МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение высшего образования
«Дальневосточный федеральный университет»
(ДВФУ)

ИНСТИТУТ МАТЕМАТИКИ И КОМПЬЮТЕРНЫХ ТЕХНОЛОГИЙ

Департамент математического и компьютерного моделирования

О Т Ч Е Т

о разработке шаблонизатора кода для работы с методологией FSD

направление подготовки 09.03.03 «Прикладная информатика»
профиль «Прикладная информатика в компьютерном дизайне»

Выполнили студенты
гр. Б9121-09.03.03пикд
_____ _____
_____ _____
_____ _____
(Ф.И.О.) (подпись)
Руководитель работы
_____
(должность, уч.звание)
_____
(Ф.И.О.) (подпись)
«_____»_____2023г.

Отчет защищен:
с оценкой
_____

Рег. № _____

«____»_____2023 г.

г. Владивосток
2023

Олавление

1. Введение

1.1. Глоссарий

1. **Feature-Sliced Design (FSD)** — это архитектурная методология для проектирования frontend-приложений. Проще говоря, это свод правил и соглашений по организации кода. Главная цель методологии — сделать проект понятным и структурированным, особенно в условиях регулярного изменения требований бизнеса. [1]
2. **Layers (слои)** - первый уровень разделения в FSD: по тому, как сильно модуль влияет на приложение. Чем ниже слой, тем опаснее вносить в него изменения.
3. **Slices (слайсы)** - второй уровень разделения в FSD: по конкретной функциональности бизнес-логики.
4. **Segments (сегменты)** - третий уровень разделения в FSD: по назначению модуля в коде и реализации.
5. **Node package manager (npm) –** Пакетный менеджер для JavaScript, разрабатываемый npm, Inc. [4]
6. **Node.js –** Окружение для запуска JavaScript кода. [5]

1.2. Описание предметной области

Архитектура приложений, в том числе и архитектура фронтенд-приложений постоянно развивается. Разрабатываются новые подходы, которые дополняют и переосмысляют существующие.

В конце 20 века были популярны архитектуры MV*: MVC, MVP, MVVM. Хотя и сейчас на их основе базируется большое количество фреймворков и проектов. Так же эти методологии лежат в основе большинства современных подходов к построению архитектуры приложений. Основная идея MV* в разделении кодовой базы по типам программных сущностей (Model, View, Controller, View Model, Presentation). Такого разделения рано или поздно начнет не хватать.

19XX – 2000 годы. Появляются и активно используются SoC, KISS, YAGNI, GRASP, DRY, SOLID – принципы проектирования систем. Большая проблема в том, что при применении всех этих принципов вместе, они начинают конфликтовать друг с другом. Кроме того, у этих принципов есть различные трактовки, и каждый программист в итоге понимает их по-разному.

19XX – 2010 годы. Component Based Development. Идея этого подхода в делении приложения на компоненты. Но возникает проблема в том, что этот подход не объясняет, как связывать компоненты воедино.

2003–2012 годы. DDD, Onion, Clean Architecture. Это методологии, в которых впервые учитывается предметная область. Их главная проблема в том, что нет единой интерпретации, по которой можно «готовить» архитектуру на основе выбранной методологии. Кроме того, в этих методологиях UI является внешней зависимостью, которая может легко меняться. В последнее время на клиенте слишком много логики, чтобы считать frontend внешней зависимостью.

2013 год – Atomic Design. Методология, которая призывает разделять UI по сложности, на: атомы, молекулы, организмы, шаблоны и страницы. Но этот подход не отвечает на вопросы: «Где хранить данные?» и «Как грамотно распределять данные по проекту?».

2010–2016 годы. Feature Based, Vertical Slices. Подходы, в которых все ресурсы разделяются не по типу, а по функциональному осмыслению. Но для этих подходов существует мало примеров и непонятно, как они ведут себя на больших проектах.

2018–2020 годы – Feature Slices (v1), Feature Driven Architecture. Sergey Sova и Oleg Isonen представляют манифесты о том, как преодолеть «детство» фронтенда, как перенять лучшие практики из существующих подходов и решить проблему масштабирования и растущей сложности проектов. Это возымело успех – начало образовываться сообщество, начали публиковаться статьи

В 2021 году появился Feature Sliced Design (v2) (FSD).

Люди, которые входят(входили 16.12.2021) в основную команду разработки FSD:

- Sergey Sova (@sergeysova)
- Karina Akaia (@RinAkaia)
- Ilya Azin (@azinit)
- Alexandr Horoshin (@AlexandrHoroshin)
- Ilya Agarkov (@ilyaagarkov)
- Ivan Chernenko (@spotsccc)
- Anton Medvedev (@unordinarty)
- Evgeny Podgaestskiy (@epodgaetskiy)

Методология FSD сосредоточена на разделении кода frontend-приложения на слои, слайсы и сегменты и на правильном их связывании. Это позволяет:

- Упростить поддержку существующей кодовой базы и добавления нового функционала.
- Оптимизировать зависимости различных слоев, что делает работу приложения предсказуемой, а фрагменты кода становятся легкими для переиспользования.

В FSD 7 слоев: shared, entities, features, widgets, pages, processes, app. На Рисунок 1 самый нижний слой расположен справа, а самый верхний – слева.

Рисунок 1. Иллюстрация слоев FSD [3]

- Каждый слой может использовать (импортировать) только нижележащие слои.
- Чем выше расположен слой, тем выше уровень его ответственности и знаний о других слоях.
- Чем ниже расположен слой – тем он больше используется в верхних слоях, а значит и тем больше опасности вносить в него изменения.

В Таблица 1 показано, какие слои можно использовать в других слоях.

Таблица 1. Слои FSD и использование их другими слоями [2]

Слой	Может использовать	Может быть использован
app	shared, entities, features, widgets, pages, processes	-
processes	shared, entities, features, widgets, pages	app
pages	shared, entities, features, widgets	processes, app
widgets	shared, entities, features	pages, processes, app
features	shared, entities	widgets, pages, processes, app
entities	shared	features, widgets, pages, processes, app
shared	-	entities, features, widgets, pages, processes, app

При этом рекомендуется, сначала выделять основные слои, которые актуальны для почти любого приложения [2]:

- **app** – для инициализирующей логики приложения.
- **pages** – для экранов приложений.
- **shared** – для абстрактной общеиспользуемой логики (UIKIT, helpers, API).

А затем, выделять остальные слои по мере необходимости:

- **widgets** – если логика на страницах начинает разрастаться и дублироваться.
- **entities** – если в проекте все равно разрастается количество деунифицированной логики.
- **features** – если в проекте становится сложно отследить начало и конец пользовательских сценариев, и контролировать их.
- **processes** – если разрастается много надстраничной «сквозной» логики.

В каждом слое находятся слайсы. Методология почти не влияет на этот уровень.

Внутри каждого слайса находятся сегменты. Каждый сегмент представляет из себя набор привычных наборов абстракций, при разработке ПО. Каждый сегмент может быть, как директорией с несколькими файлами, так и отдельным файлом, в зависимости от сложности реализуемого слайса. На Рисунок 2 показан пример сегментов, которые могут находиться внутри слайса.

```
{layer}/
    ├── {slice}/
    |   ├── ui/                    # UI-логика (components, ui-widgets, ...)
    |   ├── model/                 # Бизнес-логика (store, actions, effects, reducers, ...)
    |   ├── lib/                   # Инфраструктурная логика (utils/helpers)
    |   ├── config*/               # Конфигурация (проекта / слайса)
    |   └── api*/                  # Логика запросов к API (api instances, requests, ...)
```

Рисунок 2. Пример набора сегментов внутри слайса [2]

При всем этом, в зависимости от слоя, в слайсах может находиться разный набор сегментов. Это показано в Таблица 2.

Таблица 2. Наборы сегментов в зависимости от слоя [6]

Слой	Содержимое	Разрешенные сегменты
app	Не включает в себя слайсы и содержит логику инициализации	Имеющиеся сегменты не совсем подходят, а потому используются обычно /providers (/hoc, ...), /styles и т. д. Очень зависит от проекта и вряд ли решается методологией
processes	Слайсы внутри включают в себя только бизнес-логику, без отображения (1)	lib, model, (api)
pages	Слайсы внутри включают в себя ui- и model- композицию различных фичей для конкретной страницы	ui, lib, model, (api)

features	Слайсы внутри включают в себя композицию сущностей и реализацию БЛ в модели + отображение	ui, lib, model, (api)
entities	Слайсы внутри представляют разрозненный набор подмодулей для использования	ui, lib, model, (api)
shared	Содержит только инфраструктурную логику без БЛ (1)	ui, lib, api

Методологию FSD используют разные компании, такие как:

- REDMADROBOT
- Фоксфорд
- Яндекс
- ~/.space307
- PMP TECH
- UPTARGET co
- KODE
- FXDD
- VIGO и т.д.

Из-за большой и достаточно сложной структуры методологии, в процессе разработки производится большое количество рутинных действий и пишется много бойлерплейт-кода. Для удобного применения методологии FSD необходимы инструменты, которые позволят разработчикам быстро и эффективно создавать слайсы со всем необходимым бойлерплейт-кодом и гибко настраивать этот процесс. В этом отчете рассматривается разработка консольного приложения, которое решает эти задачи.

Проблемы:

- В данный момент «ручная» работа с методологией ощутимо замедляет процесс разработки, а также является монотонной.
- Отсутствие удобного и гибкого инструмента для генерации слайсов по методологии FSD.

1.3. Неформальная постановка задачи

В рамках проекта необходимо разработать инструмент командной строки, который будет способен генерировать слайсы в рабочей директории в соответствии с принципами методологии FSD, конфигурировать процесс генерации, обрабатывать пользовательские шаблоны, на основе которых и будет производиться генерация.

1.4. Обзор существующих решений

Критерии сравнения:

- Политика распространия − знаком “+” помечаются решения, поставляющиеся на безвозмездной основе, а знаком “-”, те решения, за использование которых требуется заплатить.
- Отсутствие завязанности на конкретном редакторе кода.
- Распространение посредством npm.
- Применимость во фронтенд-разработке.
- Покрытие задач, связанных с FSD.
- Открытость исходного кода.
- Готовность к использованию “из коробки”.

Таблица 3. Сравнение существующих решений

Название	Политика распространения	Отсутствие завязанности на конкретном редакторе	Распространение через npm	Применимость к фронтенд-разработке	Покрытие задач, связанных с FSD	Открытый исходный код	Готовность к использованию «из коробки»
JetBrains templates	+	-	-	+	+	-	+
Snippets Vs-code	+	-	-	+	-	+	+
nest-cli	+	+	+	-	-	+	+
codesmith	-	+	-	+	-	-	+
vgent	+	+	+	+	-	+	+
generate-react-cli	+	+	+	+	-	+	+
hygen	+	+	+	+	+	+	-
yeoman	+	+	+	+	+	+	-
feature-sliced-design cli	+	+	+	+	+	+	-
Emmet	+	+	-	+	-	+	+
Suipta	+	+	+	+	+	+	+

JetBrains Template. Очень удобное и конфигурируемое решение. Самая большая проблема – невозможность использования вне продуктов JetBrains.

Snippets VsCode. Удобное решение для генерации небольших фрагментов кода, но не получится создавать с помощью сниппетов файлы и директории.

Nest-cli. Хороший инструмент для генерации всего, что связано с Nest.js. Но это не применимо к фронтенд-разработке.

Codesmith. Платное решение. Исходя из описания на официальном сайте решает проблему.

Vgent. Генератор компонентов для Vue. Подходит только для Vue. Не решает задачи, связанные с FSD.

Generate-react-cli. Генератор компонентов для React. Подходит только для React. Не решает задачи, связанные с FSD.

Hygen. Фреймворк для микрогенерации. Для того, чтобы использовать нужно много конфигурировать. На нем можно было бы реализовать Suipta, но было выбрано другое решение.

Yeoman. Фреймворк для микрогенерации. Для того, чтобы использовать нужно много конфигурировать. На нем можно было бы реализовать Suipta, но было выбрано другое решение.

Feature-sliced-design cli. Находится в зачаточном состоянии и уже год не развивался.

Emmet. По своей сути поход на Snippets VsCode. Применяется для генерации html разметки.

Suipta. Берет в основу методологию FSD и учитывает различные сценарии использования. Является готовым к использованию из коробки, но также имеет возможность конфигурации.

2. Требования к окружению

Минимальны и полностью соответствуют системным требованиям NodeJS:

- ЦП с тактовой частотой от 1ГГц
- Оперативная память: 512Мб и выше
- Пространство на жестком диске: от 20Мб
- Платформа: Windows, MacOS, Linux
- Разрядность системы: x32/x64

Прочие требования:

- Аккаунт на Github для получения доступа к исходникам и документации по установке

3. Архитектура системы

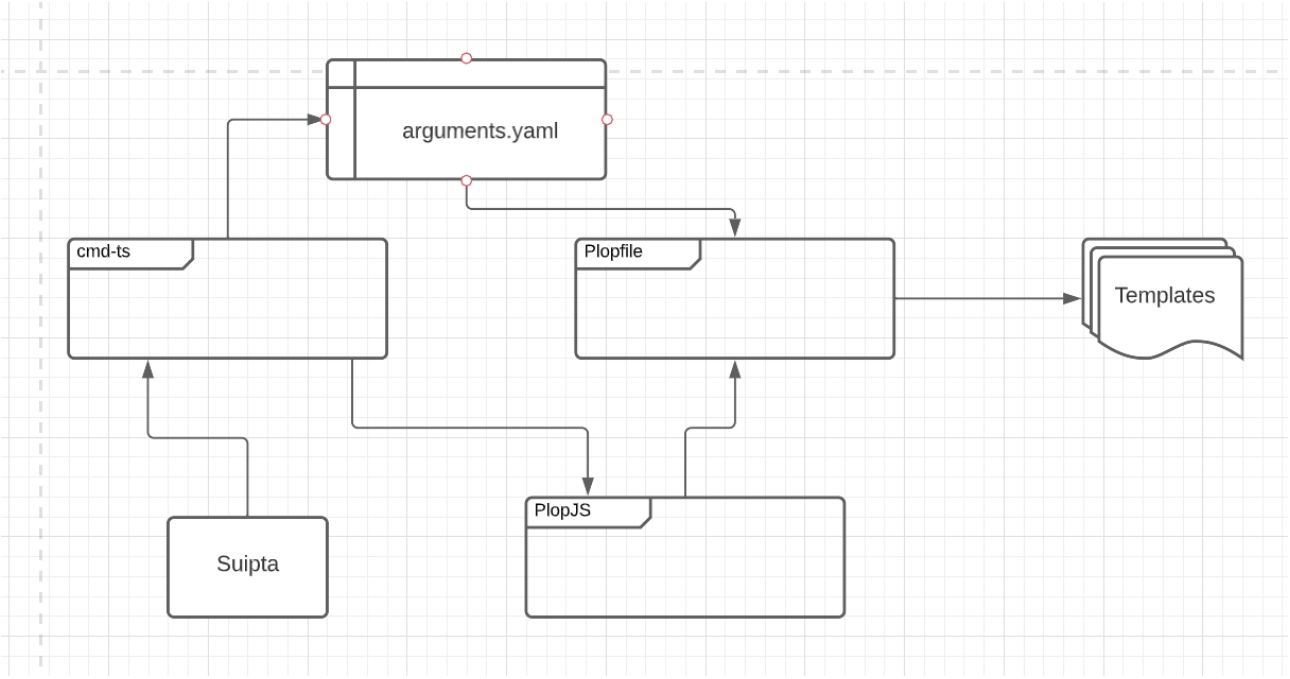

Рисунок 3. Диаграмма пакетов Suipta

Пояснение: пользователь запускает приложение посредством командной строки. Введенные аргументы записываются в файл **arguments.yaml** и читаются **Plopfile**. **Plopfile** является файлом конфигурации для библиотеки **Plop**. **Plop** отвечает непосредственно за генерацию шаблонов. Шаблоны определены в директории **Templates**.

4. Спецификация данных

4.1. Формат конфигурационного файла

Конфигурационный файл может быть написан в следующих форматах:

- Json
- Yml, Yaml

И может содержать следующие настройки:

1. Layers – список слоев
2. TemplatesDir – директория шаблонов
3. rootDir – корневая папка разрабатываемого проекта

4.2. Формат команды

suipta [список аргументов*]

Формат аргументов командной строки:

- Обязательные аргументы
    1. -l --layer [name = название слоя]
    2. -s --slice [name = название слайса] при условии, что указан -l или –layer
- Опциональные аргументы
    3. -h --help
    4. -c --config [path = путь до файла конфигурации]
    5. -m --model [effector | mobx = название библиотеки для управления состоянием] при условии, что указан -l или --layer и -s или --slice
    6. -- ui [react | solid ... = название ui библиотеки] при условии, что указан -l или --layer и -s или –slice

4.3. Формат шаблонов

Для создания собственных шаблонов:

1. Шаблоны помещаются в директории /templates (по умолчанию) или в заданной пользователем через конфиг [4.1 Формат конфигурационного файла].
2. Каждый шаблон является папкой с файлами. Папка имеет название одного из слоев из конфигурационного файла.
3. В шаблонах можно использовать следующие аргументы:
    a. slice – название слайса, который будет использоваться во время работы приложения, используя название, описанное в аргументе командной строки -s –slice [4.2 Формат аргументов командной строки].
    b. layer – название слоя.

    Для этого используется следующий синтаксис: {{helper_name slice | layer}}, где helper_name – название помощника (необязательный элемент, в случае его отсутствия аргумент будет вставлен без изменений). Помощники могут быть следующими: camelCase, snakeCase и другие [8].

5. Функциональные требования

Система должна:

1. Иметь возможность установки средствами пакетного менеджера npm
2. Обеспечивать взаимодействие как в интерактивном режиме, так и посредством обработки аргументов командной строки
3. Принимать аргументы командной строки [4.2 Формат аргументов командной строки]
4. Разделять аргументы на обязательные и опциональные и, соответственно, обрабатывать их в соответствии с приоритетами
5. При обработке конфига и аргументов, приоритет отдавать аргументам, затем конфигу, затем «заводским установкам»
6. При ошибке в конфиге выводить соответствующее сообщение при попытке использовать утилиту
7. Выводить ошибку при попытке создать уже существующий слайс
8. Информировать пользователя о природе и статусе ошибки
9. Создавать слайс в соответствии с пользовательской настройкой
10. Выводить информацию о созданном слайсе при успешном исполнении
11. При некорректной комбинации аргументов выводить ошибку

6. Требования к интерфейсу

- Интуитивность и логичность для пользователя, а также предоставление возможности использования естественного языка, синонимов, аббревиатур и других средств.
- Предоставление пользователю информации о доступных командах и опциях, а также об ошибках и результатах выполнения.
- Обеспечить удобочитаемость, посредством акцентирования внимания на процессах следующими способами
    - Цветовое выделение элементов
    - Отображение индикатора прогресса
- Поддержка интерактивного режимы работы. В интерактивном режиме пользователь может выбирать (вводить, в случае с названиями) команды по одной в консоли.
- Универсальность и совместимость с различными операционными системами и терминалами.
- Документированность и сопровождаемость, а также предоставление возможности получения справки и обратной связи.

7. Проект

Проект состоит из следующих модулей:

- Служебные файлы node js и pnpm
- Файлы для разработки
  - Линтер
  - Хуки для GIT
  - Прочие файлы для GIT
- Исходники проекта
- Тестирующая система
- Внешние библиотеки

7.1. Средства реализации

Стек языков программирования: JavaScript, TypeScript

В целях упрощения работы на проекте используются внешние библиотеки. Главные требования: простота в использовании, документируемость, поддержка разработчиками. Все используемые в приложении библиотеки удовлетворяют этим условиям. Был проведен сравнительный анализ, в результате которого в состав проекта входят:

- Cmd-ts – парсер аргументов командной строки
- PlopJS – средство создания файлов и директорий из заготовленных шаблонов

Хранилищем для исходников служит GitHub, как самое распространенное средство для вещей такого рода. Плюсом в комплекте есть поддержка CI/CD (Continuous delivery/Continuous integration)

На проекте используется стандарт Conventional commits для упрощения чтения истории разработки. CLI – утилита Commitizen позволяет быстро создать коммит, удовлетворяющий стандарту conventional commit, посредством консольного интерфейса с подсказками.

7.2. Проблемы и решения

В процессе разработки выявлены следующие проблемы:

1 Особенности plop и plopfile не позволяли получить доступ к аргументам.

Решение: использование дополнительного файла **arguments.yaml**

2 Внедрение кастомных аргументов командной строки.
Решение: использование более низкоуровневой версии plop (**node-plop**)

7.3. Стандарт кодирования

В целях соблюдения однообразности кодирования, на проекте используются ESLint и Prettier с пользовательскими настройками конфигурации.

ESLint

ESLint - это инструмент для статического анализа кода на JavaScript, который позволяет выявлять потенциальные ошибки и проблемы в коде, а также помогает обеспечивать единообразный стиль кодирования.

В конфигурационном файле **.eslint.cjs** определены настройки правил, которые определяют, какие ошибки будут проверяться в коде. Каждое правило определяет определенный аспект кода, который нужно проверить, и описывает, как должен быть написан этот аспект, чтобы соответствовать правилу.

- "indent" - определяет, сколько пробелов должно быть в отступе.Текущее значение: 2
- "linebreak-style" - определяет, какие символы перевода строки должны использоваться. Текущее значение:должны использоваться символы перевода строки Unix (LF).
- "quotes" - определяет, какие кавычки должны использоваться для строк. Текущее значение: используются одинарные кавычки.
- "semi" - определяет, должны ли точки с запятой использоваться в конце выражений. Текущее значение: не используются.
- "camelcase" - определяет, должны ли имена переменных и свойств объектов быть написаны в стиле camelCase (с первой буквой каждого слова в верхнем регистре, кроме первого слова). Текущее значение: нет.
- "eqeqeq" - определяет, должны ли операторы сравнения (== и !=) использоваться вместо операторов строгого сравнения (=== и !==). Текущее значение: нет.
- "prefer-const" - определяет, следует ли использовать ключевое слово const для переменных, которые не переназначаются.

Кроме того, в этом конфигурационном файле указано, что используется плагин "@typescript-eslint", который позволяет использовать ESLint с TypeScript, и задан парсер "@typescript-eslint/parser", который позволяет ESLint правильно обрабатывать TypeScript-код.

Prettier

Prettier - это инструмент, который позволяет автоматически форматировать код, чтобы он соответствовал определенным стандартам и правилам стиля кодирования.

В конфигурационном файле **.prettierrc.cjs** заданы настройки форматирования, которые определяют, как должен быть отформатирован код. Некоторые из настроек, которые заданы в данном файле, включают:

- "trailingComma" - определяет, должна ли запятая стоять в конце последнего элемента массива или объекта. Текущее значение: запятая должна стоять в соответствии со стандартом ES5.
- "tabWidth" - определяет, сколько пробелов должно быть использовано для отступов при использовании табуляции. Текущее значение: 2 пробела.
- "semi" - определяет, должны ли точки с запятой использоваться в конце выражений. Текущее значение: не используется
- "singleQuote" - определяет, должны ли использоваться одинарные кавычки для строковых литералов. Текущее значение: одинарные кавычки.
- "bracketSpacing" - определяет, должны ли использоваться пробелы вокруг скобок. Текущее значение: используются
- "arrowParens" - определяет, должны ли использоваться скобки при определении стрелочных функций. Текущее значение: если возможно, то скобки следует опускать.

Все эти настройки помогают создавать более читабельный и единообразный код.

7.4. Проект интерфейса

Взаимодействие пользователя с приложением осуществляется с помощью консольного интерфейса, подразумевающего классический ввод команд и интерктивный

режим в виде набора списков, итерация по которым осуществляется с помощью клавиш стрелок.

Каждая команда начинается с ключевого слова **suipta,** за которой следуют команды конфигурации, описанные выше.

Пример: **suipta -l layer -s slice –** создает слой с названием layer и слайс slice.

8. Реализация и тестирование

8.1. Физические характеристики системы:

- Объем написанного кода:
  - Вес: 450Кбайт
  - Количество строк кода: более 600 (по данным Github Gloc)
- Количество модулей: 5
- Количество сделанных изменений (коммитов): 60
- Количество автоматических тестов:
- Ссылка на репозиторий с исходным кодом: https://github.com/suipta/suipta
- Фактические затраты оперативной памяти: не удалось измерить ввиду слишком малых значений

8.2. Методика тестирования

Тестирование проводилось посредством Unit тестов с использованием фреймворка vitest.

Описание тестовой системы

В директории test-env находятся файлы с исходным кодом, тестирующим систему. Каждый такой файл содержит в себе набор тестов для конкретного слоя. Каждый тест представляет собой проверку генерации шаблона по заданным аргументам. Проверка производится путем сравнения сгенерированного шаблона, с эталоном. Эталонные результаты находятся в директории tests/result-examples. В свою очередь сгенерированные в процессе теста шаблоны попадают в директорию tests/src.

Пример теста

```
it('with model argument = effector', async () => {
  const slice = 'with-effector'
  await suiptaHandler({
    generator: 'slice',
    layer: 'entities',
    slice,
    model: 'effector',
  })

  const slicePath = path.join(generationPath, 'entities', slice)
  expect(fs.existsSync(slicePath)).toBeTruthy()
  testEntities(slice)
})
```

Рисунок 4 Пример теста

По приведенному выше листингу видно, что он проверяет корректность генерации слоя **entities**, слайса с названием **with-effector** и используемой библиотекой управления состоянием **effector**. Далее шаблон сравнивается с эталоном, формируется результат.

Результаты тестирования можно узнать, выполнив команду **pnpm test.**

Важно отметить, что в случае хотя бы одного проваленного теста, система не даст разработчику сделать коммит и как следствие опубликовать некорректные изменения. Это позволяет избежать некоторых ошибок при разработке.

9. Заключение

Таким образом, в процессе разработки этого проекта был проведен анализ предметной области, конкурентных и схожих решений, составлены требования к программному продукту, построена архитектура проекта, определены языки программирования и библиотеки, выявлены проблемы и их решения, создан генератор шаблонов для методологии FSD, проведено Unit тестирование.

Разработанная в рамках данной работе система существенно упростит разработку frontend приложений на проектах, использующих FSD как главную архитектурную методологию, за счет максимально простого и понятного консольного интерфейса и поставки через пакетный менеджер npm.

Дальнейшее развитие системы будет направлено на доработку удобства использования и будет зависеть в первую очередь от отклика конечных пользователей.

10. Источники

1. https://feature-sliced.design/ru/docs
2. https://feature-sliced.design/ru/docs/reference/units
3. https://feature-sliced.design/ru/docs/get-started/tutorial
4. https://ru.wikipedia.org/wiki/Npm
5. https://ru.wikipedia.org/wiki/Node.js
6. https://feature-sliced.design/ru/docs/reference/units/decomposition
7. https://github.com/plopjs/plop
8. https://systemreq.ru/node-js/
9. https://google.github.io/styleguide/tsguide.html
10. https://habr.com/ru/companies/inDrive/articles/693768/
11. https://github.com/feature-sliced/documentation
12. https://www.youtube.com/watch?v=af-PD2yIUiU
13. https://medium.com/@fed4wet/feature-sliced-design-modern-architectural-methodology-on-angular-d0ef705ef598
14. https://okusov.ru/metodologiya-feature-sliced-idealnyj-sposob-strukturirovat-rastushij-proekt
15. https://plugins.jetbrains.com/plugin/21638-feature-sliced-design-helper
16. https://soer.pro/codelabs/arch_stream_23/index.html?index=..%2F..index
17. https://hackernoon.com/understanding-feature-sliced-design-benefits-and-real-code-examples
18. https://effector.dev/ru/
19. https://github.com/effector/effector
20. https://habr.com/ru/companies/domclick/articles/532016/
21. https://mobx.js.org/README.html
22. https://www.npmjs.com/
23. https://docs.npmjs.com/
24. https://pnpm.io/
25. https://pnpm.io/pnpm-cli
26. https://github.com/pnpm/pnpm