измерения, а также условными обозначениями в виде переменных, которые будут участвовать в итоговых математических моделях.

Таблица 4 - Критерии оценки первой модели

Критерий	Единицы измерения	Условное обозначение
Процент покрытия интеграционными тестами	%	IT
Количество выполненных методологий	Единицы	Mn
Размер библиотеки	Килобайты	W
Количество критических ошибок	Единицы	LE
Количество замечаний	Единицы	LC

Таблица 5 - Критерии оценки второй модели

Критерий	Единицы измерения	Условное обозначение
Количество строк JSX разметки	Единицы	J
Количество строк CSS(SASS) стилей	Единицы	S
Разница во времени сборки тестового проекта с библиотекой	Миллисекунда	ΔT
Количество компонентов	Единицы	C

Каждое исследуемое свойство будет вносить свой вклад в общую оценку эффективности. В связи с этим вводятся частные оценки: *Q1, Q2, Q3, Q4, Q5*, для первой модели и оценки: *Ej, Es, Et,* для второй модели каждая из которых отражает влияние конкретного параметра.

Поскольку при определённых обстоятельствах и условиях работы может возникать такое, что какое-то из установленных свойств будет более или наоборот менее важным по сравнению с другими необходимо предусмотреть возможность регулирования значимости каждого из них с помощью дополнительно определяемых коэффициентов(весов): w_1, w_2, w_3, w_4, w_5 и w_j, w_s, w_t.

Для более универсального и удобного представления конечная оценка моделей должна принимать значения в диапазоне от 0 до 1, поэтому оценка каждого из параметров также должна принимать такие же значения с последующим её умножением на соответствующий весовой коэффициент.

Если рассматривать каждую формулу для первой модели оценки

качества библиотеки каждого из вышеописанных свойства более подробно, то это все можно сформировать в список ниже:

Q1 (Покрытие тестами):

- Это показатель, который отвечает за процент покрытия кода интеграционными тестами. Он измеряется в процентах, поэтому для приведения значения к нормированной форме, его нужно разделить на 100. Таким образом, этот показатель будет лежать в пределах от 0 до 1.
- После нормализации значение умножается на вес, который отражает важность данного показателя в общем контексте оценки проекта. Вес позволяет скорректировать влияние этого показателя на итоговую оценку в зависимости от приоритетов проекта.
- Формула оценка для покрытия тестами (1).

$$Q_1 = w_1 * \frac{IT}{100} \quad (1)$$

где

w_1 - вес критерия показателя покрытия тестами;

IT - процент покрытия кода интеграционными тестами.

Q2 (Соблюдение методологий):

- Это свойство измеряет степень соблюдения введенных методологий и стандартов разработки. Обозначение Mnm – это суммарное количество методологий, которые могут быть изменены в зависимости от изменений в проекте. В данном контексте Mnm равно 3, что означает, что для оценки этого показателя учитываются три ключевых методологии, которым должны следовать разработчики.
- Влияние данного показателя на итоговый результат зависит от того, насколько хорошо соблюдены эти методологии.
- Формула оценка для соблюдения методологий (2).

$$Q_2 = w_2 * \frac{Mn}{Mn_m}; \quad (2)$$

где

w_2 - вес критерия соблюдения методологий;

Mn - количество выполненных методологий;

Mn_m - общее количество методологий в проекте.

Q3 (Вес библиотеки):

– Этот показатель отвечает за размер конечной библиотеки. Он учитывает оптимальность и экономию ресурсов в коде. Для вычисления используется максимальное наилучшее значение Wm, которое задает идеальный размер библиотеки.
– Если размер библиотеки меньше или равен этому значению, то показатель равен 1, что означает отличную эффективность. Если размер библиотеки превышает Wm, то эффективность будет снижаться по экспоненциальной функции, стремясь к нулю. Это отражает важность минимизации размера библиотеки для улучшения производительности.
– Формула оценки веса библиотеки (3).

$$Q_3 = 1, \text{при } W <= W_m;\ w_3 * e^{-(W-W_m)*0.02}, \text{при } W > W_m; \quad (3)$$

где

w_3 - вес критерия размера библиотеки;

W - итоговый размер библиотеки;

W_m - максимальное допустимое идеальное значение веса.

Q4 (Критические ошибки):

– Этот показатель связан с количеством критических ошибок, которые могут быть найдены линтером в процессе сборки. Критические ошибки имеют очень серьезные последствия, поэтому функция

убывает очень резко.

– С увеличением количества критических ошибок эффект на итоговую оценку будет стремительно падать, что отражает важность минимизации таких ошибок.
– Формула оценки количества критических ошибок (4).

$$Q_4 = w_4 * e^{-0.6*LE}; \quad (4)$$

где

w_4 - вес критерия количества критических ошибок;

LE - количество критических ошибок.

Q5 (Замечания линтера):

– Это количество замечаний, которые линтер может выдать в процессе сборки. Хотя замечания также важны, их влияние на проект не такое критическое, как у ошибок.
– Функция убывания для этого показателя будет происходить медленнее, чем для критических ошибок, что дает возможность учитывать их, но с меньшим влиянием на итоговую оценку.
– Формула оценки замечаний (5).

$$Q_5 = w_5 * e^{-0.1*LC}; \quad (5)$$

где

w_5 - вес критерия количества замечаний;

LC - количество замечаний.

Результирующее значение *Q* получается путем перемножения значений эффективности каждого из свойств (*Q1*, *Q2*, *Q3*, *Q4*, *Q5*). Каждый из этих показателей оценивает определенную характеристику проекта. Поскольку показатели нормированы в интервал от 0 до 1, их перемножение дает итоговый показатель, который также будет находиться в этом диапазоне.

Формула 6 демонстрирует конечный вид математической модели для всей библиотеки.

$$Q = \frac{Q_1 + Q_2 + Q_3 + Q_4 + Q_5}{w_1 + w_2 + w_3 + w_4 + w_5}; \quad (6)$$

Теперь необходимо рассмотреть свойства для оценки эффективности внедрения самой библиотеки в проект, что будет представлено в списке:

E_j (Количество JSX строк):

- Это показатель, демонстрирует то, насколько было сокращено использование строк JSX разметки в проекте после внедрения библиотеки.
- В расчётах оценки участвуют 2 значения, количество строк до внедрения и после, ожидается, что после внедрения количество будет меньше либо равно, в противном случае параметр может негативно сказаться на качестве оценки.
- Формула оценка для JSX строк (7).

$$E_j = w_j \frac{J_1}{J_0}; \quad (7)$$

где

w_j - вес критерия;

J_1 - количество JSX строк после внедрения;

J_0 - количество JSX строк до внедрения.

E_s (Количество CSS(SASS) строк):

- Это показатель, демонстрирует то, насколько было сокращено использование строк CSS(SASS) разметки стилей в проекте после внедрения библиотеки.
- В расчётах оценки участвуют 2 значения, количество строк до внедрения и после, ожидается, что после внедрения количество будет

меньше либо равно, в противном случае параметр может негативно сказаться на качестве оценки.

– Формула оценка для JSX строк (8).

$$E_s = w_s \frac{S_1}{S_0}; \quad (8)$$

где

w_s - вес критерия;

S_1 - количество CSS(SASS) строк после внедрения;

S_0 - количество CSS(SASS) строк до внедрения.

E_t (Время сборки проекта):

– Это показатель, демонстрирует то, насколько было изменено время сборки проекта.

– Формула оценка для времени сборки (9).

$$E_t = w_t \frac{T_1}{T_0}; \quad (9)$$

где

w_t - вес критерия;

T_1 - количество строк после внедрения;

T_0 - количество строк до внедрения.

Итоговая формула для оценки того, насколько эффективно внедрение разработанной библиотеки в проект представлена под номером 10.

$$E = 1 - \frac{C_1}{C_0} * \frac{(E_j + E_s + E_t)}{w_j + w_s + w_t}; \quad (10)$$

В ней учитывается влияние каждого из вышеописанных свойств, путём суммирования их значений. Поскольку веса задаются в определённом диапазоне сумму необходимо нормализовать с помощью её деления на сумму

значений весов, что позволит получить значение в диапазоне от 0 до 1. Также было важно учесть влияние количества компонентов, что отразилось в умножение полученного значения после нормализации на частное от деления количества компонентов после внедрения на количество компонентов до внедрения, что позволяет получить более гладкую картину представления ситуации. Результат всех этих операций вычитается из единицы, что даёт значение, показывающие эффективность внедрения, где 0 отсутствие каких-либо положительных изменений, а 1 сильное ускорение разработки.

Подводя итоги, были выбраны критерии, которые бы участвовали в анализе посредством построенных математических моделей для оценки качества библиотеки, а также оценки того, как интеграция этой библиотеки сказывается на разработке. Были описаны формулы для этих моделей с детальными описаниями каждого из параметра участвующих в них, как они высчитываются и что означают. Конечные значения каждой модели принимают значения от 0 до 1, что улучшает понимание и интерпретацию конечного результата, где 0 наихудший результат, а 1 наилучший. Большим плюсом этих моделей является избирательное определение значимости каждого критерия, которые участвуют в анализе, что задаётся с помощью весом, каждый из которых соответствует заданному ему критерию.

3 Разработка и тестирование библиотеки компонентов

3.1 Разработка алгоритма для оптимизации компонентов

По файловой структуре, которая была описана ранее, была создана основа проекта, которая показана на рисунке 10.

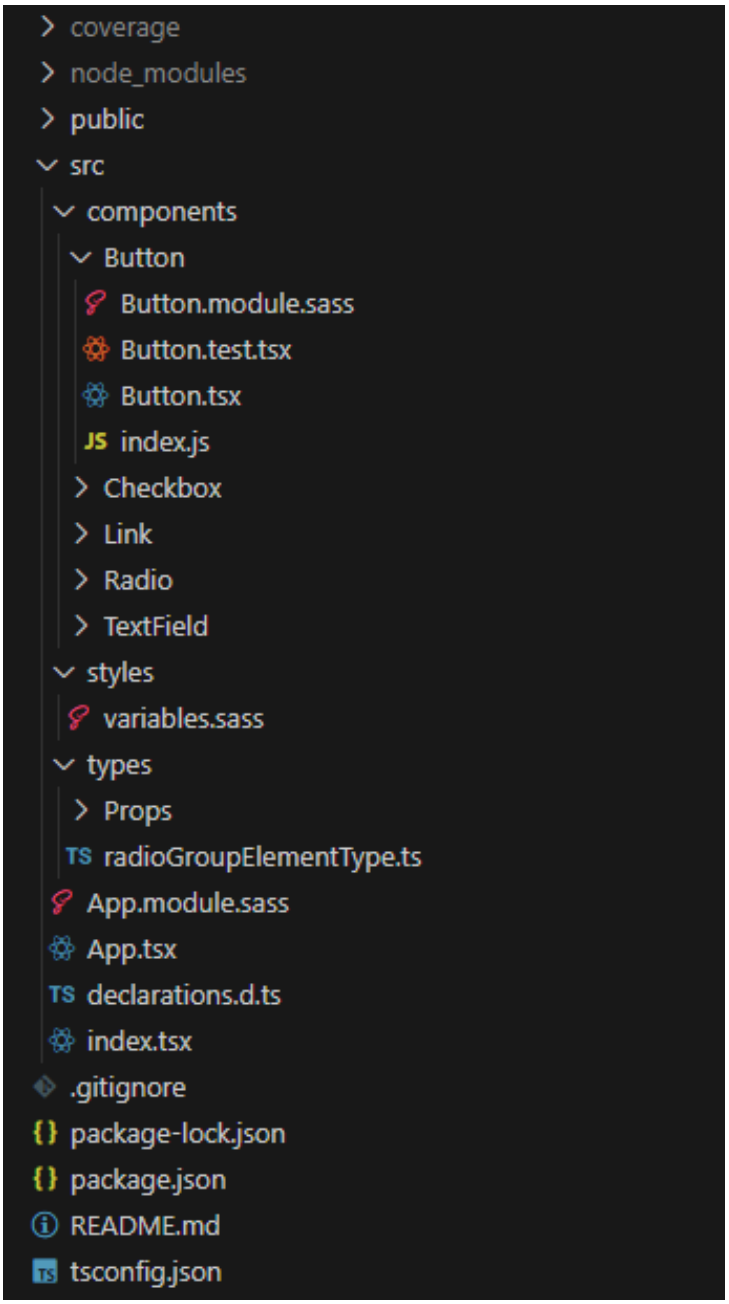


Рисунок 10 - Структура проекта

Всю структуру проекта можно разбить на 4 части: конфигурационные файлы, публичные файлы, рабочие элементы и вспомогательные элементы. Первый тип файлов ответственен за настройку всего проекта: указание информации о проекте, указание рабочих команд, рабочих и релизных зависимостей, конфигурационной информации для других библиотек, используемых в работе, примером таких файлов в проекте являются: package.json, package-lock.json, tsconfig.json, .gitignore, declarations.d.ts [8]. Рассмотрим описанное подробнее.

package.json – это основной конфигурационный файл любого Node.js-проекта, содержащий базовую информацию о проекте (название, версия, описание), список зависимостей и dev-зависимостей, а также скрипты для запуска и сборки проекта. Представлен на рисунке 11.

package-lock.json – автоматически генерируемый файл, фиксирующий точные версии всех установленных зависимостей и их зависимостей. Это обеспечивает воспроизводимость окружения при установке зависимостей на разных машинах или в разных CI/CD процессах.

tsconfig.json – файл конфигурации TypeScript, определяющий, как компилятор должен обрабатывать исходный код. Содержит информацию о путях, строгих проверках, таргете компиляции и других параметрах, влияющих на качество и структуру выходного JavaScript-кода. Указан на рисунке 12.

.gitignore – файл, определяющий, какие файлы и папки должны быть проигнорированы системой контроля версий Git. Обычно в него включаются временные файлы, артефакты сборки и конфиденциальные данные, такие как .env или node_modules. Показан на рисунке 13.

declarations.d.ts – файл с глобальными объявлениями типов, используемый для добавления или расширения типов в TypeScript, особенно в случае, когда сторонние библиотеки не предоставляют собственные типы или требуется создать кастомные глобальные интерфейсы.

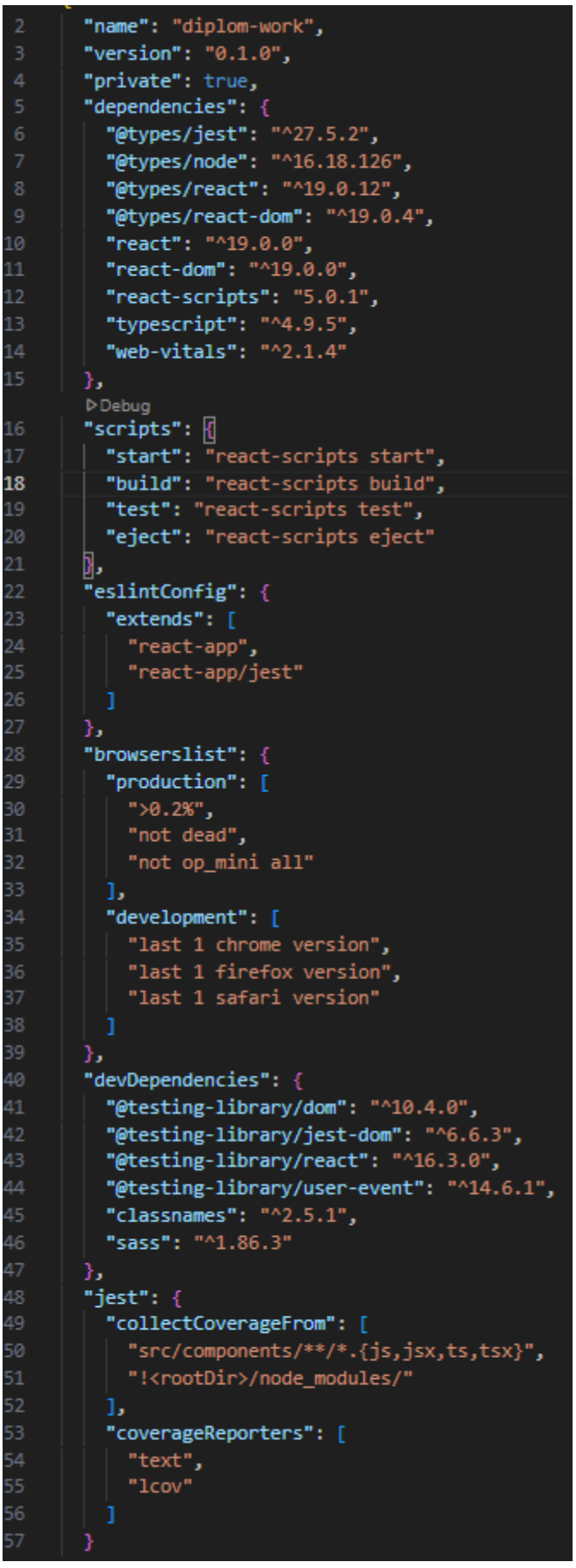


Рисунок 11 - Файл package.json

Рисунок 12 - Файл tsconfig.json

Рисунок 13 - Файл .gitignore

Ко второму типу относятся файлы, которые находятся в папке public, содержимое представлено на рисунке 14. Их особенность и отличие от других

заключается в том, что они никак не преобразуются сборщиками (Webpack, Vite и другие), которые всегда присутствуют в таких проектах, а используются и вставляются в корень итоговой сборки как есть в файл index.html.

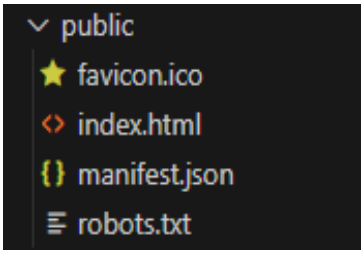


Рисунок 14 - Содержимое папки public

В этом случае туда были помещены:

– index.html – основной HTML-файл, в который React монтирует всё приложение.
– favicon.ico - иконка, используемая браузером (иконка вкладки, манифест иконки и т.п.).
– manifest.json – описание PWA (Progressive Web App), содержит метаинформацию о приложении: название, иконки, цвета темы и т.д.
– robots.txt – инструкция для поисковых роботов, какие страницы индексировать, а какие – нет.

К третьему типу можно отнести те папки и файлы, которые являются главным составляющим всего проекта: папки компонентов и их содержимое, index.tsx, App.tsx, App.module.sass.

Подробнее стоит рассмотреть файлы, которые составляют структуру каждого компонента, для примера будет выбран компонент Button. Он, как и каждый из других компонентов проекта, состоит за 4 файлов: *.tsx, *.module.sass, *.test.tsx и index.js [3].

Главным файлом, где описываются вся логика, работа с пропсами, работа с переменными, классами и состоянием в данном случае является Button.tsx, полное описание которого показано на рисунке 15.

```tsx
import classNames from "classnames";

import ButtonUIProps from "../../types/Props/ButtonUIProps";

import styles from "./Button.module.sass";


export default function ButtonUI({
  className,
  disabled = false,
  outlined = false,
  type = "button",
  size = "medium",
  variant = "primary",
  children = "Button",
  clickEvent,
}: ButtonUIProps) {
  const classes = classNames(
    className,
    styles.button,
    styles[`button_${size}`],
    styles[`button_${variant}`],
    {
      [styles["button_outlined"]]: outlined,
    }
  );

  return (
    <button className={classes} type={type} disabled={disabled} onClick={clickEvent}>
      {children}
    </button>
  );
};
```

Рисунок 15 - Файл Button.tsx

Обычно они состоят из раздела с импортами библиотеки, других компонентов или же иных элементов, которые могут применятся для логики или же разметки. Из-за того, что в разработке использовался функциональный подход, то все компоненты представляются как функции, которые впоследствии экспортируются из файла и переиспользуются в других необходимых местах, например, других компонентах, либо же главном файле. В этом случае экспортируется функция с названием ButtonUI, которая принимает в себя лишь одну переменную - объект с типом ButtonUIProps. По причине использования в разработке TypeScript и необходимости указывать жёсткие ожидаемые типы аргументов, поэтому необходимо было создать отдельный объект type, который бы указывал сколько элементов ожидается в объекте-аргументе и какого типа должны быть эти элементы, поэтому был создан вышеописанный тип ButtonUIProps, который показан на рисунке 16.

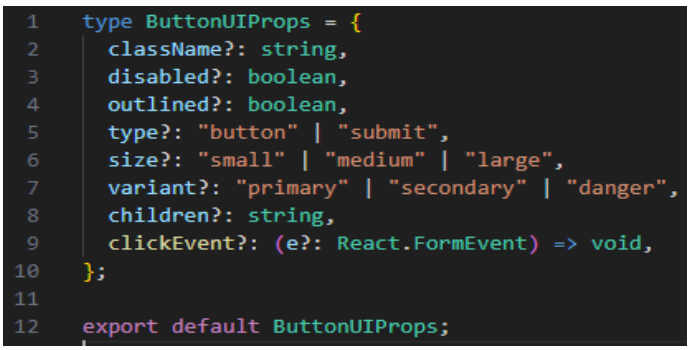


Рисунок 16 - Тип ButtonUIProps

Он сообщает о том, что объекты с этим типом данных включают в себя только 7 элементов: className, disabled, outlined, type, size, variant, children, clickEvent, с заранее указанными типами. Особое внимание стоит уделить элементам type, size, variant, который принимают строковые значения, но только в нескольких вариациях, которые указываются через прямой слэш, а также элементу clickEvent, который является функцией. Все из описанных пропсов не являются обязательными для работы компонента, его можно будет вызывать без передачи значений для указанных полей, так как одним указывается значение по умолчанию, а другие являются опциональными, и работа без них не будет нарушена.

Тело компонента в данном случае состоит из двух частей: константы, которая определяет классы в зависимости от переданных аргументов, а также возвращаемого значение, которое представляет из себя JSX разметку, подхода, который позволяет совмещать HTML и JavaScript, чтобы было возможно манипулировать первым в зависимости от переданных аргументов в компонент с помощью методов второго.

Константа определяется с помощью вспомогательной библиотеки classNames, которая позволяет объединить несколько значений в одну строку статичных названий классов, классов, которые составляются из переменных, а также добавлять классы в зависимости от значений boolean переменных, где, если значение true, то строка добавится, в противном случае нет. Что позволяет

динамически манипулировать классами, что в свою очередь даёт возможность динамического изменения внешних составляющих каждого отдельного компонента на основании того, какие аргументы были переданы и какие значения эти аргументы принимают, а также какое значение принимают внутренние переменные компонента, так называемое - состояние(state).

Стоит сказать о том, что в проекте используется подход style-component, который позволяет создавать уникальные идентификаторы классов, чтобы избежать коллизий, а также он позволяет оперировать с классами CSS, как с элементами объекта styles, который представляет собой импорт файла классов SASS, например, styles.button, что означает, что необходимо указать класс button из импортированного файла стилей, в данном компоненте используется Button.modules.sass, фрагмент которого показан на рисунке 17.

```
@use "../../styles/variables.sass" as var

.button
  color: #fff
  border-radius: 4px
  cursor: pointer
  transition-duration: 0.1s

  &:disabled
    background: var.$disabled-color
    border: 1px solid var.$disabled-color;
    cursor: default
    &:hover
      box-shadow: none

  &:disabled.button_outlined
    background: none;
    color: var.$disabled-color;

  &:hover
    box-shadow: 3px 3px 5px #a3a0a0

  &_primary
    background: var.$primary-color
    border: 1px solid var.$primary-color;
```

Рисунок 17 - Файл Button.modules.sass

Как уже было сказано ранее, функция возвращает JSX разметку, которая определяется переменными и константой classes. Таким образом возвращается

элемент button с указанными для него классами, взятыми из константы, указанными атрибутами типа кнопки, в данном случае, это обычная кнопка с типом «button», состоянием «отключённости» кнопки, возможно ли с ней взаимодействовать с помощью нажатия или нет, а также переданным из аргументов “слушателем”, который реагирует на нажатия на кнопку.

Готовые компоненты экспортируются из файла, которые позже можно получить в любом другом месте с помощью импорта с указанием пути расположения компонента в проекте, однако надо указывать путь конкретно до .tsx файла из-за чего путь получается длиннее, поэтому создаётся отдельный файл index.js, который служит точкой экспорта всего компонента, что позволяет сократить путь для его импорта, например «import Button from './Button/Button'» становится “import Button from './Button'”. Содержимое файла представлено на рисунке 18.

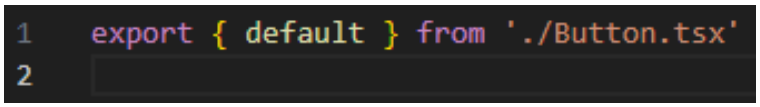


Рисунок 18 - Файл index.js

После того как были описаны компоненты, их типы, логика и стилизация, необходимо проверить корректную работоспособность их функционала. Для этого отдельно были созданы интеграционные тесты(unit - тесты), которые рендерят каждый взятый компонент в изолированной среде и создают для них разные ситуации, чтобы проверить каков будет результат их работы, что поменяется, после чего это будет сравниваться со значениями, которые ожидаются, если значения совпадают, значит всё работает корректно. Реализация тестов для компонента Button представлена в файле Button.test.tsx и показана на рисунке 19.

Структура файла состоит из импортов, после которых идут блоки самих тестов. Для удобства восприятия они объединяются в логические блоки с

помощью describe, благодаря чему в отчёте тесты объединяться в одну группу, в данном случае они все будут собраны в «Button component».

```
import '@testing-library/jest-dom'
import { render, screen } from "@testing-library/react";
import ButtonUI from "./Button";

describe("Button component", () => {
  test("Button is rendered", () => {
    render(<ButtonUI></ButtonUI>);
    expect(screen.getByText("Button")).toBeInTheDocument();
  });

  test("Button change text", () => {
    render(<ButtonUI>Click</ButtonUI>);
    expect(screen.getByText("Click")).toBeInTheDocument();
  });

  test("Button change size", () => {
    render(<ButtonUI size="large">Click</ButtonUI>);
    expect(screen.getByText("Click")).toHaveClass("button_large");
  });

  test("Button change variant", () => {
    render(<ButtonUI variant="secondary">Click</ButtonUI>);
    expect(screen.getByText("Click")).toHaveClass("button_secondary");
  });

  test("Button set outlined", () => {
    render(<ButtonUI outlined={true}>Click</ButtonUI>);
    expect(screen.getByText("Click")).toHaveClass("button_outlined");
  });

  test("Button is disabled", () => {
    render(<ButtonUI isDisabled={true}>Click</ButtonUI>);
    expect(screen.getByText("Click")).toHaveAttribute("disabled");
  });
});
```

Рисунок 19 - Интеграционные тесты компонента Button

Каждый тест отдельно для себя рендерит компонент ButtonUI, после чего выполняет поиск по заданным критериям элементы на виртуальном «мониторе». Например, в первом тесте создаётся компонент без передачи каких-либо аргументов, так как при отсутствии их указываются значения по умолчанию, то ожидается, что текст в этом блоке будет равен стандартному «Button», что и проверяется. На «мониторе» ищутся элементы с текстом «Button», и ожидается, что они должны присутствовать в документе, если всё будет выполнено, то тест считается пройденным. Подобным образом

проводятся все оставшиеся, за исключением ожиданий, например, в тесте изменения размера ожидается, что элемент с указанным тестом будет иметь класс, который должен соответствовать указанному значению размера, или же будет ожидаться, что элемент будет иметь атрибут, отвечающий за отключённое состояние кнопки.

К четвёртому типу можно отнести файлы, которые являются вспомогательными элементами для основных, которые выносят общую логику, переменные, стили или дополнительные типы. Некоторые из таких файлов уже были рассмотрены выше, например, ButtonUIProps, который создавал отдельный тип для пропсов компонента ButtonUI. Такие отдельные типы существуют для каждого самостоятельного компонента, потому что они все оперируют своими входными аргументами. Папка с отдельными для каждого из компонента типами представлена на рисунке 20.

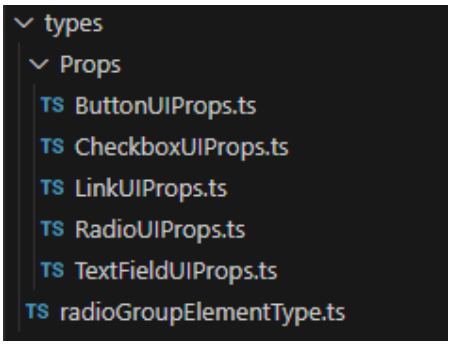


Рисунок 20 - Папка с типами

Помимо этого, в папке располагается дополнительный тип - radioGroupElementType, который используется в другом RadioUIProps. Так как последний включает в себя массив объектов, который также необходимо описать и типизировать, было необходимо добавить сложный тип, который бы описывал один элемент этого массива, после чего можно было бы описать этот элемент, как массив этих сложных типов данных. Такая зависимость показана

на рисунке 21, где показан сам сложный тип, а также на рисунке 22, где показано применения этого типа.

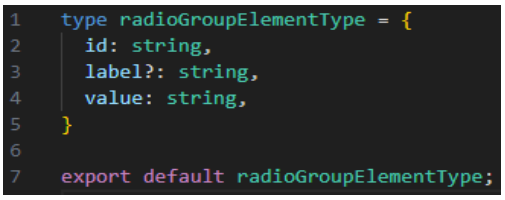


Рисунок 21 - Сложный тип

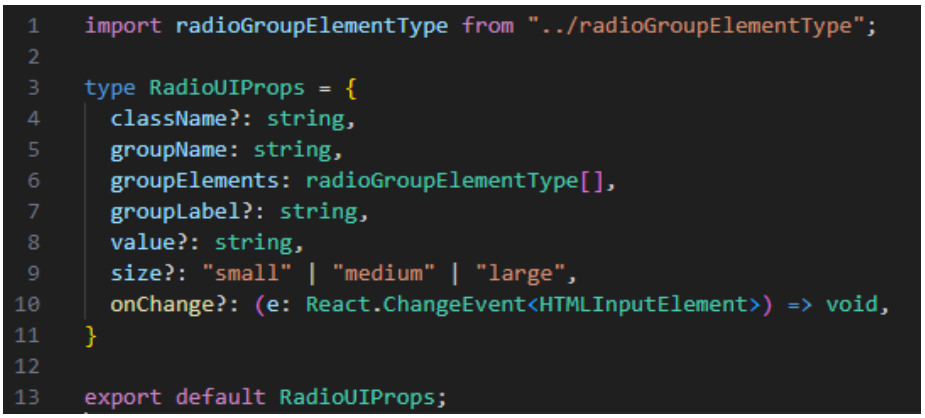


Рисунок 22 - Применение сложного типа

Кроме того, к этому типу можно отнести файл variables.sass, который включает в себя переменные, которые используется внутри других sass файлов, таким образом одинаковые цвета и размеры шрифтов возможно менять из одного места. Его реализация показана на рисунке 23

Последним этапом разработки являлось фиксирование внесённых изменений, решений, наработок и настроек, что выражалось в применении системы контроля версии Git.

Если быть подробнее, то на каждом этапе: инициализация, настройка проекта, разработка и подготовка конечного представления, создавался коммит, который включает в себя историю изменений (удаление, добавление, редактирование) файлов, которые относятся к теме, с добавление к нему

комментария, который объяснял бы краткую информацию, что именно было сделано в этом коммите, каким изменения были внесены.

```
$primary-color: #52aeff
$secondary-color: #1e4dff
$danger-color: #ff3f3f
$disabled-color: #afafaf

$small-fontSize: 12px
$middle-fontSize: 14px
$large-fontSize: 16px
```

Рисунок 23 - Файл variables.sass

Полную информацию с историей коммитов можно получить с помощью команды git log в консоли, после чего будет выведена вся их история с дополнительной информацией: автором коммита, временем и датой, когда он был совершён, кратким описание, а также уникальным хешем. Пример этого можно увидеть на рисунке 24.

```
commit 6f3aa173ad317d6720ef81f5fb1470f99f4b2d05 (HEAD -> main)
Author: Ivan <yanickiy.ivan@mail.ru>
Date:   Thu Apr 24 00:29:58 2025 +0400

    Change state logic

commit 93c12c33c1400c246974d15248685f075faffd87
Author: Ivan <yanickiy.ivan@mail.ru>
Date:   Sat Apr 12 19:07:07 2025 +0400

    Demonstration of functionality

commit 0c6bad209ef325f674aa94b1b1fdd9d35be23526
Author: Ivan <yanickiy.ivan@mail.ru>
Date:   Sat Apr 12 19:05:46 2025 +0400

    Development of components and their environment

commit abfb0d739bb14f71955f1650d35f944349d31d19
Author: Ivan <yanickiy.ivan@mail.ru>
Date:   Sat Apr 12 19:03:50 2025 +0400

    Preparatory stage

commit 51b7cf2fc89da044c48155b6037da0134f99cb89
Author: Ivan <yanickiy.ivan@mail.ru>
Date:   Sun Mar 23 20:35:46 2025 +0400

    Initialize project using Create React App
```

Рисунок 24 - Логи коммитов

Всё это необходимо для того, что если в будущем понадобится вернуться к старым решениям, чтобы посмотреть, что было до этого или же вернуть всё к изначальному виду до них, то это возможно будет свершить с помощью обращения к любому из зафиксированных коммитов по их хеш-коду.

Результатом разработки данного проекта стало создание минимально жизнеспособного продукта, реализованного с использованием функционального подхода в React. Он представляет собой базовое решение, включающее ключевой функционал, необходимый для дальнейшего применения и интеграции в сторонние веб-проекты.

В процессе разработки применялись современные подходы и инструменты, такие как TypeScript для обеспечения строгой типизации и повышения надёжности кода, а также система контроля версий Git для эффективного управления изменениями. Архитектура компонентов строилась на основе атомарного дизайна, что обеспечило модульность и переиспользуемость UI-решений. Для именования классов применялась методология БЭМ, способствующая читаемости и структурированности CSS [25].

Кроме того, были реализованы юнит-тесты, позволяющие гарантировать корректность работы отдельных компонентов и повысить общее качество системы. Разработка велась в соответствии с заранее определёнными требованиями, а также методологиями и идеями, сформулированными на этапе планирования, что позволило достичь высокого уровня стабильности, масштабируемости и соответствия проектным задачам.

3.2 Применение разработанного алгоритма на примере реальных данных

После того, как разработка была завершена и получен код, который способен удовлетворять минимальным потребностям, необходимо рассмотреть наглядно то, что он способен делать, какие функции способен

выполнять и какие возможности есть для его кастомизации в зависимости от задач. Также обязательно продемонстрировать состоятельность и устойчивость всего конечного проекта путём демонстрации процента покрытия всего кода интеграционными тестами с описанием, какие критерии входят в конечную оценку.

Если обычная проверка выполнения тестов показывает, работает ли конкретный участок кода корректно, так как ожидается, показывает пройден тест или нет, то покрытие отвечает за общее состояние всего кода, который проверяется, имеет качественную характеристику, которая состоит из 4 критериев:

- Statements - сколько всех операторов покрыто.
- Branches - условные конструкции: if, switch, ?:
- Functions - сколько функций было вызвано хотя бы раз.
- Lines - просто количество строк кода, которые выполнились.

Все эти критерии оцениваются в каждом файле, которые были обозначены как тестовые, после чего формируется среднее арифметическое по каждому из пунктов.

Конечный результат представляется в виде таблицы, где представлены все тестируемые файлы, итоговое значение по всему проекту, критерии оценивания и процент, который был выполнен для них, также указываются строки, которые не были покрыты, если таковы имеются. Итоговый результат по проекту, где видно, что процент покрытия достигает 100 процентов по каждому из критериев, а также видно общее количество тестов, представлен на рисунке 25.

Для более удобной, интерактивной и детальной формы такой отчёт может представляться в разных формах в зависимости от целей, например, для отладки обычно представляется в виде HTML документа, который позже можно открыть в браузере и посмотреть, какие участки были выполнены, а какие нет, или же в формате текстового документа - Info, который может быть использован для процессов CI.

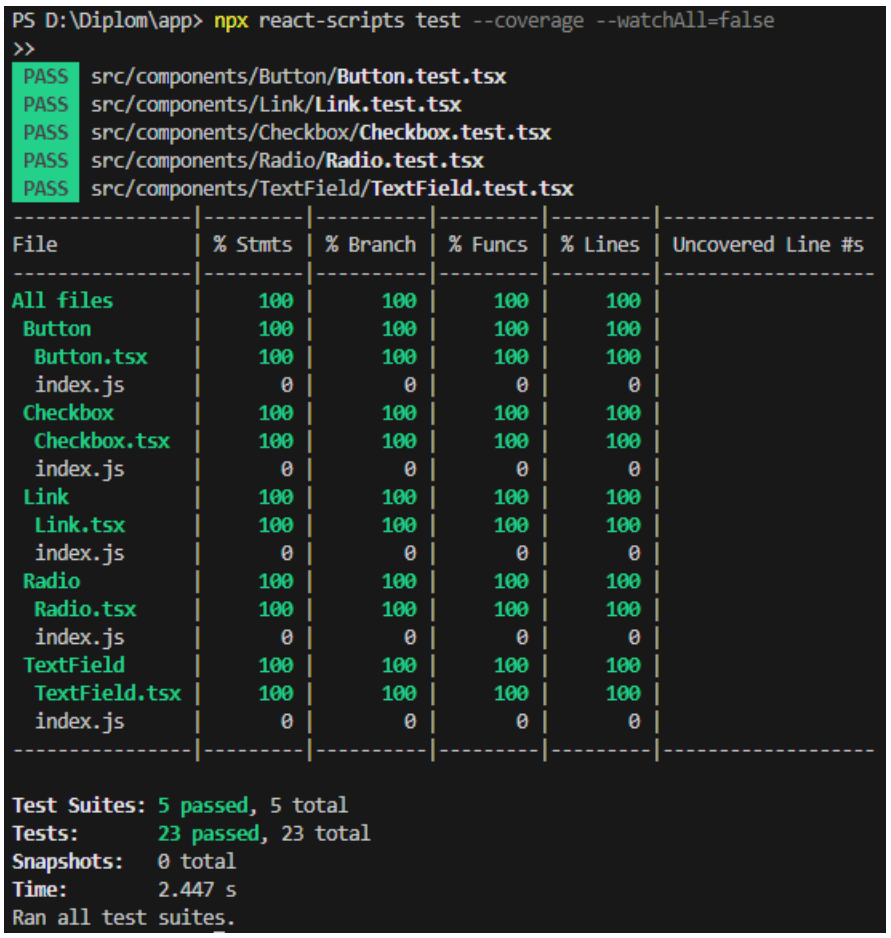


Рисунок 25- Покрытие тестами

Все такие отчёты формируются после проверки на процент покрытия всего проекта в указанной директории. В данном случае были сформированы 2 вида: HTML и Info. Директория с отчётами показана на рисунке 26.

После того, как были сформированы все отчёты и проверено покрытия и устойчивость кода следует ознакомиться с возможностями функционала. Всего в проекте было реализовано 5 компонентов: кнопка, чекбоксы, радиокнопки, поля для ввода текста и ссылки. Каждый из них имеет аргументы и атрибуты, которые влияют на их внешний вид и представление, некоторые из них могут повторяться и иметь общий функционал, поэтому, чтобы не повторяться, их следует вынести и рассмотреть отдельно перед всеми, например: size, variant(color), className, disabled. Первый из них отвечает за размеры самих компонентов, размер шрифта и внутренние отступы. Всего

было разработано 3 варианта: small, middle, large, что соответствует 12px, 14px и 16px размера шрифта. Второй отвечает за цветовую вариацию компонента, всего доступно 4 цвета, которые были определены для всего проекта и будут встречаться в других компонентах, обозначенные как primary(голубой), secondary(синий), danger(красный), disabled(серый), можно указать лишь первые 3, так как последний появляется лишь при условии, если элемент является недоступным, что определяется значением другого параметра(disabled). ClassName показывает на то, какие классы необходимо добавить к компоненту, чтобы можно было внести правки, если уже готовых решений недостаточно, и необходимо что-то дополнительное, например, задать для каждого такого компонента отступы. Последний отвечает за состояние заблокированности, которое принимает значения boolean(true или false), что сообщает о том, будет ли элемент отвечать на взаимодействия с ним, а также это сказывается на цвете.

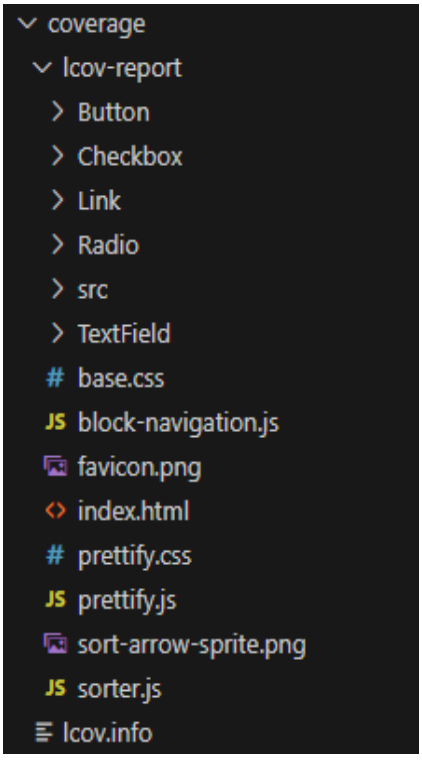


Рисунок 26- Директория с отчётами

На рисунке 27 продемонстрированы разновидности кнопок, которые можно получить. Они имеют 8 параметров: цвет, размер, заблокированность, дополнительные классы, стиль заполнения цветом, текст, слушатель нажатия, тип кнопки. Первые четыре параметра были рассмотрены ранее, так как они имеют общее поведение и представление для всех разработанных компонентов. Стиль заполнения(outlined) указывает на то, следует ли компоненту заполнить внутреннее пространство указанными цветом или оставить прозрачным изменив лишь цвет текста и рамки. Переданный текст будет указан внутри самой кнопки, если же ничего не будет передано, то будет выведен стандартный текст “Button”. Слушатель нажатия является функцией, которая при её передаче будет вызвана при нажатии на кнопку, если она будет доступна, например, такой подход позволяет собранные данные из формы, в которой располагается кнопка, отправить на внешние ресурсы.

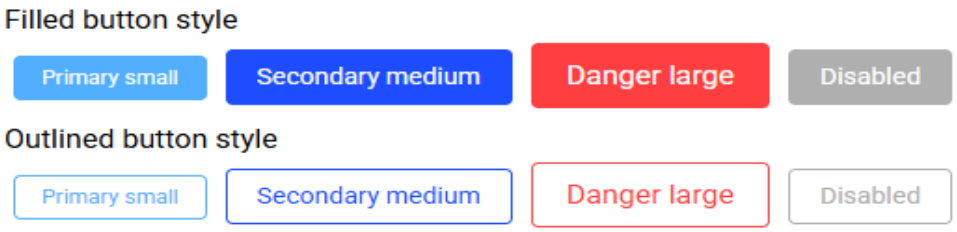


Рисунок 27 - Представления кнопок

На рисунке 28 показано пример того, как вызываются эти компоненты для разных вариаций и какие параметры при этом указываются. Также следует обратить внимание на то, что каждый компонент имеет указанный параметр className, поведение которого рассматривалось ранее, это позволяет задать отступ всем кнопкам отступ справа, что не предусмотрено стандартной реализацией.

Следующим компонентом является input с типом checkbox, который используется в формах, когда необходимо предоставить пользователю множественный выбор или же его полное отсутствие, если такое допускают

требования. Чекбоксы принимают в себя 8 параметров: доп.классы, текст лейбла, id лейбла, название инпута, доступность чекбокса, состояние чекбокса, размер и слушатель изменения состояния.

```
<ButtonUI
  className={styles.buttons}
  size='small'
  type='button'
>
  Primary small
</ButtonUI>
<ButtonUI
  className={styles.buttons}
  variant='secondary'
  disabled={true}
  outlined={true}
  type='submit'
>
  Secondary medium
</ButtonUI>
```

Рисунок 28 - Пример вызова ButtonUI

Текст лейбла отвечает за значение текста, который будет располагаться рядом с самим чекбоксом, по умолчанию никакого значения не задано, то есть при его отсутствии будет исключительно блок выбора. ID лейбла отвечает за то, какой лейбл будет относиться к какому полю выбора, эти значения обязательно должны быть уникальными, иначе корректная работа будет нарушена, и при нажатии на один компонент может меняться состояние другого, поэтому оно является единственным обязательным значением, которое необходимо указывать при вызове этого компонента. Название инпута отвечает за значение атрибута name у input, который позволяет в более удобной форме обращаться к ним при сборе либо изменении их значений, например, будет объект, который хранит информацию о значениях элементов в форме, где ключи, это поля name у этих компонентов. Параметр доступности

отвечает за то, возможно ли взаимодействовать с этим компонентом или нет, если он таковым является, то его цвет лейбла и самого блока выбора меняется на серый(disabled). Параметр состояния чекбокса отвечает за то, считается он выделенным или нет, по умолчанию его значение false и изменяется посредством слушателя. Параметр размера отвечает за величину шрифта, а также за размер самого блока с галочкой. Параметр слушателя представляет из себя функцию, которая срабатывает при изменении состояния компонента, что позволяет, как минимум, изменять его значение, так как оно задаётся снаружи. Демонстрация того, как выглядят вариации чекбокса показаны на рисунке 29, а пример его использования на рисунке 30.

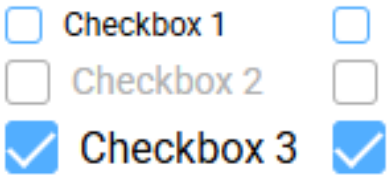


Рисунок 29 - Компонент чекбоксы

```
<CheckboxUI
  labelText="Checkbox 2"
  labelID="Checkbox 2"
  size="medium"
  disabled={true}
  onChange={handleChange}
/>
<CheckboxUI
  labelText="Checkbox 3"
  labelID="Checkbox 3"
  size="large"
  name="checkbox3"
  checked={formData.checkbox3}
  onChange={handleChange}
/>
```

Рисунок 30 - Вызов компонента CheckboxUI

Компонент «радиокнопка» очень схож своим функционалом и параметрами с чекбоксами за исключением того, что они предоставляют выбор лишь одного варианта из предложенных, а также тем, что чекбоксы являются самостоятельными, а радиокнопки формируются группой. Рассмотрев его параметры: имя группы, элементы группы, лейбл группы, выбранное значение, размер и слушатель изменения состояния, становится понятно, что последние два своим функционалом идентичны одноименным, что представлены в предыдущем компоненте. Имя группы - это параметр, который является идентификатором группы, то есть без него не будет возможности однозначно определить в какой группе был выбран элемент, поэтому он является обязательным при вызове компонента. «Элементы группы» также являются обязательным параметром, который определяет состав списка вариантов для выбора, он представляет из себя массив объектов, в которых содержится 3 поля: ID, лейбл, значение, они нужны для идентификации элементов списка, текста рядом с полем выбора, а также значения, которые возвращаются при выборе элементов в списке, соответственно. Лейбл группы является текстом, который будет указан над всем списком вариантов, который при необходимости может отсутствовать. Выбранный элемент указывает на то, какой из представленных в списке ранее должен быть отмечен, по умолчанию никакой из них является выбранным. На рисунке 31 показаны вариации, как этот элемент может выглядеть, а на рисунке 32 же показано то, как этот компонент вызывается.

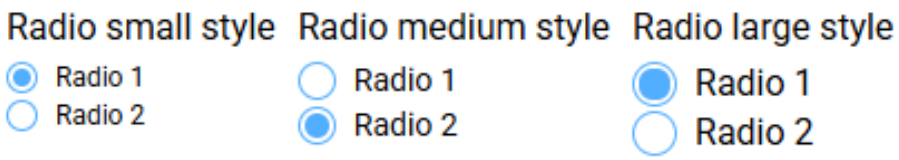


Рисунок 31 - Представление радиокнопки

```
<RadioUI
  className={styles.radio}
  groupName="option"
  groupLabel="Radio medium style"
  groupElements={radioButtonsGroupMedium}
  value={formData.option}
  size="medium"
  onChange={handleChange}
/>
```

Рисунок 32 - Вызов компонента RadioUI

Компонент текстового поля представляет из себя блок, куда вводится любая строчная информация: имя, логин, почта, пароль. Он принимает в себя, за исключением ранее описанных и схожих параметров доп.классов, текста лейбла, id лейбла, слушателя, имени инпута и размера, 3 аргумента: плейсхолдер, значение инпута и тип инпута. Плейсхолдер - это подсказка, которая показывается до тех пор, пока пользователь не введёт какой-либо текст, обычно она включает в себя пример того, что именно ожидается, чтобы это передали, например, образец почты или логина. Значение инпута - это значение, которое изначально задано в поле ввода текста, в отличии от плейсхолдера, он не исчезает полностью, а редактируется, по умолчанию это значение пустой строки. Последним является тип инпута, который может принимать два значения: «text» или «password», первый подразумевает, что вводится текст, который не обязательно скрывать во время ввода, например, почта или логин, второе же означает, что текст будет замещён звёздочками при отображении, чтобы скрыть его на этапе ввода, и чаще всего применяется при вводе пароля. Реализации вариантов этого компонента показаны на рисунке 33. Способы его применения в коде со всеми параметрами проиллюстрированы на рисунке 34.

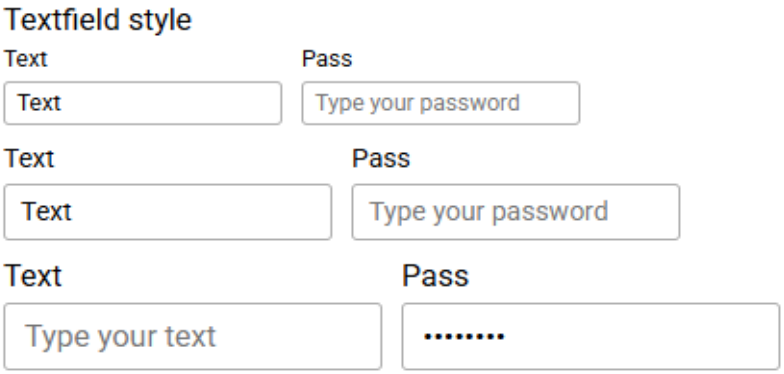


Рисунок 33 - Компонент ввода текста

```
<TextFieldUI
  className={styles.textField}
  labelId="TextField1m"
  labelText="Text"
  placeholder="Type your text"
  type="text"
  size="middle"
  value={formData.name}
  onChange={handleChange}
  name='name'
/>
<TextFieldUI
  className={styles.textField}
  labelId="TextField2m"
  labelText="Pass"
  placeholder="Type your password"
  type="password"
  size="middle"
  name='password'
  value={formData.password}
  onChange={handleChange}
/>
```

Рисунок 34 - Применение компонента в коде

Последним компонентом является ссылка, которая используется для навигации по страницам веб-приложения. Для него используются 6 параметров: дополнительные классы, текст ссылки, ссылка, нижнее подчёркивание, цвет и размер. Стоит уделить внимание лишь некоторым, поскольку часть из них уже были приведены и описаны ранее в тексте: текст

ссылки, который будет отображаться внутри ссылки, по умолчанию у него ничего не указано, поэтому ссылки даже не будет видно, сама ссылка, куда необходимо совершить переадресацию, изначально указывается #, поэтому ссылка никуда не приведёт после перехода по ней, и параметр нижнего подчёркивания, который при значении true изменит стиль отображения и добавит черту под текстом.

Внешнее представление компонента показано на рисунке 35, а на рисунке 36 показано его использование.

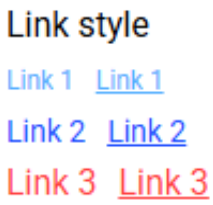


Рисунок 35 - Варианты ссылок

```
<LinkUI
  className={styles.link}
  text="Link 1"
  href="#"
  size="small"
  color="primary"
/>
<LinkUI
  className={styles.link}
  text="Link 1"
  href="#"
  size="small"
  underline={true}
  color="primary"
/>
```

Рисунок 36 - Вызов ссылки в коде

В заключение можно сказать, что была проведена комплексная проверка UI-компонентов с использованием интеграционного тестирования. Все ключевые элементы проекта были протестированы на корректность отображения, обработку пользовательских событий и взаимодействие с пропсами. Это позволило удостовериться в корректной работе компонентов в рамках общей системы. Были детально рассмотрены каждый из критериев, по которым строится оценка покрытия кода: Statements, Branches, Functions, Lines. Для каждой метрики рассчитан процент покрытия, и обеспечено соответствие проекту целевым показателям (не менее 90%). Также были разобраны методики интерпретации этих показателей и даны критерии оценки эффективности покрытия, учитывающие не только количественные, но и качественные аспекты тестирования.

Интеграционные тесты подтвердили стабильность и надежность реализованных компонентов, а высокие показатели покрытия кода обеспечивают удобство поддержки и возможность масштабирования библиотеки в будущем.

После тестирования был выполнен детальный обзор каждого из компонентов библиотеки – Button, Link, Checkbox, Radio, TextField. Для каждого компонента были описаны:

- их основной функционал – как они ведут себя в интерфейсе;
- возвращаемая разметка и структура DOM;
- принимаемые пропсы и их влияние на внешний вид и поведение компонента;
- роль компонента в общей системе и влияние на пользовательский опыт.

Таким образом, проведённый анализ и тестирование обеспечивают уверенность в качестве компонентов и подтверждают их готовность к использованию в полноценных веб-приложениях.

3.3 Тестирование алгоритма на реальных проектах и данных

Продемонстрировав возможности разработанной библиотеки и её компонентов, следующим этапом было продемонстрировать её возможности на живом примере, создать минимальный макет, который возможно было бы встретить в реальных проектах, где эта библиотека могла бы использоваться, помимо этого необходимо реализовать этот же макет без использования каких-либо подобных решений и с использованием альтернативного решения для возможности наглядного сравнения.

Сначала было принято решение создать макет без использования библиотек с использованием исключительно HTML и SASS, полученный результат представлен на рисунке 37.

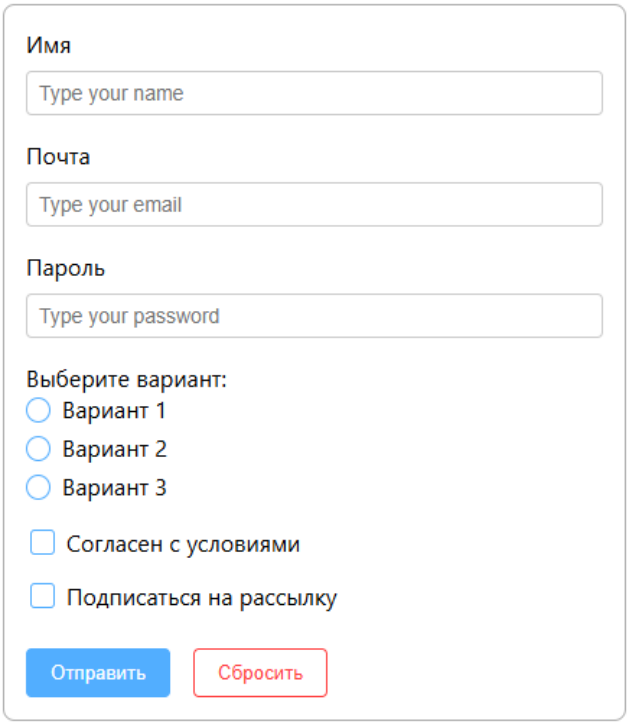


Рисунок 37 - Макет без готовых решений

Основным преимуществом такого подхода является точечная настройка каждого элемента, указание наличия каждого элемента в дереве, а также

детальная настройка стилей для каждого из них. Однако основным минусом является то, что итоговое дерево разметки получается значительно больше, чем если бы использовались библиотеки, включающие готовые компоненты с готовой стилизацией, помимо этого для каждого из них необходимо отдельно описывать стили.

На рисунке 38 показана часть HTML разметки, которая получилась при этом подходе. На нём видно, что для правильного изображения элементов и их позиционирования необходимо использование дополнительных обёрток и элементов, например, для формирования одной строки с полем текстового ввода «Имя» понадобился дополнительный div, чтобы можно было назначить обособленную вертикальную структуру, а для текстового интерактивного лейбла понадобился дополнительно label.

```
<div className="form-group">
  <label htmlFor="name">Имя</label>
  <input
    type="text"
    id="name"
    name="name"
    value={formData.name}
    onChange={handleChange}
    placeholder='Type your name'
  />
</div>
```

Рисунок 38 - HTML разметка без библиотек

На рисунке 39 показана часть SASS файла, который был необходим в данном подходе, чтобы указать даже минимальные стили.

Следующим был разработан макет с использованием популярного готового решения MaterialUI, которое включает в себя обширный список готовых компонентов и способов настройки их.

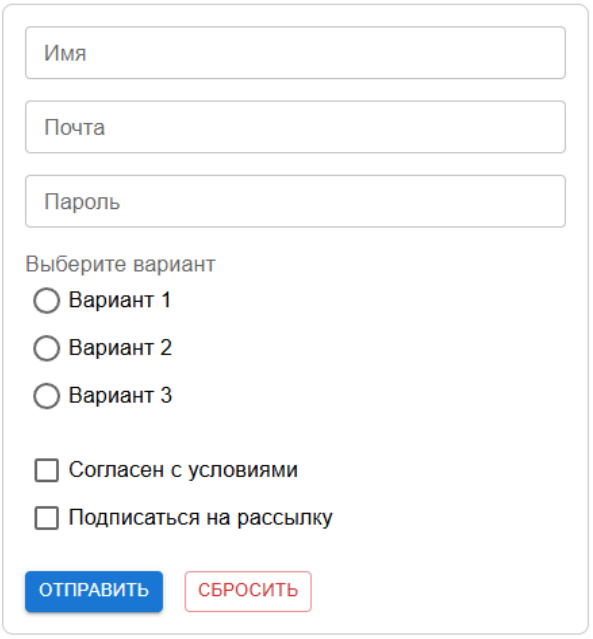
```sass
.form
  max-width: 400px
  margin: 20px auto
  display: flex
  flex-direction: column
  gap: 1rem
  border: 1px solid #afafaf
  padding: 15px
  border-radius: 8px

  &-group
    display: flex
    flex-direction: column

    &__label
      margin: 0 0 10px 0
```

Рисунок 39 - SASS файл

На рисунке 40 показан макет с использованием вышеописанной библиотеки.



Рисунке 40 - Макет на MaterialUI

Главным моментом оказывается то, что в этом случае SASS файл отсутствует, так как всю настройку внешнего вида, организации и позиционирования на себя берут параметры и дополнительные атрибуты самих компонентов, помимо этого присутствуют компоненты, которые используется исключительно для правильного представления макета на экране, которые также настраиваются своими собственными параметрами, помимо этого имеются дополнительный параметр, куда указываются собственные строчные представления для стилей, например, размер шрифта, отступы, размер и цвет границ блока. Но как видно по конечному макету цвета отличаются, поскольку используются заложенные, помимо этого используется довольно много собственных параметров, которые могут быть непонятны человеку, который ранее с не работал с ними. На рисунке 41 показан пример использования.

```
<Box
  component="form"
  onSubmit={handleSubmit}
  sx={{
    maxWidth: 400,
    mx: 'auto',
    p: 2,
    border: '1px solid #ccc',
    borderRadius: 2,
    display: 'flex',
    flexDirection: 'column',
    gap: 2,
  }}
>
  <TextField
    label="Имя"
    name="name"
    value={formData.name}
    onChange={handleChange}
    fullWidth
    size="small"
  />
```

Рисунок 41 - Разметка с MaterialUI

Последним был собран макет с использованием разработанной библиотеки, который показан на рисунке 42. Он очень похож на первый сделанный макет, но следует сделать уточнения: во-первых, цвета кнопок схожи, потому что цвета совпадают с заданными в библиотеке, в противном случае они бы различались, за исключением той ситуации, что их пришлось дополнительно указывать через классы, однако такой вариант отсутствует в ранее рассмотренной библиотеке, во-вторых, заметны расхождения в размере шрифтов, но это также возможно дополнительно редактировать через классы. Однако в отличии от первого варианта здесь отсутствуют дополнительные обёртки и классы для более точного расположения, многое настраивается посредством заданных параметров, что показано на рисунке 43, многое уже включено в сами компоненты, также это отображается на конечном SASS файле, где присутствует исключительно два класса для дополнительных настроек отступов, это можно увидеть на рисунке 44.

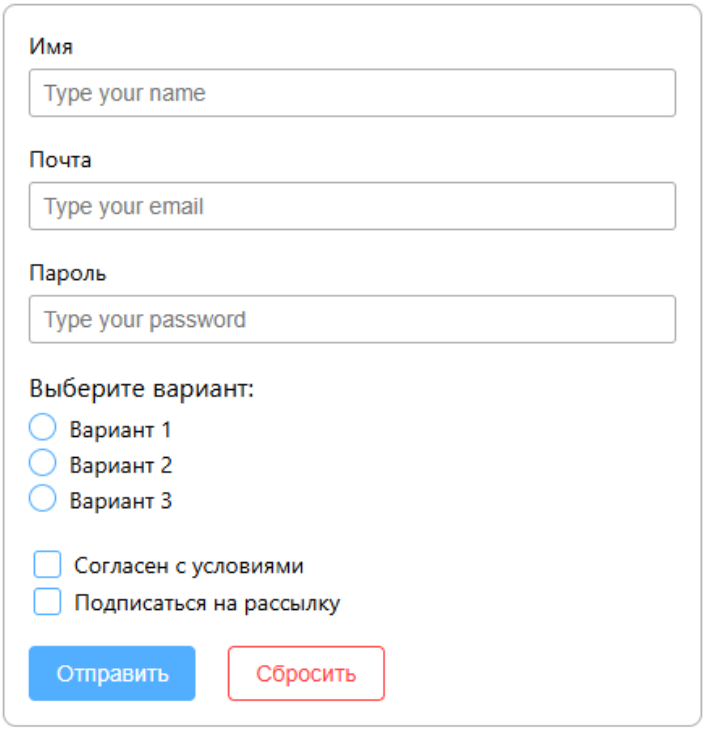


Рисунок 42 - Макет, созданный разработанной библиотекой

```jsx
<form className="form" onSubmit={handleSubmit}>
  <TextFieldUI
    labelId="name"
    labelText="Имя"
    placeholder="Type your name"
    type="text"
    name='name'
    value={formData.name}
    onChange={handleChange}
  />

  <TextFieldUI
    labelId="email"
    labelText="Почта"
    placeholder="Type your email"
    type="text"
    name='email'
    value={formData.email}
    onChange={handleChange}
  />
```

Рисунок 43 - Разметка JSX при разработанной библиотеке

```scss
.form
  max-width: 400px
  margin: 20px auto
  display: flex
  flex-direction: column
  gap: 1rem
  border: 1px solid #afafaf
  padding: 15px
  border-radius: 8px

.button
  margin: 0 10px 0 0
```

Рисунок 44 - SASS файл при разработанной библиотеке

Проведённое сравнение трёх подходов к созданию макета продемонстрировало преимущества и недостатки каждого из них. При разработке макета без использования библиотек (только с HTML и SASS) удалось добиться полной гибкости в настройке структуры и стилей. Однако этот подход оказался крайне трудоёмким: разметка усложнилась из-за

необходимости создавать дополнительные обёртки и элементы для правильного позиционирования, а также вручную прописывать стили даже для базовых вещей. Такой способ требует больше времени на разработку и влечёт за собой усложнение поддержки кода.

Использование готового решения на базе Material UI значительно упростило процесс разработки. Компоненты библиотеки обеспечивают удобные параметры настройки внешнего вида и поведения элементов без необходимости написания собственного SASS-кода. Однако в процессе работы проявились и минусы: структура компонентов стала зависимой от специфичных параметров Material UI, а стили по умолчанию не всегда соответствовали ожидаемому дизайну, что потребовало дополнительного изучения библиотеки для более точной настройки.

Разработка макета с использованием собственной библиотеки компонентов показала наилучший баланс между удобством и гибкостью. Библиотека позволила сократить объем разметки за счёт встроенных параметров компонентов, избавила от необходимости вносить излишние обёртки и минимизировала написание пользовательских стилей. При этом сохранена возможность тонкой настройки внешнего вида через параметры и классы, обеспечивая адаптивность под разные требования проекта.

Таким образом, результаты демонстрации макетов доказывают, что интеграция собственной UI-библиотеки в процесс разработки существенно повышает эффективность создания современных веб-приложений.

3.4 Сравнение полученных данных с идеальными компонентами и оценка эффективности

Последним этапом является оценка эффективности разработанной библиотеки с использованием ранее созданной математической модели, которая позволит оценить качество и эффективность самой библиотеки, а

также получить показатель того, как она сказывается на проекте, в который интегрируется.

Для расчёта показателя качества библиотеки исходя из ранее описанной модели необходимо получить значения 5 показателей: размера библиотеки, процент покрытия тестами, количество соблюдённых методологий, количество критических ошибок линтера и количество замечаний линтера. Итоговый размер решения до сжатия составляет 59.9 КБ, что довольно маленький показатель, но в силу небольшого количества реализованных компонентов и низкого количества зависимостей, это значение оказывается ожидаемым. Весь разработанный код был покрыт интеграционными тестами, таким образом процент покрытия составляет 100%. Изначально было выделено 3 методологии, по которым следовало вести всю разработку, в конечном счёте удалось реализовать их все: Atomic Design, BEM и DRY, таким образом этот показатель принимает значение 3. Последние два значения получаются путём запуска службы линтера в разрабатываемом проекте, в результате он не показал ни замечаний, ни ошибок, что говорит о том, что оба эти значения равняются 0, пример его работы показан на рисунке 45, а на рисунке 46 продемонстрировано так, как выглядят ошибки(красным цветом) и замечания(жёлтым цветом) с указанием файла, строки и символа, где была найдена проблема.

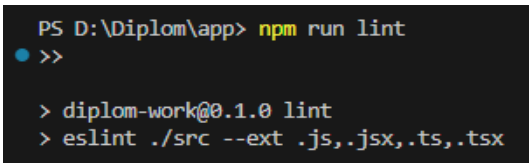


Рисунок 45 - Показатели линтера

Полученные значения необходимо привести к формату переменных для их дальнейшего использования из чего получается: $IT = 100, Mn = Mn_m = 3, W = 0{,}058, W_m = 1, LE = 0, LC = 0$.

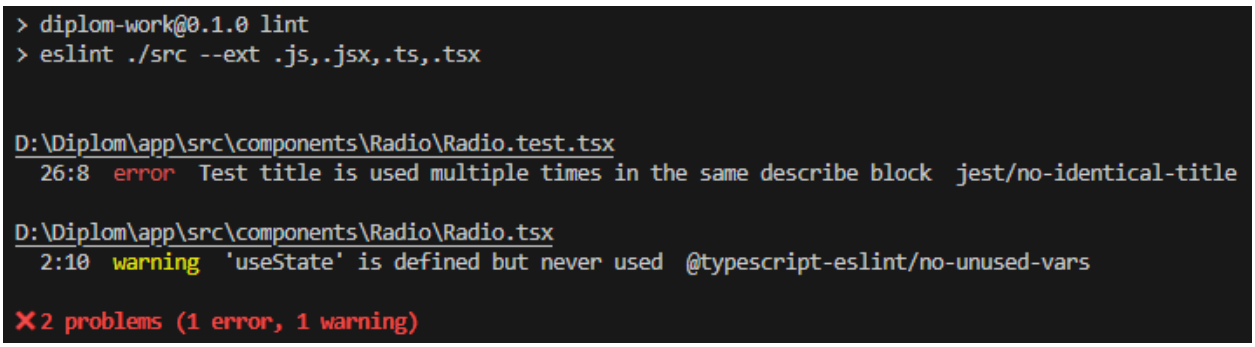


Рисунок 46 - Примеры ошибок и замечаний

Для получения итоговой оценки необходимо рассчитать оценку каждого из показателя по ранее описанным формулам: Q1, Q2 Q3, Q4, Q5, которые позже преобразуются в результирующую. Также необходимо для каждого из них указать их значимость в итоговой оценке, возьмём, что все они равнозначны и показатель их веса будет равняться 1. Расчёты каждого из свойства представлены ниже в формулах 11.

$$
\begin{gathered}
Q_1 = 1 * \frac{IT}{100} = 1 * \frac{100}{100} = 1 * 1 = 1, \\
Q_2 = 1 * \frac{Mn}{Mn_m} = 1 * \frac{3}{3} = 1 * 1 = 1, \\
Q_3 = 1, \text{поскольку } W < W_m, \\
Q_4 = 1 * e^{-0.6 * LE} = 1 * e^{-0.6 * 0} = 1 * e^0 = 1 * 1 = 1, \\
Q_5 = 1 * e^{-0.1 * LC} = 1 * e^{-0.1 * 0} = 1 * e^0 = 1 * 1 = 1.
\end{gathered} \tag{11}
$$

После того, как были получены значения критериев каждого свойства можно рассчитать итоговое, которое будет представлено в формуле 12. Поскольку определение значимости каждого из критериев не было сверх необходимым в рамках поставленной задачи, а также того, что сложность и объём разработки не были велики, удалось достичь итогового значения качества равном 1, что является верхней границей оценки этой математической модели.

$$Q \quad = \frac{Q_1 + Q_2 + Q_3 + Q_4 + Q_5}{w_1 + w_2 + w_3 + w_4 + w_5}; = \frac{1+1+1+1+1}{1+1+1+1+1} = 1; \qquad (12)$$

Проведя оценку библиотеки, можно приступить к определению эффективности того, как она ускоряет и меняет процесс написания кода. Для проведения оценочных работ будут взяты 2 макета, которые рассматривались ранее: с использованием библиотеки и без. В этом случае необходимо также определить значения необходимых критериев и формализовать их для использования в математической модели: количество строк JSX разметки, количество строк CSS(SASS), количество компонентов, а также время сборки, их необходимо взять для каждой отдельной модели.

Проведя анализ получаем, что количество строк JSX разметки становится равным 106 и 73 для первого и второго способа разработки соответственно, это соответствие будет сохраняться дальше, количество строк CSS(SASS) принимает значение 151 и 12, в обоих случаях общее количество компонентов составило 0, время же сборки составило 6.248 и 6.203 секунд, что показано на рисунках 47 и 48.

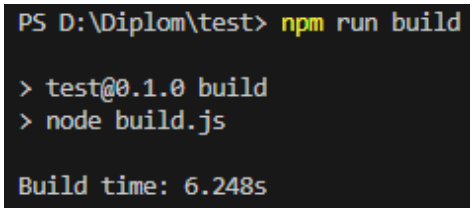


Рисунок 47 - Время сборки без библиотеки

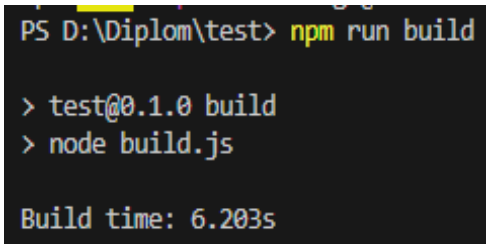


Рисунок 48 - Время сборки с библиотекой

Приведя всё к формальному вида, для возможности использования этих данных в модели, не забывая, что следует учесть количество компонентов в каждом подходе, а он равен 1, так как кроме главного файла иных нет, получаем значения равные $J_0 = 151, J_1 = 73, S_0 = 106, S_1 = 12, T_0 = 6.248, T_1 = 6.203, C_0 = 1, C_1 = 1$.

Теперь возможно провести дальнейший анализ свойств, который представлен в формуле 13 для каждого из свойств: CSS(SASS), JSX и время. Для конечного вычисления также понадобятся значения весов, которые бы определяли значимость каждого из фактора. Поскольку показатель строк кода для JSX и CSS являются самыми наглядными и ощутимым их значимость примем за 0.8, а критерию времени зададим меньшее значение 0.2, так как в силу того, что тестовый проект небольшой, и время сборки сильнее меняется и этот эффект сложнее ощутить.

$$\begin{aligned} E_j &= w_j \frac{J_1}{J_0} = 0.8 * \frac{73}{151} = 0.39, \\ E_s &= w_s \frac{S_1}{S_0} = 0.8 * \frac{12}{106} = 0.09, \\ E_t &= w_t \frac{T_1}{T_0} = 0.2 * \frac{6.203}{6.248} = 0.2. \end{aligned} \quad (13)$$

Выполнив расчёты для каждого из свойств, можно узнать финальную эффективность. Вычисления представлены в формуле 14.

$$E = 1 - \frac{C_1}{C_0} * \frac{(E_j + E_s + E_t)}{w_j + w_s + w_t} = 1 - \frac{1}{1} * \frac{(0.39 + 0.09 + 0.2)}{0.8 + 0.8 + 0.2} = 1 - \frac{0.68}{1.8} = 0.62 \quad (14)$$

Результатом вычисления эффективности внедрения библиотеки стало значение 0.62, это можно интерпретировать как ускорение и улучшение разработки, хоть не самое крупное. Самый большой вклад имеет сокращение количество строк стилей, поскольку основная их часть была вынесена в

содержимое компонентов библиотеки, на втором месте оказалось сокращение строк JSX разметки по той же причине, на третьем месте было время, которое сократилось очень слабо и внесло незначительный вклад, а поскольку количество компонентов не изменилось, то это никак не повлияло на конечную оценку.

Подводя итоги, можно сказать, что следуя выделенными подходам, методологиям, архитектуре и технологиям получилось разработать библиотеку, которая по оценкам первой разработанной математической модели имеет хорошее качество, но стоит обратить внимание на то, что в ней было реализован самый малый необходимый функционал, который мог бы быть использован в настоящих коммерческих проектах, поэтому с её масштабированием и более оценочным отношением к каждому её критерию, показатель её качества может упасть, что однако отражает способность оценочной модели реалистично оценивать качества решений, которые бы участвовали в анализе.

Решение смогло положительно сказаться на скорости и качестве разработки проверочного проекта, в который оно внедрялось: удалось сократить количество используемых строк кода как HTML разметки, так и TSX кода, пусть и не самым лучшим образом, какой можно было бы ожидать, но такое значение можно оправдать тем, что размер тестового макета не был соизмерим с масштабами настоящих проектов, где количество строк кода и компонентов значительно больше, из-за чего оцениваемые параметры, которые используются в математической модели, могли принять более весомые значения.

Заключение

Выпускная квалификационная работа была посвящена разработке и интеграции UI-библиотеки компонентов на базе React, направленной на повышение качества и скорости разработки веб-приложений. В современных условиях цифровой трансформации бизнеса всё большее значение приобретает стандартизация интерфейсов и ускорение разработки с сохранением высокого уровня качества. Проведённое исследование и реализация проекта показали, что грамотно спроектированная библиотека компонентов способна существенно повлиять на эффективность разработки интерфейсов, улучшить читаемость, переиспользуемость и стабильность [9][4].

Первым этапом было выполнение анализа уже готовых, существующих решений, которые положительно сказываются на скорости и качестве разработки, UI-библиотек таких как Material UI, Ant Design, Bootstrap. Все они оценивались по определённым критериям и каждое из них имеет свои положительные стороны, на которых изначально и был упор, но также были выделены и слабые стороны, которые могли бы стать критичными при разработке, из-за чего было принято решение о создании собственной библиотеки, которое бы реализовало функционал уже имеющихся, а также закрывало из слабые стороны.

Проанализировав имеющиеся разработки были выделены требования и рекомендации, которым необходимо было бы следовать на этапе разработке собственной UI библиотеки. Определены методологии, которые позволяли бы сохранить читаемость кода, уменьшить логическую нагрузку на один компонент, распределяя её между другими более мелкими. Была составлена файловая структура проекта, которой необходимо придерживаться для большой организованности и читаемости. Также определены дополнительные инструменты, которые должны были бы использоваться для большего контроля качества, повышения устойчивости и большей гибкости на этапе

создания: React, TypeScript, SASS [8][3].

Утвердив принципы и технологии, которые могли бы быть обозначены как теоретическая составляющая, необходимо было продумать, по какому принципу, каким метриками оценивалось бы качества библиотеки, а также то, насколько она эффективно выполняет свою функцию, насколько она положительно сказывается на разработке, для этого необходимо было разработать математические модели, которые бы позволяли это оценить в нормальном виде. Сперва были выделены основные свойства, по которым необходимо было бы проводить оценки, показатели, которые бы отражали суть, по которым были разработаны модели. Поскольку при разных обстоятельствах каждый критерий может иметь разную значимость, для всех них были добавлены весовые коэффициенты, для регулировки их вклада в конечную оценку. Формула 15 служит для оценки качества библиотеки, где множители от Q1 до Q5 оценивают каждое отдельное свойства [6][26].

$$Q = \frac{Q_1 + Q_2 + Q_3 + Q_4 + Q_5}{w_1 + w_2 + w_3 + w_4 + w_5}; \quad (15)$$

Формула 16 отражает улучшение качества после интеграции в проект, где Ej, Es, Et - отдельные критерии, по которым проводится оценка.

$$E = 1 - \frac{C_1}{C_0} * \frac{(E_j + E_s + E_t)}{w_j + w_s + w_t}; \quad (16)$$

Этап разработки включал в себя создание базовых компонентов, из которых возможно создать минимальный рабочий прототип, каждый из которых был создан в нескольких стилевых вариациях с применением заранее определённых цветов: primary, secondary, danger, disabled. Большая часть настройки поведения и визуального представления осуществляется посредством передачи необходимых параметров например, цвета, значения, слушателя, что уменьшает необходимость дополнительного описания стилей

и логики [8][3]. Все разработанные компоненты и их представления с вариациями показаны на рисунке 49 ниже.

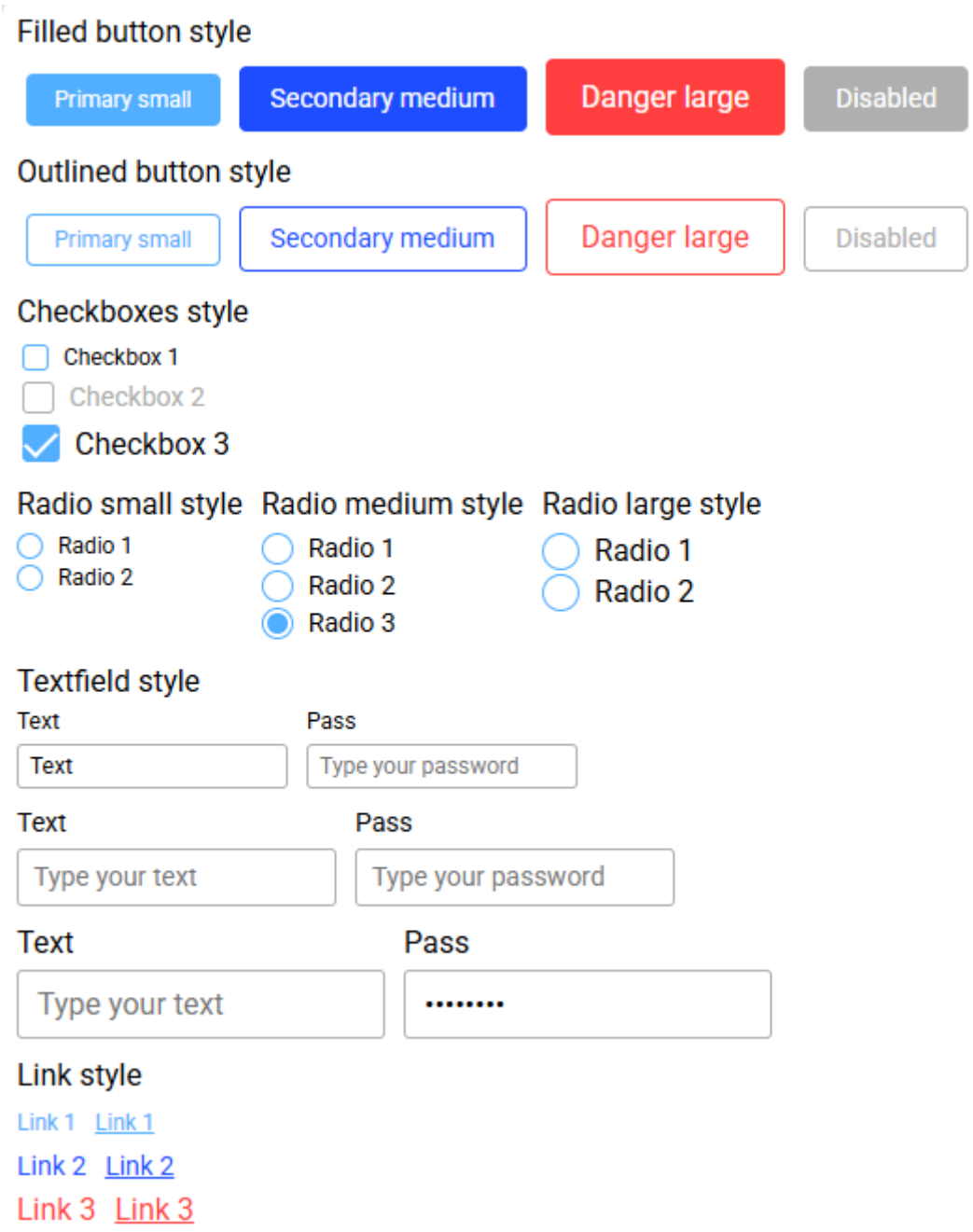


Рисунок 49 - Разработанные компоненты

Для обеспечения надёжности каждый из компонентов был протестирован с помощью Unit-тестов, что позволяет гарантировать их устойчивость и вести дальнейшую разработку и улучшение библиотеки [19].

Последним этапом стало проведения тестирование и анализа результатов работы с помощью составленных ранее математических моделей. Первым этапом было проверено качество самой библиотеки. Был оценён каждый параметр и назначен каждому из них весовой коэффициент 1, поскольку каждый критерий был важен и никакие обстоятельства не ограничивали их вклад. Результирующей оценкой стала 1, что говорит о максимальной оценке качества самой библиотеки, однако стоит отметить, что будь проект значительно крупнее, сложнее и ведись он в условиях корпоративной разработки показатели могли быть хуже, поскольку не было жёстких требований и ограничений по времени. Вторым этапом стала оценка вклада библиотеки в разработку по второй разработанной математической модели. После определения весовых коэффициентов и конечных расчётов оценка была равна 0.62, что является довольно хорошим показателем, говорящем о том, что её интеграция положительно сказывается на скорости и эффективности разработки. Об этом стоит сказать, что тестовый проект, куда интегрировалась библиотека был не самого большого масштаба и сложности, вследствие чего оценка могла принять совсем другое значение как в большую, так и меньшую сторону [16].

Практическая значимость работы заключается в том, что разработанная библиотека может быть интегрирована в любые проекты, использующие React, и адаптирована под конкретные задачи без необходимости создания компонентов «с нуля». Она повышает производительность команды, снижает количество ошибок за счёт типизации и тестирования, упрощает сопровождение проекта и способствует стандартизации разработки UI [7].

Таким образом, поставленные цели выпускной работы достигнуты в полном объёме. Разработанная библиотека соответствует современным требованиям к качеству интерфейсов и может быть основой для построения внутренней дизайн-системы компании. Результаты работы могут быть использованы как в учебных целях, так и в реальной коммерческой практике при разработке и масштабировании веб-приложений.

Список используемой литературы

1. Абрамов А.И. Современная разработка на React и Redux. – СПб.: Питер, 2022.
2. Баранов Д.А. Тестирование веб-приложений: методологии и инструменты. – СПб.: Питер, 2021.
3. Глухов А.В. Git и командная разработка. – М.: БХВ-Петербург, 2020..
4. Голованов И.Ю. Тестирование в JavaScript. – М.: Эксмо, 2020.
5. Емельянов А.В. Разработка веб-приложений на TypeScript и React. – СПб.: Питер, 2022.
6. Жуков С.Н. Технологии разработки с использованием React и TypeScript. – СПб.: Питер, 2022.
7. Иванов П.С. Проектирование пользовательских интерфейсов. – М.: ДМК Пресс, 2021.
8. Климов С.Ю. Проектирование и разработка интерфейсов с применением современных веб-технологий. – М.: Инфра-М, 2023.
9. Кузнецов С.О. TypeScript для профессионалов. – М.: ДМК Пресс, 2022.
10. Лебедев А.А. JavaScript. Подробное руководство. – М.: Вильямс, 2020.
11. Матвеев И.А. Практика модульной архитектуры во фронтенде. – М.: ДМК Пресс, 2021.
12. Мищенко Д.В. UI/UX-дизайн: от идеи до реализации. – М.: БХВ-Петербург, 2022.
13. Никитин Р.А. Инженерия программного обеспечения. – СПб.: Питер, 2021.
14. Овчинников Е.М. Frontend-разработка: паттерны проектирования. – СПб.: БХВ-Петербург, 2021.
15. Пастухова Н.С. Разработка интерфейсов на React. – СПб.: БХВ-Петербург, 2021.

16. Поляков Д.Ю. Веб-компоненты и архитектура SPA. – М.: ДМК Пресс, 2021.

17. Скуратов А.В. Архитектура фронтенда: принципы и практика. – М.: Наука, 2020.

18. Соловьев А.В. Принципы компонентной архитектуры. – М.: Диалектика, 2020.

19. Сорокин Е.С. Автоматизация сборки проектов: Webpack и Vite. – М.: ДМК Пресс, 2023.

20. Черкасов И.Е. Основы веб-разработки. – СПб.: Питер, 2022.

21. Bierman G., Abadi M. Understanding TypeScript. – Microsoft Research, 2021.

22. Crockford D. JavaScript: The Good Parts. – O’Reilly Media, 2008.

23. Freeman E., Robson E. Head First Design Patterns. – O’Reilly Media, 2020.

24. Johnson A. React Design Patterns and Best Practices. – Packt Publishing, 2018.

25. Larman C. Applying UML and Patterns. – Prentice Hall, 2004.

26. Martin R.C. Clean Architecture: A Craftsman's Guide to Software Structure and Design. – Prentice Hall, 2017.

27. Wieruch R. The Road to React. – Self-Published, 2021.